A Framework for Aggregation of Presence Information Based on User-Provisioned Rules

von Olaf Bergmann

Dissertation

zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften

— Dr.-Ing. —

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)

der Universität Bremen

im November 2006

Datum des Promotionskolloquiums: 01.06.2007

Gutachter: Prof. Dr.-Ing. Ute Bormann
(Universität Bremen)

Prof. Dr.-Ing. Jörg Ott
(Helsinki University of Technology)

Abstract

Presence systems are widely used to get aware of other users' availability and willingness to communicate before actually contacting them. While early presence services offered only text-based communication and relied on the users to enter appropriate status descriptions, modern applications have evolved that integrate multi-media communication and facilitate automatic detection of status changes. To achieve this, unstructured data provided from various sources is aggregated and set into relation with user-specific context information.

As most of the algorithms used for automatic status inference require extensive information acquisition to construct an initial rule set describing the user's habits and activities, several research groups have investigated mechanisms for transforming low-level sensor data into abstract presence descriptions.

In this thesis, we enhance the sensor-fusion approach to aggregate abstract presence descriptions from multiple *presence-zones* a single user might be part of. We propose a vocabulary based on ECMAScript for controlling the aggregation process. With this language, users can specify how status update notifications generated by their personal presence sources should be aggregated at a presence server, and how this information should be published to subscribed watchers. The latter allows for explicitly choosing specific communication channels offered to a particular set of users. For example, the mobile phone might be offered as a means for communicating with family members while this media is not offered to any other group of subscribers.

To prove the applicability of our approach in specific environments, we have implemented a presence server that supports the evaluation of presence aggregation scripts as defined in this thesis. In addition, the standardized presence format was extended by a set of attributes that describe the degrading exactness of presence values over time. Based on this information, a user's presence status can be calculated by a presence server even if no recent status updates have been received.

Danksagung

Die hier dargestellte Forschungsarbeit wurde am Technologie-Zentrum Informatik (TZI) der Universität Bremen in der Arbeitsgruppe Rechnernetze durchgeführt. Die Betreuung erfolgte durch Prof. Dr.-Ing. Ute Bormann und wurde unterstützt von Prof. Dr.-Ing. Jörg Ott vom Networking Laboratory an der Helsinki University of Technology.

Prof. Dr.-Ing. Ute Bormann hat meine wissenschaftliche Karriere seit dem Eintritt in das Hauptstudium begleitet und gefördert und dabei mein Interesse für die Themengebiete der *Transformation von strukturierten Dokumenten* und *Internettechnologien* geweckt. Im Vordergrund standen dabei im wesentlichen die Konzepte und Architekturen, deren Verständnis für die Entwicklung komplexer Systeme unabdingbar ist. Prof. Dr.-Ing. Jörg Ott hat diese Linie fortgeführt und war mir insbesondere auf dem Gebiet der Multimedia-Kommunikation über das Session Initiation Protocol (SIP) ein ausgezeichneter Lehrer. Im Forschungsprojekt MECCANO habe ich gemeinsam mit ihm die erste Implementierung eines SIP-Proxys begonnen und diesen in späteren Projekten nach mehreren Umstrukturierungen bis zum heutigen Stand fortgeführt. Dabei profitierte ich von den tiefen Detailkenntnissen, die Jörg Ott aus jahrelanger Arbeit für die Standardisierung von H.323 und SIP einbrachte und die eine kritische Sichtweise auch auf Fehlentwicklungen bei der Protokollentwicklung förderten.

Neben diesen beiden Personen waren vor allem die Mitarbeiter der AG Rechnernetze am TZI eine große Unterstützung. Mein besonderer Dank gilt Prof. Dr.-Ing. Carsten Bormann für seine detaillierten und kritischen Anmerkungen, mit denen er einen großen Anteil an der wissenschaftlichen Qualität der vorliegenden Arbeit hat. Seiner Initiative ist es zudem zu verdanken, daß begleitend zu meiner Forschungsarbeit über Echtzeitkommunikation im Internet mehrere Lehrbücher über Webtechnologien und Dokumenttransformation entstanden sind. Die Erfahrung aus diesen Buchprojekten war ein wesentlicher Aspekt, der die Fertigstellung der vorliegenden Arbeit erst ermöglichte. Prof. Dr.-Ing. Carsten Bormann und Dr.-Ing. Dirk Kutscher waren es auch, die mir die notwendigen Programmiertechniken und Entwurfsmuster zur Entwicklung großer Softwaresysteme vermittelten. Der von Dr. Kutscher entwickelte Message Bus wurde von mir in zahlreichen Projekten erfolgreich eingesetzt und stellte somit seine Praxistauglichkeit als schnell erlernbare Middleware zur Entkopplung von Anwendungen eindrucksvoll unter Beweis.

Den Mitarbeitern der AG Rechnernetze danke ich für sieben Jahre in einem angenehmen und produktiven Umfeld, in dem neue Ideen konstruktiv hinterfragt wurden und sich so weiterentwickeln konnten. Besondere Unterstüzung fand ich durch Eilert Brinkmann, der den TLS-Server zu meinem SIP-Stack beisteuerte und die Idee für die Generalisierung des Message-Parsers lieferte. In den Projekten DTI, PASST und GEOCOOP konnte ich unter anderem mit Andreas Büsching zusammenarbeiten, der die von mir verwendete DDA-Implementierung vorantrieb und Mbus-basierte Sensoren als Quellen von Presence Informationen entwickelte. Eine besondere Leistung ist die Portierung der Mbus-Bibliothek auf die exotische Hardware eines handelsüblichen IP-Telefons. Des weiteren entwickelte er in Zusammenarbeit mit Dirk Meyer eine zweite Implementierung für Mbus-RPCs, die ich als Gegenpart zum Testen meiner Referenzimplementierung *libmbusapp* verwenden konnte. Als Beta-Tester meiner SIP-Software stellten sich Tobias Hartmann, Sönke Schwardt und Kevin Loos der Herausforderung und lieferten nicht nur wertvolles Feedback über die Benutzbarkeit der Bibliotheken, sondern un-

terstützten tatkräftig die Suche und Behebung von Fehlern. Dr. Volker Wittpahl hielt nach meinem Verlassen der Universität stets den Kontakt aufrecht und gab nützliche Tips zur Systematisierung der Arbeit. Als stetiger Mahner übernahm er zuletzt die Aufgaben des schlechten Gewissens, und beschleunigte so die Fertigstellung dieser Arbeit.

Zum Abschluß sei auch noch denjenigen gedankt, die über Jahre unter der Fertigstellung dieser Arbeit zu leiden hatten und die wissentlich oder unwissentlich zu ihrem Gelingen beigetragen haben. Neben zahlreichen Freunden, die immer ein aufmunterndes Wort fanden, war es vor allem meine Familie, die immer moralischen Beistand geleistet hat und mir in der entscheidenden Phase den Rücken freihielt. Ohne diese Unterstützung wäre eine Fertigstellung der Arbeit nicht möglich gewesen.

Kiel, im November 2006

Contents

1 Introduction **1**
- 1.1 Automatic Presence Status Determination . . . 3
- 1.2 The Need for Multiple Abstraction Levels . . . 5
- 1.3 Aging of Presence Information . . . 7
- 1.4 Our Contribution . . . 9
- 1.5 Structure of this Thesis . . . 11

2 The Role of Presence Systems for Interpersonal Communication **13**
- 2.1 Historical and Technical Background . . . 13
  - 2.1.1 Evolution of Network Technologies . . . 14
  - 2.1.2 Conferencing Applications . . . 16
  - 2.1.3 Event Notification Systems . . . 17
- 2.2 Theoretical Foundations . . . 18
- 2.3 Meaning of “Presence” . . . 23
- 2.4 Presence-Aware Applications: Two Examples . . . 25
  - 2.4.1 Telepresence Systems . . . 25
  - 2.4.2 Location-based Services . . . 28
- 2.5 On User Acceptance of CSCW Systems . . . 32
- 2.6 A Use Case for User-Controlled Presence Aggregation . . . 34
- 2.7 Summary . . . 36

3 Requirements Analysis **39**
- 3.1 Guidelines for Presence Application Design . . . 39
  - 3.1.1 Minimizing Information Overload . . . 40
    - 3.1.1.1 Limiting the Notification Rate . . . 41
    - 3.1.1.2 Aggregation of Notifications . . . 43
    - 3.1.1.3 Prediction of Availability . . . 45
  - 3.1.2 Patterns of Interpersonal Communication . . . 47
    - 3.1.2.1 Establishing Common Ground . . . 47
    - 3.1.2.2 Media Selection . . . 48
    - 3.1.2.3 The Role of Presence Services . . . 49
  - 3.1.3 Summary . . . 50
- 3.2 Functional Requirements . . . 53
  - 3.2.1 Local Presence Environment . . . 56
  - 3.2.2 A Data Format for Presence Aggregation . . . 57
  - 3.2.3 The Presence Aggregation Process . . . 59
  - 3.2.4 An Aggregation Language for Presence Information . . . 59
  - 3.2.5 User-Specific Configuration . . . 63
- 3.3 Distribution of Presence Information . . . 63
  - 3.3.1 Internet-scale Event Notification . . . 63
  - 3.3.2 The Session Initiation Protocol (SIP) . . . 66

3.3.2.1 The Basic SIP Architecture . . . 67
3.3.2.2 Application-layer Message Routing . . . 68
3.3.2.3 SIP Event Notification . . . 69
3.3.2.4 Aggregation of Events . . . 71
3.3.2.5 Summary and Evaluation . . . 73
3.3.3 Alternative Distribution Architectures . . . 75
3.3.3.1 The Extensible Messaging and Presence Protocol (XMPP) . . . 75
3.3.3.2 Content-Addressable Networks (CAN) . . . 78
3.3.3.3 Summary . . . 79
3.4 Security . . . 80
3.4.1 Goals . . . 81
3.4.2 Threat Analysis . . . 81
3.5 Summary . . . 84

4 Related Work **87**
4.1 Context-Awareness . . . 88
4.1.1 Solar . . . 89
4.1.2 Technology for Enabling Awareness (TEA) . . . 90
4.1.3 Mobile Interactive Space . . . 92
4.1.4 Context-aware Communication Services . . . 94
4.1.5 Evaluation . . . 97
4.2 Presence Data Models . . . 98
4.2.1 The Microsoft Windows Messenger Presence Format . . . 99
4.2.2 Presence Information Data Format (PIDF) . . . 100
4.2.2.1 XML Syntax . . . 101
4.2.2.2 Extensions . . . 103
4.2.3 Extensible Messaging and Presence Protocol (XMPP) . . . 107
4.2.4 Evaluation . . . 109
4.3 Specification Languages for Data Aggregation . . . 110
4.3.1 Document Transformation . . . 111
4.3.2 Policy-based Filters . . . 113
4.3.3 Evaluation . . . 115
4.4 Summary . . . 116

5 An Architecture for a Distributed Presence Aggregation Service **119**
5.1 Overview . . . 120
5.2 Processing Model for Script-based Aggregation . . . 123
5.2.1 Basic Data Model . . . 124
5.2.2 Update Notifications . . . 125
5.2.3 Handling Inexact Status Descriptions . . . 126
5.2.4 Event History . . . 129
5.2.5 Transformations . . . 130
5.2.6 Output Generation . . . 131
5.3 Enhanced Presence Information Data Format . . . 131
5.4 The Local Presence Environment . . . 135
5.4.1 Mbus Entities . . . 136

5.4.2 Addressing Scheme . . . 137
5.4.3 Relevant Mbus Interaction Models . . . 140
5.4.4 A Presence-specific Mbus Command Set . . . 141
5.4.4.1 Publication of Presence Events . . . 142
5.4.4.2 Managing Subscriptions . . . 148
5.4.4.3 Controlling the Aggregation Engine . . . 150
5.5 A Policy Framework for Presence Aggregation . . . 157
5.6 Summary . . . 159

6 Using ECMAScript for Presence Aggregation 161
6.1 Language Characteristics . . . 161
6.1.1 Set . . . 162
6.1.2 Associative Set . . . 163
6.2 Native Objects . . . 164
6.2.1 Abstract Representation of Presence Information Documents . . . 164
6.2.1.1 Presentity . . . 164
6.2.1.2 Channel . . . 165
6.2.1.3 Attribute . . . 166
6.2.2 System-generated Events . . . 168
6.2.2.1 Timer . . . 168
6.2.2.2 Trigger . . . 169
6.3 Language Semantics . . . 170
6.3.1 Script Processing . . . 170
6.3.2 Runtime Environment . . . 172
6.3.2.1 History . . . 172
6.3.2.2 System Configuration Parameters . . . 173
6.4 Example Specifications . . . 173
6.5 Summary . . . 175

7 Implementation and Evaluation 177
7.1 Implementation Considerations . . . 177
7.1.1 Layered Architecture . . . 178
7.1.2 Server Abstraction Layer . . . 179
7.2 Architectural Components . . . 184
7.2.1 Local Presence Environment . . . 185
7.2.2 Presence Aggregation Module (PAM) . . . 187
7.2.2.1 The Aggregation Process . . . 187
7.2.2.2 Building the Internal Representation of Presence Documents 190
7.2.2.3 Evaluation of Aggregation Specifications . . . 193
7.2.2.4 Generating XML Output . . . 196
7.2.3 Presence Distribution Infrastructure . . . 198
7.2.3.1 Handling of SIP Transactions . . . 199
7.2.3.2 Application Logic . . . 199
7.3 Evaluation . . . 200
7.3.1 Test Scenario . . . 201
7.3.1.1 Abstract Setup . . . 202

7.3.1.2 Used Hardware and Software Components . . . 203
7.3.1.3 Authoring and Management of Aggregation Specifications . 205
7.3.1.4 Sample Aggregation Specification . . . 205
7.3.1.5 Lessons Learned . . . 209
7.3.2 Review of the Initial Requirements . . . 210
7.3.2.1 Functional Requirements . . . 210
7.3.2.2 Usability . . . 211
7.3.2.3 Summary . . . 213
7.3.3 Further Evaluation Strategies . . . 214
7.3.3.1 Field Studies . . . 215
7.3.3.2 Simulating the Presence Service . . . 216
7.3.3.3 Performance . . . 217
7.4 Summary . . . 219

8 Conclusions **221**
8.1 Conceptual Achievements . . . 222
8.2 Engineering Results . . . 224
8.2.1 Component Architecture for Local Presence Environments . . . 224
8.2.2 Object Model for Presence Documents . . . 225
8.2.3 A Language for Controlling Presence Aggregation . . . 225
8.2.4 Distribution of Aggregated Presence Information . . . 226
8.2.5 Presence-Based Call Routing . . . 227
8.2.6 Securing Presence Information . . . 227
8.3 Comparison With Other Approaches . . . 228
8.3.1 Solar . . . 228
8.3.2 Context-aware Communication Services . . . 229
8.4 Open Issues and Next Steps . . . 229
8.4.1 Improving the User Interface . . . 229
8.4.2 Generalized Document Transformations . . . 230
8.4.3 Location-Aware Resource Allocation for Conferencing Systems . . . 231
8.4.4 Simulation of Aggregation Rules Using OMNet++ . . . 232
8.4.5 Enhancing the System Performance . . . 232
8.4.6 Security . . . 233
8.5 Concluding Remarks . . . 233

A Example RPID Document **235**

B An Mbus Command Set for the Local Presence Environment **237**

C Example for an Enhanced PIDF Document **245**

D PAL Function Library **247**

E XML Transformation Specification for Presence Documents **251**
E.1 Transformation from XPIDF to PIDF . . . 251
E.2 Transformation from PIDF to XPIDF . . . 252

Bibliography **255**

List of Figures

1.1 Typical user interface of a modern messaging application . . . 2
1.2 Example sensors . . . 3
1.3 Example architecture for automatic presence status detection . . . 4
1.4 Multiple presence zones . . . 6
1.5 Status aggregation for multiple presence zones . . . 7
1.6 Applying decay functions to presence attributes . . . 8

2.1 Example view of the Portholes system (source: [Bux95b]) . . . 26
2.2 GEOCOOP demo scenario (source: [OKB+05]) . . . 30
2.3 GUI-based conference initiation (source: [OKB+05]) . . . 31
2.4 Presence zones for use case scenario . . . 35
2.5 Publication of distinct media channel descriptions . . . 36
2.6 Aggregation of multiple channel descriptions . . . 37

3.1 Server-based limitation of the update notification rate . . . 41
3.2 A presence client subscribing to multiple devices . . . 50
3.3 Status aggregation for multiple presence zones . . . 55
3.4 Using a common presence data format for multi-step aggregation . . . 57
3.5 Aggregating multiple channel descriptions . . . 58
3.6 Distribution architecture for pull-based event notification . . . 65
3.7 Basic SIP architecture . . . 67
3.8 Adding `Via` headers in a SIP transaction . . . 68
3.9 Push-based communication in a SIP network . . . 70
3.10 Using multiple presence sources . . . 72
3.11 Dissemination of status changes . . . 72
3.12 Syntax-based composition of presence documents . . . 73
3.13 XMPP overlay network . . . 76
3.14 Example XMPP protocol flow . . . 77
3.15 Watcher-specific filtering of presence attributes . . . 84

4.1 The IETF presence data model . . . 103
4.2 Sample instantiation of abstract presence data model . . . 106
4.3 Mapping from PIDF to XMPP . . . 108
4.4 Different placements for user-provisioned aggregation rules . . . 111
4.5 Instantiation of a template rule . . . 112

5.1 Presence Aggregation in a global SIP network . . . 121
5.2 Subscribing the presence status of a busy user for automatic call completion . . . 122
5.3 Complete call when the subscribed user becomes available . . . 123
5.4 Server-controlled call completion to busy subscriber . . . 123
5.5 Example for the accumulated idle time of a single user . . . 125
5.6 Example for a decay function for status descriptions . . . 127

5.7 Different types of decay functions 129
5.8 Aggregation engines throttling status update notifications 136
5.9 Tuple-based Addressing in the Mbus Environment 138
5.10 Event notifications on the Mbus 144
5.11 Querying information from external resources 148

6.1 Aging of objects with `MAX_HISTORY == 3` 173

7.1 External interfaces of the presence aggregation server 178
7.2 Abstract schema of implementation layers 179
7.3 Partial class hierarchy from server abstraction layer 183
7.4 De-coupled implementation of a presence aggregation server 185
7.5 Mbus-communication of sensors within a local environment 186
7.6 Generic libraries of the Mbus implementation 187
7.7 Class hierarchy of PAM components 188
7.8 Class hierarchy of `PALInterpreter` 194
7.9 Generic libraries of the SIP implementation 199
7.10 Abstract architecture of a presence aggregation system 200
7.11 Aggregated information from a presence service 201
7.12 Abstract view of the demo scenario 202
7.13 Network setup 204
7.14 Activity diagram created from Awarenex presence information (source: [BTS+02]) 216
7.15 An example activity schedule (source: [Ran02]) 217

List of Tables

2.1 Application building blocks (source: [OKB+05]) 29

4.1 Presence states of the Microsoft Messenger application 100
4.2 XMPP presence states 107

5.1 Address tags for distinct sensor types 139
5.2 Command arguments for `presence.notify` 143
5.3 Return values for `presence.publish` 145
5.4 Command arguments for `presence.publish` 146
5.5 Command arguments for `presence.fetch` 147
5.6 Return values for `presence.fetch` 148
5.7 Command arguments for `presence.subscribe` 149
5.8 Options for `presence.subscribe` 149
5.9 Return values for `presence.subscribe` 150
5.10 Command arguments for `pam.parameter.set` 151
5.11 Command arguments for `pam.parameter.get` 152
5.12 Return values for `pam.parameter.get` 152
5.13 Return values for `pam.parameter.describe` 153
5.14 Command arguments for `pam.script.add` 153
5.15 Options for `pam.script.add` 154
5.16 Return values for `pam.script.add` 154
5.17 Command arguments for `pam.script.modify` 155
5.18 Return values for `pam.script.modify` 155
5.19 Command arguments for `pam.script.delete` 156
5.20 Return values for `pam.script.delete` 156
5.21 Command arguments for `pam.script.list` 157
5.22 Command arguments for `pam.script.fetch` 157

6.1 Properties of `Presentity` 165
6.2 Properties of `Channel` 166
6.3 Properties of `Attribute` 167
6.4 Properties of `Trigger` 169
6.5 Possible values for `Trigger.status` 169
6.6 Pre-defined authorization classes in PAL 171
6.7 Properties of the object `SYSTEM` 176

7.1 Reference benchmark for presence services 218
7.2 Performance benchmark for presence aggregation 218

Chapter 1

Introduction

Interpersonal communication is an important aspect of enterprise organization and thus can have significant impact on the employees' productivity if not handled efficiently. Recently, *instant messaging* (IM) and presence systems have been recognized to break the perceived predominance of email as the most important means of informal asynchronous communication. Studies show that especially teenagers and young professionals have adopted IM as Internet-based real-time text communication as a tool to manage their daily life in private as well as business. [GrPa02, ReGo03] When asked about reasons for using IM instead of traditional email, users expressed their preference for interactive ad-hoc communication over the more formalized message exchange that was inherited from postal mail. [NWB00] In a number of European countries a similar effect has been observed when the *Short Message Service* (SMS) of the GSM network became widely available with cheap end-devices.

One reason for IM being considered more interactive than email is the tight coupling of the messaging system with a presence service that shows whether the intended recipient of a message is able and willing to participate in a conversation. Additional information may be transmitted by the presence service, e.g. a textual description of the peer's current activity. Figure 1.1 shows the user interface of a typical presence-enabled messaging application. A pre-defined list of contacts is shown together with the current presence status of each contact if known. Several contacts may be clustered in groups to enhance readability.

Applications like this not only offer ad-hoc text messaging but also enable synchronous multimedia communication between human users. When a listed contact is marked as being available for communication, the local user who wants to start a conversation can select which communication media to use. For example, two-point audio communication between both peers would be the equivalent of a traditional phone call. The more detailed the information on available communication media and current activities are, the better a caller can select an appropriate timing and media for interactive communication, an important factor for establishing *common ground* between the participants. [McMo94, AAS03] This not only avoids annoying the user with improper timing of call attempts or inappropriate media being used therein but also improves system dependability as calls are more likely to be accepted by the called party. Especially in distributed organizations with sites being at different geographical locations, significant productivity gains are expected as a result of minimizing overhead in setting up communication.

Obviously, there is a trade-off between the called user's preference not to be disturbed at specific times—or by specific persons, respectively—and the amount of information that must be made available to achieve this goal. The common understanding of privacy in distributed

information systems is that the owner of sensitive data can control who is granted access and what the granularity of the revealed information is. Most existing presence services therefore provide a simple authorization mechanism that allows for a human user to accept or decline subscriptions to his presence status. Denial can also be made implicit, i.e. the system pretends acceptance but never sends actual presence data (sometimes this is called *polite blocking*).

To our opinion, this amount of user control is too limited for modern communication services as it is based on the equivalence of various media being offered for communication. In this thesis, we discuss an alternative architecture we have developed to enable users to define groups of subscribers with specific access permissions regarding the published presence information. A presence server managing the subscriptions on behalf of that particular user then must filter outgoing presence status descriptions to provide subscribers with only the data they are meant to see.

Figure 1.1: Typical user interface of a modern messaging application

Once users are given explicit control over their presence information being published it will be possible to control the *aggregation* of status descriptions as well. Aggregation is necessary when multiple sources contribute to a user's unique presence status record and thus have to be merged somehow. It "comprises collection of high volumes of raw data from data sources, composition if the raw data into less voluminous refined data, and timely delivery of the refined data to applications" [CPT+01]. Figure 1.2 gives an example of various devices that could be used as data sources to provide information on the presence of a particular person. Among these are low-level sensors such as thermometers or passive infrared detectors, as well as more intelligent devices such as mobile phones with built-in GPS receivers or notebook computers that run dedicated programs for collecting activity data. While the first class of sensors requires additional systems that connects the proprietary access protocol of the particular sensor (e.g. using an RS-232 serial port) with the more generic protocol used by the personal presence server for data collection, the second class of sensors can be configured to talk directly to this server.

Depending on the complexity of the sensors' output formats, different techniques must be used to transform the sensor data into presence information. In this thesis we discuss a standards-based framework that enables aggregation of low-level data from multiple sources at a *personal presence server* and provides user-controlled data aggregation at a very high abstraction level. User-provisioned aggregation specifications in this system are evaluated similar to server-based scripts in modern web servers. Users who are accustomed to the development of scripts for Web page creation therefore will face a flat learning curve and do not need to have deep knowledge about machine-based learning and uncertain reasoning.

Figure 1.2: Example sensors

Before diving into the discussion of our approach the remainder of this chapter gives a brief motivation of presence aggregation and explains the architecture that is used when aggregation happens at multiple levels of abstraction. We define *presence zones* to be areas where a presence server knows the complete status of this subsystem and thus can automatically calculate the presence status in effect for this particular zone. We further show that users typically own more than one presence zone whereas a special presence server is necessary to merge the (possibly inconsistent) status descriptions of the various zones. The chapter ends with an outline of our contribution to relevant research areas followed by an overview of this thesis' formal structure.

1.1 Automatic Presence Status Determination

Most communication applications today have rather limited support for automatic detection of the users' current activities. Therefore, the presence status usually must be set manually. At least some applications may be configured to show a status description `idle` when the system has not received any user input for some time and switch back to `available` after a mouse movement or keyboard input was observed. In standard software, only little effort has been made to allow for more complex heuristics for presence status determination as this would require more sensors than most off-the-shelf computers for office use are equipped with.

Being a vivid research area for some years now [FLC04b, HKK+02, SAT+99, WCK04], *sensor fusion* provides a number of valuable results for use in presence systems. We suppose that sensor fusion in the near future will be a key element of any presence service to facilitate

services such as the initiation of ad-hoc conferences in a group of people standing at a particular location, as has been demonstrated in the research project GEOCOOP [OKB+05]. Figure 1.3 shows a possible architecture for a messaging system that relies on sensor fusion for status detection. Various sensors record environmental data that may or may not be relevant for the user's current presence status. Based on this data, usage patterns, and possibly other resources a central processor then calculates the current presence status to be published by the messaging system.

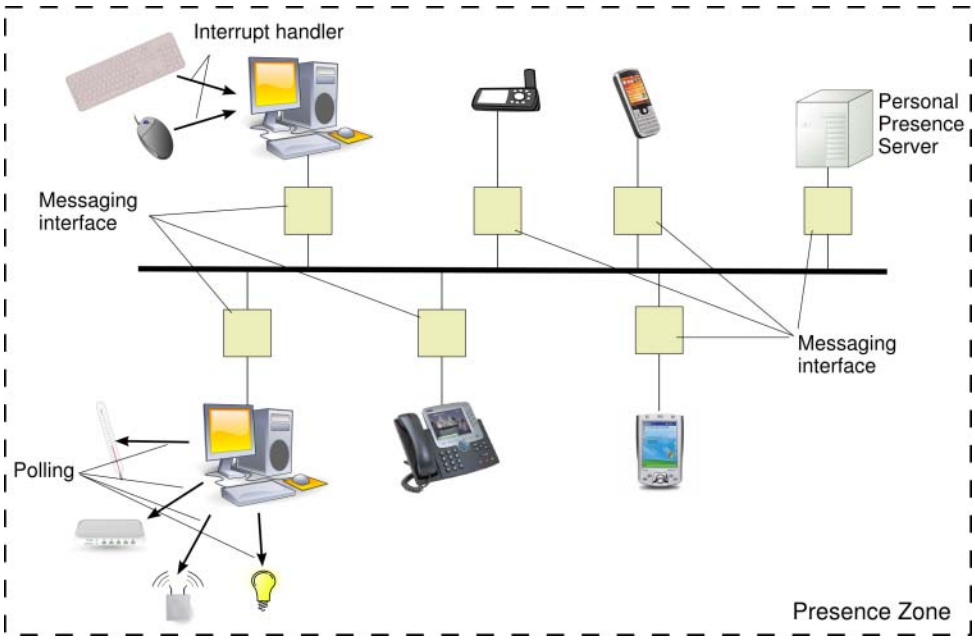

Figure 1.3: Example architecture for automatic presence status detection

In this architecture, low-level sensors such as motion detectors and personal communication devices are used to infer a user's presence at a specific location and to infer whether or not the user is currently busy. To do so, the raw data provided by the numerous sensors is aggregated resulting in a higher abstraction level, usually phrased in a dedicated presence description language (e.g. RPID, [RFC4480]). This high-level information on the user's availability then can be published to anyone who has indicated interest and who is authorized to obtain this information. To facilitate efficient and secure distribution, a generic event notification protocol can be used such as the *Session Initiation Protocol* (SIP) [RFC3261] extension defined in [RFC3265]. Standardized syntax and semantics for presence information exchange are specified in [RFC3859].

The set of sensors together with an aggregation engine form a local *presence zone* which provides a consistent view on the user's presence status. The high data rate of the sensors typically requires the devices constituting this presence zone to be in near vicinity of each other. Exceptions to this rule of thumb may exist depending on the available bandwidth and update rate (for example, SIP registrations of remote devices could be taken into consideration as the update frequency usually ranges from a few minutes to an hour or more). After the sensor data has been aggregated to a consistent view on this presence zone, the information can be distributed to authorized users by a presence server implementing one or more protocols for wide-area event

notifications. Depending on the internal format used to represent the presence status description, the data must be converted into a content format suitable for the respective notification protocol. For protocols complying to [RFC3859], the XML-based *Presence Information Data Format* (PIDF) [RFC3863] is used.

1.2 The Need for Multiple Abstraction Levels

As long as a presence zone constitutes a closed world, common algorithms for data aggregation and status inference yield satisfactory results, as several research projects in this area show. [TBH+04] A widely adopted approach to solve the problem of predicting the availability of a user for communication is to map the sensor data (or the results of further aggregation steps, respectively) to a single element out of a finite set of distinct *contexts*. If, e.g., a user's context has switched from "`in the office`" to "`gone home`", further conversation attempts from co-workers in an office context could be suppressed automatically.

Unfortunately, under several conditions, the heuristics used in this approach do not deliver sufficiently accurate information. In most cases, additional information from the user will be necessary to eliminate inconsistencies. Those conditions might for example emerge from the following situations:

- *Contexts overlap*: When meetings are held at lunch time or freelancers work at home, a clear separation of contexts is difficult. Several solutions come in mind to overcome this problem. First, contexts may have an associated accuracy value indicating the probability that this particular context is correct. Second, a user may be presented with a list of all contexts which might be in effect. Both methods are far from perfect as interpretation of the results is up to the observing user. This does not only result in unnecessary complexity but also introduces privacy risks as it may disclose more information than desired.

- *Sensors moving from one zone to another*: Many users carry mobile devices that can provide information about the user's activities. The most obvious examples are mobile phones and PDAs that are equipped with a wireless network interface. In addition, any device that has at least one input device such as a keyboard or a microphone can be used to generate input for the aggregation process.

  For context detection, the location of these sensors is significant: When a notebook computer records speech data in the user's living room this might indicate that the user is in home context. Otherwise, if neither at home nor in the office the user may or may not be on business travel. To distinguish between these different contexts, it must be known to which presence zone the specific device belongs.

- *Sharing of sensors*: Some devices such as motion detectors in hallways or conference phones are not associated with a single user. The sensor data produced by these devices therefore cannot be used for status inference directly. Instead, other data sources must be taken into consideration to determine whether or not the sensor data has any influence on the user's current status at all. In general, this kind of data source is useful only in combination with sensors that give a strong indication of the user's location, i.e. a positioning system or a scanner of a time and attendance system.

Although the results of applying existing heuristics for presence status inference might produce acceptable results even from inconsistent input, this list illustrates that the availability of mobile devices and shared equipment with logging facilities induce multiple presence zones that must be looked at in parallel. Figure 1.4 gives an example for a system with three distinct presence zones to be taken into account for a specific user.

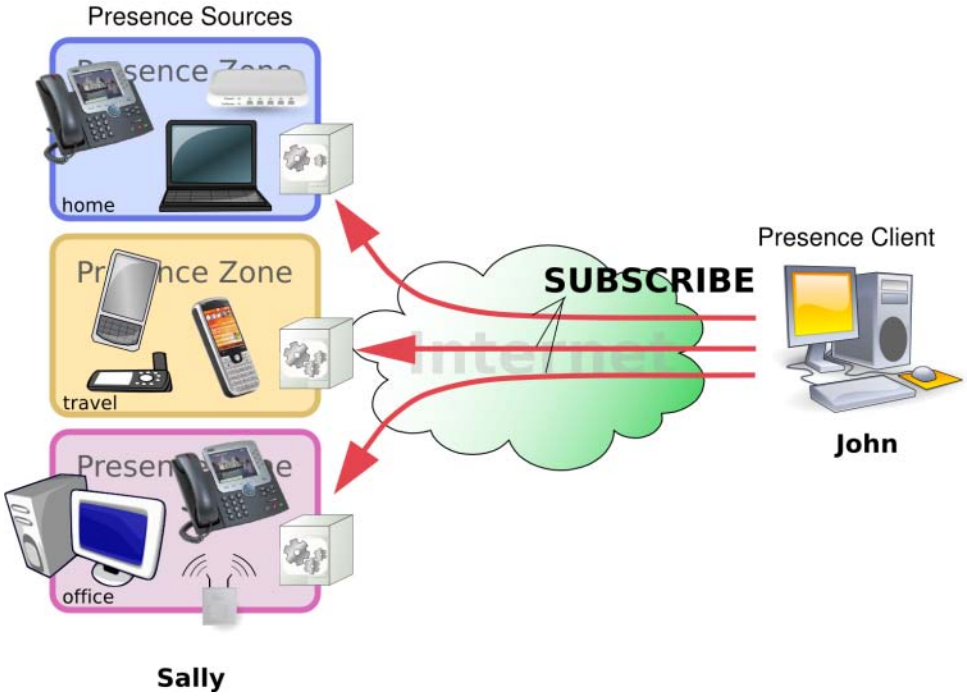

Figure 1.4: Multiple presence zones

In this example, the user Sally has defined three personal presence zones according to her habits. The presence zones reflect her to reside at home, be at work in her office, or traveling around at some unspecified geographic location. Every sensor device that is associated with a particular presence zone then contributes to the presence aggregation process. Mobile devices such as Sally's cellular phone in addition can change the presence zone they are associated with. In our example, bluetooth sensors might be used to determine if Sally's personal phone is in the office, at home, or somewhere else. The latter is especially useful to define the presence zone *travel*, containing mobile devices that are not covered by any other presence zone.

John, being interested in Sally's presence status, would have to subscribe all of these three presence zones to get the complete information. Now, as the status information published for one zone is not related to any other zone, John would be provided with three different views on Sally's actual status. For example, if sensors in the office as well as sensors at home detect a person moving around, both could take this as indication of Sally being present at that particular place. John, who has subscribed both entities therefore would be provided with inconsistent information.

Presence aggregation therefore must happen at different abstraction levels and at different points within an event notification system. After end systems have inferred a user's presence status for a single presence zone (typically using the sensor fusion approach) this information is

published to a dedicated server that aggregates the presence descriptions of multiple zones and thus creates a consistent view. In the previous example, the location of the cellular phone and a specific rule set could be used to resolve the inconsistency. To do so, the presence information published by every isolated presence zone is aggregated at another presence server that calculates Sally's official presence status, manages subscriptions to this user's status and generates status update notifications to interested clients. Figure 1.5 shows the resulting architecture.

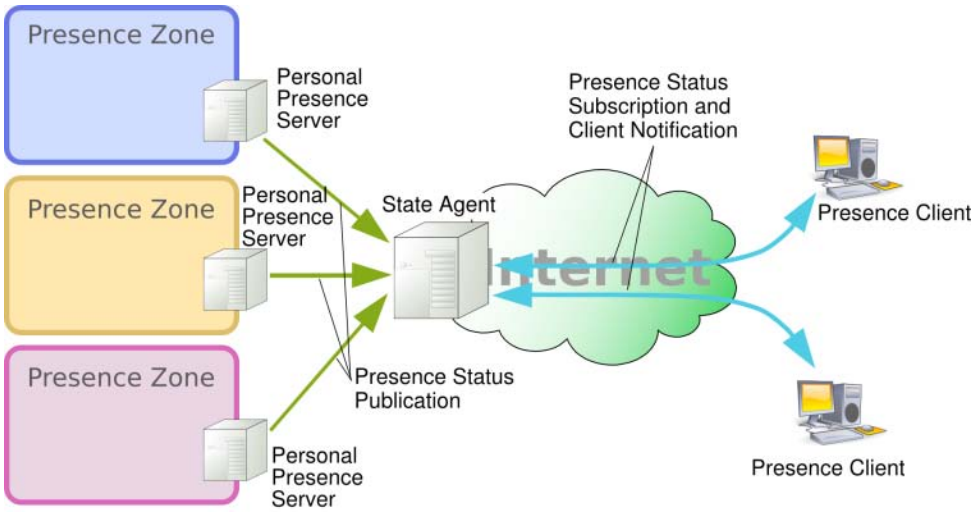

Figure 1.5: Status aggregation for multiple presence zones

A key element of this architecture is a presence server that can aggregate presence status descriptions generated by *personal presence servers* (PPS) each of which is responsible for a single presence zone. After merging the presence information from distinct presence zones, the aggregation server can distribute the results of this process to clients having subscribed to the user's presence status. To support the notion of different *watcher* identities and permissions to access specific details of the status description, the presence server may generate different views on the user status depending on the watchers' identities and may control the sending frequency of update notifications.

Some presence services such as the *presence event package* for SIP defined in [RFC3856] already cover status aggregation and thus offer a useful basis for further work. The processing envisioned by RFC 3856 merely comprises the combination of incoming status notifications from multiple presence sources in a single document on a syntactic basis. Being designed with efficient data distribution in mind, the protocol gives no explicit advice on further processing of information contained in presence documents on a semantic level. In particular, this aggregation mechanism has no immediate support for watcher-specific views on a user's status record. Complex operations such as applying user-specific scripts to incoming status notifications still require implementation-specific enhancements to this presence service.

1.3 Aging of Presence Information

Another important aspect of presence awareness is the dependability of the published information. As the quality of presence information primarily depends on timeliness and accuracy,

these factors will have great impact on the user acceptance of the entire system. Especially if the published presence status has been calculated based on uncertain information as motivated in Section 1.1, watchers need to know how reliable the status prediction is. Given enough bandwidth and low transport delay, this can be achieved by a high notification rate. Users then have the feeling of being instantly notified as soon as the observed person's status changes.

At Internet-scale, event notifications must be throttled down to a very low rate (say, once in a minute depending on the actual transport mechanism and payload size being used). Hence, not only the granularity of status information changes as mentioned in Section 1.2 but also the perception of having the latest news about the status of a particular user degrades.

The use of *state agents* and multi-step aggregation of presence information now offers a feasible approach to mitigate this problem: To show the decay of its exactness, the presence data is annotated with a probability value and a mathematical function that indicates the decrease of that probability over time. An additional threshold value can be given with each function to specify the lowest meaningful probability value. If the probability goes below that threshold, the aggregated presence status should be re-calculated.

Figure 1.6 gives an example of two decay functions with associated threshold values for the corresponding probability values. The possible presence states considered in this example are "`busy`", "`available`", and "`idle`". On the y-axis, the exactness values for a linear and an exponential decay function are shown over time. Two threshold values are indicated by dotted lines, one associated with the exponential curve, the other with the descending line.

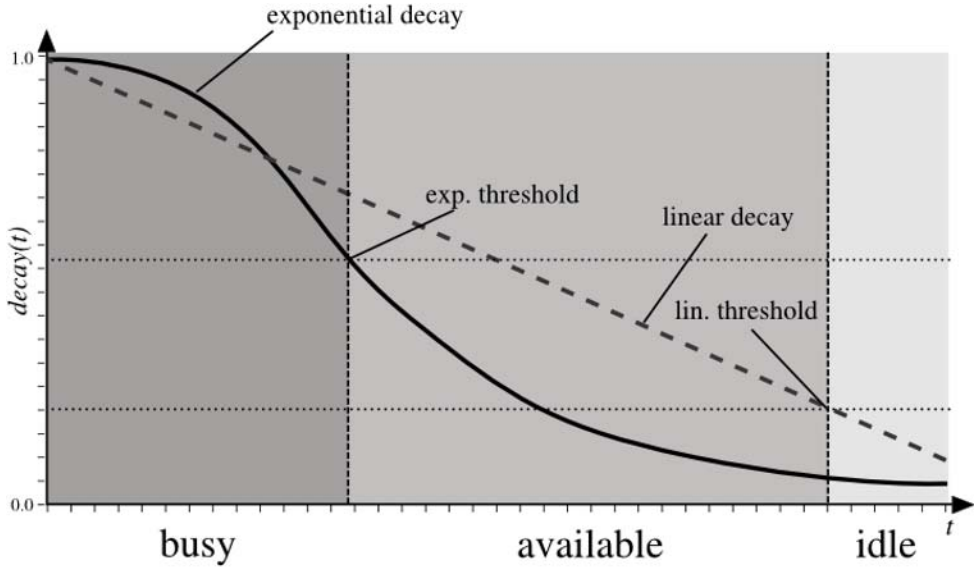

Figure 1.6: Applying decay functions to presence attributes

Suppose the aggregation engine has determined the presence status `busy` right after reception of the presence information. After some time—at the first vertical line—the exponential curve goes beyond its associated threshold value, and the presence status is re-calculated. As the corresponding status information (e.g. a presence attribute indicating that the user is currently in a phone call) is not considered up to date any more, the operation's result changes from `busy` to `available`, following the idea that the user has just put the phone on hook and is still in the

office. As more time passes, this assumption gets less precise and might not be accurate at all. Therefore, another function is being used to trigger a re-calculation event at a later point in time (indicated by the second vertical line). Now, the processor cannot determine any activity, so the status is set to default (`idle` in this case).

This example clearly shows that appropriate threshold values as well as the mapping from input values to a valid presence status cannot be inferred automatically or determined from log analysis. Instead, the user has to specify a number of parameters that reflect personal preferences and device capabilities. Our approach not only allows the specification of appropriate threshold values to trigger re-calculation of a presence status, it also lets users specify their own functions to determine the current presence status depending on the watcher's identity. A standards-compliant extension to the presence document format provides for backwards-compatible processing on presence servers that do not support this aggregation mechanism. The next section gives a more detailed overview of our contribution to research in this area.

1.4 Our Contribution

In this thesis, we discuss an approach to integrate multiple presence sources on a syntactic and semantic level, where the data aggregation is entirely under the control of the user whose data is being processed. The resulting framework is based upon open standards for representation and distribution of presence documents to facilitate support of existing systems. To prove the applicability of this approach, we have implemented a SIP presence server with a co-located aggregation engine that can be used stand-alone, or can be combined with existing SIP presence servers.

The main focus of our work has been on providing a standards-based solution to a specific requirement. Special care was taken not to force users into a service they do not need or want to use. The aggregation service therefore was designed as opt-in service, i.e. users have the option to stay with the system they are accustomed to. In particular, we have investigated the following specific topics in our research work:

- *Requirements analysis and architecture development*: Starting from a literature analysis on existing collaboration systems, we have identified the drawbacks that impeded global deployment of presence-based conference systems. Having elaborated the factors known to affect user acceptance, we have derived a list of requirements to be addressed in our work. To leverage use of presence aggregation at Internet-scale, we have developed an architecture that follows existing standards and adheres to latest insights in system design.

- *Presence information data format enhancements*: The existing content format for presence documents developed by the *Internet Engineering Task Force* (IETF) does not contain hints on the quality of the data conveyed. Specifically, there is no gradation of the exactness of listed presence values hence making automatic re-calculation of presence status records impossible if no guidance from the data source is available. Our extension to the widely used data format for presence information documents provides additional information on the probability of a given presence value to be correct. Moreover, its decay over time is modeled together with a lower bound indicating when the data has become useless.

- *Rule-based control of presence information processing*: User control of the aggregation process is offered by means of rule sets to be uploaded to a presence server that supports the extensions described here. A Turing-complete scripting language has been enhanced with a specific library of commands for manipulating presence status records. Triggers in the execution environment allow for asynchronous notification of external events that might affect the current presence status and cause sending of related notification messages to subscribed users.

- *Implementation*: To show the effect of user-controlled presence aggregation at multiple levels, we have implemented a presence server that supports the evaluation of user-provisioned aggregation specifications as well as the aging mechanism described previously. Due to its modular architecture, the server can be used for processing of low-level sensor data as well as abstract presence information used in SIP messages. The server application and the protocol-specific messaging components use a flexible and lightweight communication infrastructure, the standardized *Message bus* defined in [RFC3259].

The results of our work presented in this document provide substantial contributions to the following areas of research:

- *Computer Supported Collaborative Work (CSCW)*: Awareness of users always has been an important feature of nearly every CSCW system. Up to now, user acceptance of those systems is still very low. Besides their prevalent fear of being continuously observed, users complain about missing control and transparency of the system. Addressing these problems, our approach aims at increased user participation and improvement of interpersonal communication in real-time as the number of failed call-attempts will go down.

- *Internet-scale Event Notification Systems*: Discrete event notifications face the problem of inconsistent status information. The lower the update frequency is, the more imprecise and outdated the presence data will be. The aging mechanism proposed here allows for better prediction of presence states, both for automated aggregation and for users who are watching this information.

- *Implementation of Distributed Systems*: We have developed an architecture for decomposed presence servers using a local message bus. A set of abstract commands for controlling the aggregation processor as well as concrete protocol engines makes the core components agnostic of actual presence transport protocols and thus provides a basis for robust presence gateways interconnecting different protocol domains.

- *Multimedia Conferencing*: Real-time communication over the Internet yet suffers from high call-completion times and large call-drop rates. In this thesis, we show how presence services can be used to improve this situation and what technical requirements are to be met to achieve these goals. Our demonstrated solution to the specific problem of user-controlled aggregation of presence documents not only promotes user-acceptance but also can take significant loads off the network as it allows for a moderate frequency of status update notifications.

- *Security and Role-based Authorization in Presence Systems*: Another important aspect of missing user-acceptance is the fear of systems that enable creation of user-specific

profiles. We have documented authorization frameworks that give users control over the process of publishing their personal data and propose a set of default rules to retain privacy of sensitive data when merging presence information from multiple sources.

1.5 Structure of this Thesis

The remainder of this thesis is structured as follows:

- Chapter 2 gives a brief overview of past and present applications that use presence information notifications to maintain a local view on the state of the global information system. These examples will show different notions of the term *presence* in their specific application contexts. The chapter ends with a definition of that term to be used in the remainder of this document.
- Chapter 3 describes the requirements posed on a presence service that facilitates interactive multimedia communication over the Internet. The focus is on the aggregation of presence information from multiple sources where the user perspective is taken into account as well as the technical constraints imposed by the wide-area dissemination of status descriptions.
- After having stated the problem to solve, Chapter 4 presents related work that was used as a starting point for our development. Moreover, existing protocols and presence information document formats are examined to get a better understanding of their specific characteristics, including a brief overview of the IETF model for instant messaging and presence.
- Chapter 5 then describes our system architecture together with some comments on its implementation. Our major contribution, a language and semantics for aggregation of status notifications from multiple presence sources, is discussed in detail in Chapter 6.
- The implementation created as proof of concept for our approach is discussed in Chapter 7, together with comments on the tests we have performed with our platform. We also present some ideas for further evaluation of aggregation results.
- In the final chapter, the achievements made from our work are summarized and set into correlation with other research projects. Moreover, we show the open issues and present some ideas for future enhancements of this work.

In addition to the literature listed at the end of this thesis, there are several references to resources in the World Wide Web throughout this document. These links have been included primarily as examples of existing sites and applications, and their contents may change at some point of time. These dynamic reference therefore can only be treated as a snapshot that was valid at the time of writing this document and possibly some time thereafter. We have checked the integrity of the references contained in this thesis on November 2, 2006.

Chapter 2

The Role of Presence Systems for Interpersonal Communication

To state the requirements for a presence service that leverages instantaneous communications between users, it is necessary to understand the implications of this service, both, from the users' perspective and from a technical point of view. To do so, this chapter gives a brief overview of the evolution of presence-aware applications from simple interactive query-tools to more elaborate enterprise communication systems. Based on these real-world examples, we define the term *presence* as the awareness of other users' availability and willingness to take part in a conversation using specific media. From the historical background of presence applications and technical restrictions that exist for Internet-scale event notification systems, we derive a typical use case for user-controlled presence aggregation. This use case also constitutes the basic application scenario addressed in our research work, targeting at a generic solution to facilitate setup of interpersonal real-time communication.

2.1 Historical and Technical Background

In the late 1980s, when the majority of corporate entities started to replace or augment central mainframe computers by cost efficient and powerful desktop computers, a paradigm shift took place, from centrally controlled, homogeneous architectures to distributed and more heterogeneous systems. The most visible indication of this process was the development of early hypertext systems which where predecessors of the *World Wide Web* (WWW) being introduced in 1989 by Tim Berners-Lee [Ber89, BCL+94].

The major motivation of hypertext systems was to leverage cooperative work based on interconnection of static objects (ranging from documents stored in a central database up to document fragments that are part of a global information system like the WWW, cf. [Bus45, Nel81]). When networked computers in the 1990s became part of the standard equipment of nearly every office (at least in the financial and technology-oriented sectors), employees started to access their company's data while at home or on business travel. At that time, the demand for conferencing systems and systems that facilitate *computer supported cooperative work* (CSCW) grew and was an important incentive to push forward standardization of conferencing technologies in the mid-1990s—the focus of collaborative work has been extended from in-house solutions to standardized, globally connected interpersonal communication systems that facilitate ad-hoc text messaging as well as synchronous audio-visual conferences.

One aspect of communication is the awareness of other people's presence as well as their current or future activities. Several publications on this topic refer to UNIX-tools like **write** and **talk** as roots of instant text-based communication mechanisms (see e.g. [GrPa02, RiKh98b]). Moreover, the commands **finger** known from the TENEX operating system [BBM+72] and **rwho** support a basic mechanism that allows for users to determine whether or not other users are logged in, and to determine for how long they have been idle. In addition, the **finger** protocol [RFC1288] has optional fields to provide information on the human user, e.g. location of his terminal, office, his phone number etc. A remote user may also provide a free-form text message that is transferred with the response to a finger query. This data is opaque to the finger client and may be used to give additional hints about the user's future plans.

A project that made systematic use of the finger protocol to coordinate actions between members of a software company is described by Harter and Hopper in [HaHo94]. The *Active Badge* system at the Olivetti Research Laboratory (ORL) used infrared sensors to determine the location of people within the office building. A simple interface allowed for human interactions with the sensor devices in order to read messages or set the current state. Within the enterprise network, the presence information not only enabled paging of co-workers but was also used to setup ad-hoc multimedia conferences.

The fact that the initial version of the finger protocol, RFC 742, was written in 1977 for use with ARPANet protocols pre-dating the Internet Protocol (IP) already shows that the desire to have immediate knowledge about other people's status was a key factor driving the development of interpersonal messaging systems from the very beginning of the Internet. Recent empirical studies [NWB00, MiSm00, GrPa02] observed that both user awareness and instant messaging in combination are used primarily for managing interpersonal communication rather than for information exchange itself. In particular, brief text messages are often used to determine the need for other forms of communication, e.g. to perform phone calls, personal meetings, or even offline document exchange via email. In [NWB00], Nardi et al. observed that this negotiation pattern they refer to as *outeraction* is prevalently used in business life, whereas Grinter and Palen [GrPa02] found that teenagers use instant messaging not only for communication management but also for exchange of content information.

In addition, Herbsleb and Grinter in [HeGr99] show that use of distributed collaboration platforms has increased in enterprises with world-wide branches, especially when IT-personnel is involved. Most of the involved parties stated that they feel better and thus more motivated when they see that they are not working alone. The exchange of short greetings via instant message—if exchanged regularly over a certain period of time—is reported to be sufficient for people to state that they *know* each other. [DoBl92]

In [HeGr99], Herbsleb and Grinter address the problems that arise from geographic distribution of development teams. Their observation was that unplanned meetings and ad-hoc communication have a significant impact on the productivity of co-located development teams. In case of geographically distributed teams, the authors suggest to invest in tools that foster awareness of team members' availability and simplify establishment of cross-site conferences.

2.1.1 Evolution of Network Technologies

Around 1999, when Herbsleb and Grinter published their findings, groupware platforms and workflow applications already addressed the key issues identified in that article. A major characteristic of those systems was their client/server architecture with a centralized compo-

nent being responsible for workflow management, conference control, data storage, and user tracking. Enterprise messaging systems like *IBM Lotus Instant Messaging (formerly called "Sametime")* or *Microsoft Exchange Server* were designed to fit large organizations' communication processes, and integrate additional functions that support management of organizational knowledge. Although most enterprise messaging systems use proprietary protocols to exchange presence information, isomorphic mappings to several standardized protocols could be defined easily, although extended functions such as logging and auditing may not be available.[1]

Standardization efforts for presence protocols within the *Internet Engineering Task Force* (IETF) began in 1998, when vendors of real-time text messaging applications with awareness mechanisms tried to outplay each other with their proprietary messaging clients after *America Online Inc.* (AOL) had bought the Israeli startup *Mirabilis Ltd.*, the developer of the most popular instant messaging application *ICQ*[2]. At that time, there were already several instant messaging systems with support for presence status distribution, those from AOL, Yahoo and Microsoft having the largest user bases (cf. [Sal04]).

An important aspect regarding the evolution of presence services is the convergence of network technologies to provide ubiquitous support of IP-based services. For example, from release 5, the *Universal Mobile Telecommunications System* (UMTS) provides interfaces to certain services of the public Internet, presence and messaging services among them. With mobile devices getting smarter, users pay more attention to IP-based services. The rising demand for a tight integration of their personal devices with the global communication infrastructure already led to instant messaging and presence clients supporting protocols such as AOL's *OSCAR* or the *Messenger protocol* of Microsoft Corporation.

Although there is large agreement between vendors of equipment for personal communication, software developers, and protocol designers to use the Internet protocol for messaging and presence services, the variety of existing solutions has brought protocol-specific islands into existence. As a result, users today have several different presence clients running (e.g. AOL Instant Messenger, MSN Messenger etc.) or use multi-protocol clients such as GAIM[3]. In addition, public gateways can be used that map between the most popular instant messaging and presence protocols. This kind of multi-protocol gateways includes unified messaging systems and most enterprise messaging systems. Some of these systems even provide a module supporting the popular *short message service* (SMS) offered by GSM (*Global System for Mobile Communication*) networks which has become very popular with teenagers in Europe as soon as cheap mobile handsets were available.

Multi-protocol clients as well as gateway services are powerful tools to interconnect disjunctive information spaces created by incompatible communication protocols. As a result, the definition of user presence for any of these information spaces must be extended to fit the definition of user presence in any other information space as well. In other words, a superset of the presence information set must be used to express a user's presence status within the interconnected information space. As most of the presence protocols used in the Internet today have a

[1] Automatic logging of communication sessions and later retrieval of transcripts are important features of IM systems when used in a company where the entire production process has to be documented according to Section 404 of the Sarbanes-Oxley Act (Pub. L. 107—204, 116 Stat. 745). <http://thomas.loc.gov/cgi-bin/query/z?c107:H.R.3763.ENR:>

[2] The announcement of the 1996 ICQ release can be found at <http://company.icq.com/info/press/press_release2.html>.

[3] <http://gaim.sourceforge.net/>

similar understanding of user presence semantics, a general presence model can be defined to be used as common model for all of these protocols.

In 2004, the IETF working group *SIP for Instant Messaging and Presence Leveraging Extensions* (SIMPLE) started development of a data model for presence information, together with operations defined on this data and various vocabularies for expressing information about the personal presence status. Subscriptions in this model always address the status of real persons, who may provide additional information on specific services they offer (e.g voice communication or text messaging), and characteristics of devices owned by these persons. Being an important basis for standards-compliant presence applications, the IETF presence data model defined in [RFC4479] still lacks some important features such as support for user-controlled aggregation or external triggers that cause re-calculation of status information.

2.1.2 Conferencing Applications

As mentioned before, distribution of presence information in the early days of networked computing has been used primarily for conferencing applications, as a means for scheduling ad-hoc multi-party conferences, or for distant awareness of co-workers in geographically distributed locations. A prominent example for presence-aware enterprise-wide conferencing is the *Portholes* system described in [Bux95a, DoBl92]. Harter and Hopper in addition show how presence-information can be used to locate people and equipment in mobile office environments when equipped with infrared transmitters (*Olivetti Research Limited infrared network*), so-called *Active Badges*. [HaHo94] Modern applications also use cost-efficient *Radio Frequency Identifier* (RFID) tags to detect if specific objects or people are in near proximity of the scanning device. While the Portholes system initially aimed only on creating a tele-presence environment to enhance instant communication between distant co-workers, the Olivetti project already was designed to support additional functions like location-sensitive communications (e.g. disabling communication channels that currently are not available), or teleport graphical X sessions to nearest display using one of two push buttons etc.

In addition, telecommunication companies, especially operators of cellular networks, explore presence-related techniques to support instant call setup, often called *push to talk*. For two-party calls, presence information can be used to determine whether or not the callee is available and willing to answer a call, thus reducing the number of non-billable traffic due to incomplete phone calls (i.e. increase probability of call completion). A series of industry standards for *push to talk over cellular networks* (PoC) has recently been defined by the Open Mobile Alliance.[4]

Both conferencing systems and applications in cellular networks have been limited to closed network environments so far. While the restrictions imposed by operators of mobile networks are merely a matter of their business strategy, conferencing has been limited to enterprise networks for a long time because the global Internet suffered from missing standards for secure group communication, distant conference management and floor control, as well as a mechanism to guarantee a specific quality of service. In [JLG91], the authors present four tasks a communication system must support:[5]

[4] See <http://www.openmobilealliance.org/tech/wg_committees/poc.html>.

[5] Jirotka et al. are referring to J.C. McCarthy, V.C. Miles, A.F. Monk, M.D. Harrison, A.J. Dix, and P.C. Wright: "Using a minimal system to drive the conceptual analysis of electronic conferencing". University of New York

- Synchronizing communication,
- maintaining conversational coherence,
- repairing conversational breakdown, and
- maintaining a shared focus.

In short, these functions can be summarized as the necessary tasks to setup and tear down communication channels, as well as controlling a running session both on a technical level and by applying suitable group policies for multiparty conferences. Existing conference systems require a tight coupling of session control and management of participants to maintain state information throughout a session.

Recent standardization efforts, especially from the IETF and industrial consortia such as the *Open Mobile Alliance* (OMA) and the *Third Generation Partnership Project* (3GPP) have leveraged network convergence using the Internet Protocol as a basis for distributed applications. With standards and regulations for all-IP networks on top of various different technologies at the link-level, developers are provided with a common framework to define new communication protocols and applications. As in IP-based telephony, session establishment, modification and termination is done using the *Session Initiation Protocol* (SIP, [RFC3261] and following). SIP-based protocols to establish and control multi-media conferences using a central server are being developed by the IETF *XCON* working group hence facilitating supplementary services like e.g. ad-hoc conferences for which the SIP Event Notification mechanism defined in [RFC3265] could be used.

For loosely coupled conferences in heterogeneous networks, presence technologies can be used to provide a common view of the participants' state throughout all end-systems in that particular conference. Currently, only experimental implementations on the basis of IP multicast data transfer exist (e.g. GRYPHON [BCM+99] and SCRIBE [RKC+01]). We expect this as a research topic of growing interest once robust session control protocols for conferences with a centralized focus as well as presence systems based on open protocols are deployed.

2.1.3 Event Notification Systems

Internet-scale *Event Notification Systems* (ENS) have been around since the early 1990s and have become a popular research topic when the limitations of the strict client/server architecture of the World Wide Web impeded specific types of new services, especially those based on push-messages. Before 1999, when the IETF started work on an architectural model for instant messaging and distribution of user-presence information (see Chapter 4), a number of conferences on this topic was held by the scientific community, documenting existing wide-scale event notification systems.

An example for an existing ENS is given in [Car98]: The SIENA system (short for *Scalable Internet Event Notification Architecture*) represents a typical distributed event notification system built upon Internet technologies. To be notified of specific events, a client must subscribe that particular event type by any potential event source. A distributed server hierarchy is used to forward event subscriptions as well as related notifications. In general, servers are co-located

Technical Report, 1990.

with monitoring components used to aggregate event notifications to be matched against complex event patterns. Thus, any monitor-component within the notification system may act as event source, depending on the event patterns being subscribed.

To simplify instantiation of routing tables, SIENA supports an announcement-based operation mode. Announcements act as filters that automatically block forwarding of notifications for events that have not been announced. Subscriptions to events are allowed even if not yet being announced. However, servers may drop these subscriptions as no notification will be sent to that destination anyway. To overcome the race condition between announcements and subscriptions, SIENA components are soft-state and thus regular refreshes of subscriptions and announcements are required.

In local environments where network bandwidth and latency are minor issues, there have also been approaches to enrich messaging systems with intelligent functions to add structuring and filtering to notification routing. *InformationLens* (see [MGT+87, Mac88]), e.g., provided a rule-based language to specify structuring of incoming messages. Similarly, the *Andrew Message System* (AMS) described in [BoTh88] provided "active messages" equipped with executable code. To protect users from unwanted messages the AMS provides a script-language to specify user-specific filter rules to process incoming mails and automatically send out an answer if wanted.

A more detailed discussion of event notification systems including design issues and architectural decisions is given in Chapter 3.

2.2 Theoretical Foundations

As our work requires setup of an interoperable and extensible presence service in an IP-based wide-area network, the underlying architecture has to meet some general requirements such as scalability and openness. Moreover, privacy and integrity of sensitive data carried over the network must be assured as well. Currently, proprietary large-scale event notification systems co-exist with systems that are based on open standards addressing interoperability, extensibility and fault tolerance of the protocols being used. Recent standardization efforts especially at the Internet Engineering Task Force (IETF) have been made to define a generic framework for presence notification services that allows for exchange of presence data between different administrative domains, independent of the transport mechanism being used. In this section, we discuss the applicability of the IETF approach for our work based on theoretical foundations found in research results on generic event notification systems. In particular, we refer to the findings of Rosenblum and Wolf [RoWo97] who define a framework for wide-scale event notification systems that can be used as a reference for a new system design, and the analysis Carzaniga has conducted in the context of his work on the SIENA system (cf. [Car98]).

As a basis for discussing the design of our presence aggregation system, we refer to the following seven dimensions of Internet-scale event notification services that have been identified by Rosenblum and Wolf (cited from [RoWo97]):

1. an *object model*, which characterizes the components that generate events and the components that receive notifications about events;
2. an *event model*, which provides a precise characterization of the phenomenon of an event;

3. a *naming model*, which defines how components refer to other components and the events generated by other components, for the purpose of expressing interest in event notifications;
4. an *observation model*, which defines the mechanisms by which event occurrences are modeled and related;
5. a *time model*, which concerns the temporal and causal relationships between events and notifications;
6. a *notification model*, which defines the mechanisms that components use to express interest in and receive notifications; and
7. a *resource model*, which defines where in the Internet the observation and notification computations are located, and how resources for the computations are located and accounted.

Being primarily developed for distributed software systems rather than infrastructure services for interpersonal communication, these dimensions only give an abstract guideline for discussion and do not qualify as evaluation criteria for our design. Therefore, this catalogue has been used only as theoretical basis for our design while the evaluation of the entire system described in Chapter 7 refers to the requirements that are specified in Chapter 3 instead. Said this, the rationale for our design is described along this foundation in the following subsections.

Object Model

The *object model* describes the acting parties of an event notification system—typically initiators of state changes and receivers of state update notifications—, and objects that represent the system's status information. In terms of user presence systems, actors are state agents and watchers, while presence state is being stored at some place in the network. When a presence agent modifies a status record, a status update notification may be triggered in order to distribute this change to interested watchers.

In case of most presence systems, the status record is hosted by a dedicated server responsible for sending of update notifications. The server then acts as a special type of state agent on behalf of the party that initiated the status change. The formal approach presented here makes use of this common object model.

The identifiable objects are:

- *Principal*

  An agent outside the presence distribution infrastructure that makes use of the presence service to disseminate information on its current *presence status*, i.e. the availability and willingness to communicate. If not explicitly mentioned, the terms "user" and "principal" are treated as equivalent in this thesis.

  The identifiable entities in the network representing a principal are called *presence entities* or *presentities* for short.

- *Watcher*

  A software agent that expresses interest to in a principal's presence status. A watcher may poll the presence information in regular intervals or install a longer-lasting subscription at

a *state agent*. Watchers have a unique identity that can be used to authorize subscriptions to a presence status record of a particular principal.

As the presence service is typically used to enable interpersonal communication between human users, we identify watchers with the actual users who are interested in the presence status of their communication peers.

- *Presentity* (*presence entity*)

  An agent that represents a principal's presence status in the network. Presentities can be queried for their current presence status and may publish updates of their status to a state agent. Presentities have a unique identity within the network that can be used to address this entity. An explicit binding between presentities and the identity of the associated principal may exist outside the presence service.

- *(Presence) state agent*

  A network entity that handles subscriptions to the presence status of the principals it is responsible for, and sends update notifications to the subscribed watchers. The state agent is typically associated with the administrative domain associated with the principal's identity.

- *Presence status record*

  The internal representation of a principal's presence status at a state agent and the resource that is addressed in requests for presence status subscriptions. An abstract view on this data structure may be conveyed through the network as reaction on a watcher's subscription to the principal's presence status. Depending on the context, the term *status record* may refer to the data structure stored at a state agent or to the watcher-specific view that is transmitted over the network.

Event Model

Events occur at an object of interest independent of any watchers that might observe it. In accordance with the model of Rosenblum and Wolf [RoWo97], events have no duration, i.e. an event reflects the object's status at a specific point in time. For the purposes of the presence architecture defined here, events occur whenever a presence agent publishes a status update to its dedicated presence state agent. In other words, a change of a principal's presence status can be observed only after the corresponding record that stored at a co-located presence state agent has been updated in some way. The event hence reflects the update operation on the stored presence record, not a specific state transition of the presence agent.

Events are identified by the unique presence URI used to identify their corresponding principal's presence status (see the description of the naming model below) together with an event type and optionally a number of type-specific attributes. As presence identifiers explicitly contain information about the location of the responsible state agent, subscriptions are directed to the particular URI.

Since presence information typically contains sensitive data, visibility of update events depends on the watcher's identity. Events therefore can be observed only if the particular observer is authorized to access the information contained in the update notification. As the authorization

information—especially the application-specific definition of different authorization classes—is not part of the naming model (instead, it is part of the event content), a presence state agent must process the event before a notification can be generated. The processing rules are part of the notification system's information policy that is described with the observation model below. A detailed discussion on this topic is given in Chapter 7.

Naming Model

As stated above, event names are protocol-agnostic *Uniform Resource Identifiers* (URIs) according to [RFC3986]. In general, applications should use the URI scheme `pres` defined in [RFC3859].

The naming of events has no implication on the event semantics, thus an arbitrary string can be used for identification as long as it is globally unique. Given this, other URI schemes than `pres` could be used as identifier. In particular, events originating in another presence system using a transport protocol with its own naming scheme (e.g. SIP [RFC3265], XMPP [RFC3920]) could be used as well. In this case, the presence state agent may gateway between different presence protocols using different naming schemes. Modern protocols that adhere to the requirements specified in [RFC2779] give detailed rules to map between their specific naming model and the `pres` URI scheme, e.g. based on the *Dynamic Delegation Discovery System* (DDDS, [RFC3401]).

In general, the presence status of a principal is identified by a presence URI of the form `pres:user@domain`, where `user` denotes a unique name within the administrative domain `domain`. This common structure allows for simple mapping between several URI schemes without loss of information. A corresponding SIP-URI should be `sip:user@domain`, i.e. the administrative domain `domain` defines an isomorphic mapping between the URI schemes `pres` and `sip`.

This URI structure denotes the principal to which the presence status refers. To subscribe this status, a watcher additionally has to specify a token representing the type of event he wants information about. This combination of principal address (the `pres` URI) and one or more event types can be used by the presence status agent to limit the number of notifications sent to the subscribers of a particular presence state. For example, a simple application might request information about the availability of a principal for communication via instant messaging, while another application might also want to know whether the subscribed user is currently typing an instant message (this is a common mechanism in instant messaging applications to avoid both peers typing their messages concurrently).

Observation Model

An event is triggered whenever a presence status record is changed, either manually or automatically. Events therefore occur typically as a result of receiving status update messages, manipulation of presence status records using a protocol for remote modification of structured documents (e.g. using the *XML Configuration Access Protocol* defined in [XCAP]), or even by user input of presence attributes via WWW-interface.

As presence status typically depends on additional parameters such as daytime or user location, event-systems may automatically trigger system-specific events. For example, a presence system could infer user availability from recent keyboard activities performed by that particular

user. Because only the abstract information is published (i.e. “available” instead of “is typing”), the quality of this information degrades over time. Hence, the presence record in question is annotated with a decay function specifying the information’s quality. If a certain event-specific threshold is reached, the user’s overall presence status will change, e.g. from “available” to “away” since there was no user activity for the last ten minutes.

The example already shows that there is no general rule saying to which amount presence information degrades over time, or what the threshold for a status change is. Because of this, the selection of an appropriate decay function as well as threshold calculation always depends on the specific event type and—probably more important—on user-preferences. As a result, the event-system supports timer objects that are instantiated by an aggregation function. Expiration of a timer object is treated as if an event of type `timeout` has been observed, and the presence status is calculated again.

Moreover, timer objects can be used by aggregation functions to implement polling of events. At each loop, the function checks some system-specific parameters and if necessary, the current presence status is updated according to the actual parameter values.

The observation of an event triggers subsequent processing steps within the presence state agent. The event description is passed to an optional user-specific transformation function to generate watcher-specific status update notifications. The notification messages are sent to subscribed watchers immediately after the transformation process. The actual set of presence attributes being sent to a watcher is further restricted by additional authorization information contained in the presence information document.

Time Model

Whenever the presence status of a principal changes (either because of the principal’s current activity, location, etc., or because of a change of the technical devices the principal uses for communication), the presence system may be triggered to publish the change to watchers that have expressed their interest in this information. As the dissemination of the status change takes a certain amount of time, some watchers may already have received the notification message while others still rely on the old status information. Applications therefore must be aware of the race condition caused by this delay and not rely on the availability of a certain resource because of the presence status that is known by the watcher.

Another issue may occur if the status changes more frequently than the dissemination of update notifications takes. If this was the case, status updates might be received by a watcher in the wrong order causing a late status update to override the more recent status description. To avoid this situation, the status updates should have a sequence number and should be self-contained, i.e. they can be interpreted without knowing the previous status change. This way, notifications might even get lost in the network without too much impact on the system’s overall consistency.

Notification Model

Notification messages must express the status change that raised the particular event being notified. Various formats exist to describe these status changes, ranging from sets of key/value

pairs carrying raw sensor data as in the Solar system (see Section 4.1.1) to structured descriptions of communication services (see Section 4.2.2.2) embedded in MIME content envelopes according to [RFC2045].

The preferred notification model in this thesis uses abstract descriptions of available media for interpersonal communication based on a fixed vocabulary for presence states. Depending on the time model and the actual format of status descriptions, notifications may contain partial updates to existing status records. If reliable transfer and correct sequencing of notification messages cannot be ensured, the status description should always be self-contained, i.e. a new notification replaces the existing status description that is stored at a watcher.

In addition, status descriptions should be independent of the event notification service used to signal status changes to subscribed watchers to facilitate creation of gateways that interconnect different presence protocols.

Resource Model

The resources of the presence service are the network entities we have previously described with the object model. Basically, the event notification service constitutes an overlay network that provides a forwarding mechanism for presence-related messages. Its main objective is the routing of subscription requests to sources of presence information and the conveyance of status descriptions to watchers that have expressed their interest by subscribing to the particular event.

In addition to presentities and watchers the resource model comprises state agents that offer subscription management and the publication of presence information provided by a presentity. Being an optional resource in most event notification services, state agents are essential components of a presence aggregation service. The reason for this is that state agents receive the status updates of all presentities associated with a specific principal to create a consistent view on the principal's current status.

2.3 Meaning of "Presence"

Comparing examples for presence systems, one will notice a rather broad notion of the term "*presence*" in distinct application environments. Besides its original meaning of "being physically there" the term has recently experienced a slight shift towards a more abstract notion of information about a resource's current status. In conferencing tools, e.g., "presence" describes the *availability* of a person to take part in an interactive conversation. Other examples include services that specify the status of dynamic processes such as flight schedules or time and attendance systems.

Given this, the meaning of the term *presence* obviously depends on the application context where it is used. Despite of this fuzziness, most authors have a common understanding of the core semantics of that term, referring to the aspect of presence to give some information on the awareness of a remote peer. In [IjRi03], Ijsselsteijn and Riva give a more elaborate introduction on presence as "experience of *being there* in a mediated environment". The focus of their work is on human perception as a result of "multisensory stimulation from both the physical environment as well as the mediated environment". By treating the physical environment as being equivalent to the artificial, mediated environment, the authors found three types of presence:

- *Physical Presence*

  A media space where people are physically at the same location. Simple media spaces only create a basic feeling of what is perceived as some type of presence but do not allow for interaction between human actors with the media space, respectively. Examples for this type of media space are paintings, theater, and cinemas.

  Interactive media spaces provide means to manipulate the presence information that is carried by that particular media space at a time. For example, one might think of a virtual reality environment that shows a museum's exhibitions. Actors in this media space can take a virtual walk through the passage, approach showcases, and read (or hear) information about displayed items. If he is not interested in a particular item, the person can just walk by and the media space does not provide him with additional information.

- *Social Presence*

  In contrast to physical presence, social presence refers to shared media spaces where people can interact with each other, even over long distance and different time zones. There is no need for actors to be located in a specific physical place (like e.g. the cinema), as the media space is completely virtual. The media space provides means for synchronous and asynchronous communication, creating a virtual context for social interaction.

- *Co-Presence*

  Co-Presence here represents the concept of *telepresence*[6] known from CSCW applications. Here, the media space combines characteristics of physical presence and abstract awareness services available in social presence environments. Co-presence often is found in enterprises with a large number of branches spread over several geographic regions. Collaboration tools support asynchronous communication and work-flow processing, as well as user awareness.

Within this thesis, the term *presence* or *user presence* always refers to an explicit indication of a principal's availability and willingness to participate in computer-based communications that can be observed by (a possibly empty set of) authorized *watchers*. A *principal* is an agent with a unique identity, typically a human user with a specific role or a software agent acting on behalf of a user or an organization. For example, for a conference bridge that is able to transform a phone call into a multiparty conference upon request, the bridge's presence could be indicated using an enhanced vocabulary describing its conferencing capabilities such as the number of voice lines, supported video codecs etc. In the user interface of a watcher's presence client, the conference server might show up as a communication device with specific parameters. A *watcher* is an agent with a unique identity that has subscribed to a principal's presence status. Watchers can actively poll status information or passively wait for incoming notifications.

This notion of *presence* has been used in several existing CSCW-applications, for example in *AETHER*, an awareness engine for a specific kind of media spaces (see [SBB97]), or in the *GEOCOOP* project (see below). A presence agent within these systems represents a human agent or an agent's application in a specific medium. Every agent in a media space has a

[6] According to [Mar02], this term was coined by Marvin Minsky in his 1980 description of a networked environment for distant manipulation of real-world objects (Minsky, M: Telepresence. Omni, pp. 45—51, 1980).

limited aura and nimbus that restrict its facilities of interaction with its environment. The system provides several levels of awareness, depending on an agent's communication capabilities, characteristics of its immediate environment, and other agents' capabilities.

Systems like *AETHER* and *GEOCOOP* transfer certain limitations of the real world into the virtual environment, preserving borders imposed by geographic location, distance, and visibility. Used primarily to simplify the application's user interface, there is no compelling reason for this artificial limitation of the interaction facilities the virtual layer provides. Instead, presence systems today are known to overcome exactly the problem of geographic segregation of individuals, leading to *social presence* as mentioned above.

2.4 Presence-Aware Applications: Two Examples

As motivated before, presence information can be used for many different applications. Even if there is a common notion of the term *presence*, the semantics of presence data still depend on the context where it is being used. For example, early co-presence applications like Xerox RAVE (*Ravenscroft Audio Video Environment*, see [GMM+92]) were based on bi-directional audio/video links between two distant research sites (the Xerox PARC in Portland, and its European counterpart, Xerox EuroPARC located in the UK). The presence data was contained implicitly in the steady media transmission between both endpoints. Users had to look at the large screen to find out whether there was any co-worker present at the remote site. Later, in the *Portholes* project [Bux95a, DoBl92, HaHo94] abstract place-holders have been introduced to indicate a person's availability and willingness to communicate.

RAVE and Portholes are typical examples for early presence systems accompanied by numerous studies on user interaction and presentation techniques for presence information. These systems hence represent a specific class of cooperation environments we will have a closer look at in the following subsection. The findings will be set into relation with modern media spaces that are targeted specifically to mobile users having appropriate devices. As example for mobile environments supporting user presence Section 2.4.2 describes the *GEOCOOP* project, which constitutes a physical media space providing location-based conferencing services for mobile users.

2.4.1 Telepresence Systems

Creating virtual proximity has been the driving factor for the development of groupware systems in networked environments for more than two decades. Since 1927, when the first two-way video call was demonstrated by AT&T Bell Laboratories, animated views of a remote scenery have been regarded the key to overcome the distance between two geographically separated places. An overview of the numerous CSCW systems developed in the late 1980s is given in [CoGr93] and [SCP91]. In this section, we give a short overview of the EuroPARC RAVE system and its successor, the Portholes system. These projects have been chosen because of their impact on today's presence systems. The key concepts for mutual awareness that have been presented in the Portholes project for the first time include *avatars*, the *door metaphor* and the concept of *reciprocity*. Moreover, the trade-off between privacy and accuracy has been discussed in several publications on these systems as well.

In the late 1980s, the *Ravenscroft Audio Video Environment* (RAVE) has been used throughout the scientific community as a platform for research and development of new paradigms in computer supported cooperative work. This central role is confirmed by the large number of publications on research projects with their focus on technical and socio-technical issues that arise from using a RAVE system. As mentioned before, one of the most popular installations was located at Xerox PARC in Portland, Oregon, and the Xerox EuroPARC in the United Kingdom. A permanent video transmission allowed for employees at both sites to have a look into the recreation area of their transatlantic counterpart.

The RAVE environment and follow-up projects based on its awareness paradigm have demonstrated a number of fundamental concepts of presence systems. The most important characteristic of presence systems since then is the abstraction of real-world data. While the RAVE system used a steady video stream representing a live scene from a remote location the Portholes user interface displayed abstract icons to show the remote party's presence state. [Bux95b] Cameras mounted in the employees' offices enabled watchers to peek into colleagues' faces without even leaving the room. The presence system has combined the live image taken from another person and this person's abstract status description in a single view. An example for a gallery of colleagues' images is given in Figure 2.1.

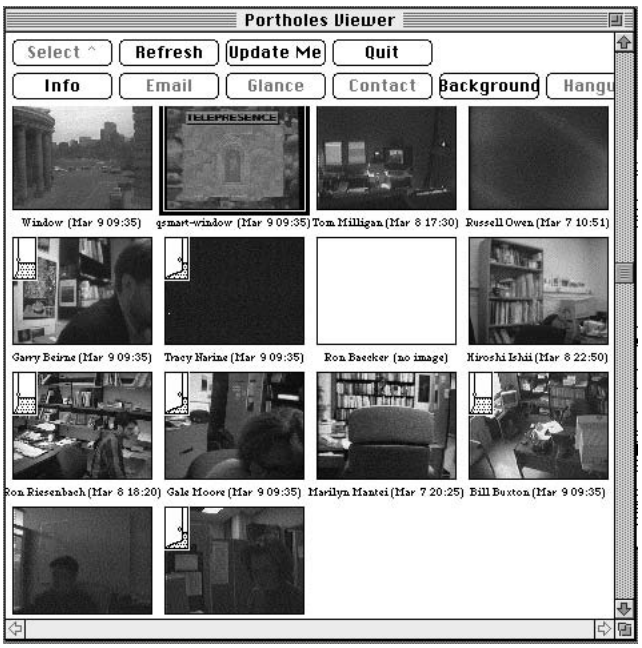

Figure 2.1: Example view of the Portholes system (source: [Bux95b])

To save bandwidth, the continuous video transmission has been replaced by series of pictures taken in periods of several minutes. Experience has shown that there is no need for a higher update frequency at all. [NLM+97] A study on a similar presence system even shows that human watchers are able to observe implicit patterns in presence history and give a reasonable forecast of a colleague's availability (cf. [HiBe03]). Therefore, a resolution of minutes seems to be adequate to get a rough impression whether or not a particular person is available for conversation.

Groupware systems that care more about privacy can use image manipulation functions to obscure recorded pictures (cf. [BEG00]). A common technique is to apply distortion filters preserving only vague shapes of persons and objects seen on a recording. Given that the camera is directed to a single workplace, and no other person ever occupies that space, the distorted image is sufficient to detect the person's presence in her office.

Destroying information can be used to synthesize interesting information from a set of samples. For example, a series of distorted images from the office camera mentioned before can be compared in order to determine whether someone is sitting in front of the camera's lens. If so, there should be a shape in front of most of the other objects in the recorded room. Detecting that shape is simplified after the information on coloring, different textures etc. has been removed from the images.

A popular example for abstract representation of presence information is the *door metaphor* used in the Portholes system. According to [Bux95b], this particular mechanism has been suggested by Abigail Sellen to parallel physical presence states within the virtual collaboration environment.[7] The door states being distinguished can vary from application to application. Usually, the following door states are available:

- open

  To indicate availability the door is left open. Visitors are welcome and might enter without request.

- half open

  When the door is ajar, visitors might glance into the office or knock to determine if the respective user is busy. This parallels polite behavior, where visitors stop by to determine unobtrusively if their contact is available and otherwise leave without disturbing.

- closed

  If the user is busy or not in her office, the door is closed. Visitors may knock at the door to request entry but there will probably a negative answer or none at all.

The door status may be annotated with additional information regarding the user's availability. For example, a note that reads "do not disturb" at a closed door denotes that requests for communication will be declined or forwarded to an automatic answering machine (e.g. a voice mailbox for audio communication). In some systems, restricted vocabularies are used to avoid illegal combinations of door status and annotations, e.g. an open door being associated with the "do not disturb" note. Some free-form text may also be used to publish a detailed description of the current activity. For example, a user could indicate a longer period of absence, like "out of office until January, 2nd".

To contact a subscribed user, the application's graphical interface could be equipped with a button to establish a suitable communication channel. RAVE [GMM+92] is an example for a system that provides a number of buttons that allow users to select a specific degree of engagement in collaboration. Starting from one-way video (either showing a recording from a

[7] See chapter 2, "Living in Augmented Reality", of [CLR95] for a detailed description of the sensor that was used to map the physical status of an office door to its virtual representation in a telepresence system. The sensor was implemented using a computer mouse mounted next to the door.

remote office as background scene, zapping from one office to another, peek into a specific office for a short time), over a shared office mode (two-way video), to focused collaboration using synchronous communication (audio/video call).

There have also been tests with controller entities that forced users to indicate the intention of a communication request. [GMM+92] However, we envision that users in a loose conversation as is typical for asynchronous communication relationships—e.g. instant messaging, email—do not care much about formalistic descriptions of the ad-hoc messages being exchanged. Especially instant message conversations have turned out to be very dynamic, not necessarily following established formal communication patterns. [NWB00, GrPa02]

In addition to these features, many modern groupware systems implement a mechanism to guarantee *reciprocity* of information exchange. In an environment where privacy of sensitive data is an issue, there is always a trade-off between user-specific control over personal information being revealed to subscribed watchers and the accuracy of published presence data. At Xerox, project participants found this problem to be less important for their work regarding the anticipated productivity gain. Thus, there is only little privacy protection available for the Portholes system.

The Portholes system [DoBl92] has been used in several other projects as presence-enabling technology, e.g. the Ontario Telepresence Project[8] at the University of Toronto. [NLM+97, Bux95a] In [Bux95a], Buxton has sketched his view of what he calls *context-sensitive interaction*: a media space that integrates multi-modal input/output interfaces to provide human users with a smart working environment. The system was designed to manage application context automatically in the background while specific events (like a human who enters the room) can cause foreground interaction with a user.

The environment is dubbed "smart" because changes in context may result in a smooth transition of foreground activities, e.g. changing from one communication channel to another without user interaction. Thus, a user might walk into another room where no video device is available. A video conference then puts the video channel on hold and switches to an audio transmission without lip synchronization, floor control etc.

The Ontario Telepresence project example shows the natural role of a presence system: To enable the context-dependent switching of communication channels, sensors are permanently tracking the users' locations, activities, available bandwidth, signal quality etc. The raw sensor data is aggregated into an abstract description of a user's devices and communication facilities.[9] Together with additional contact information, user preferences and perhaps other application data, the synthesized description comprises the context that represents a human user within the media space. As in other virtual groupware environments, the context object has a set of attributes describing the characteristics and the availability of the user's communication facilities.

2.4.2 Location-based Services

Although media spaces initially have been designed to bring together users independent of their current location, technical equipment and social context, location information always has been an important aspect of the presence status of remote peers. For a long time, location information

[8] The project's homepage can be accessed at <http://www.dgp.utoronto.ca/tp/tphp.html>.

[9] As a smart working environment, the system additionally provides support for tasks that are not related to interpersonal communication, e.g. illuminating a room as soon as a user enters. As these functions are not relevant for the subject of this thesis, we will focus on the system's facilities that are related to presence services.

was only implicitly available as for example in the RAVE system. Here, you had to know where the two-point video link ended to infer the exact location of the people being visible. Even for wide-area communication services such as the Portholes system, location information only played a marginal role because location support only meant that static meta-data was associated with a number of sensors at a certain place.

A dramatical change of the role location information had in presence systems was caused in the late 1990s by mobile devices with wireless connectivity. Equipped with a mobile phone, a personal digital assistant or similar, users could be located by tracking their devices' positions. Today, cost-efficient GPS (*Global Positioning System*) sensors even allow for users to determine their geographic position at a high precision independent from any provider-specific infrastructure. In addition, indoor positioning is getting popular as many buildings are equipped with wireless infrastructure that facilitates triangulation of radio signals, or sensors have been disseminated at strategic places in a building to track users wearing tags that emit radiation at a specific frequency when activated (e.g. RFID technology).

Recently, several research projects have been investigating the benefit of location information on interpersonal communication. The most obvious example for added value brought by transmitting location information when initiating a phone call is the `911` emergency call service for commercial mobile radio services regulated in Title 47, Sec. 20.18, of the US code of federal regulations.[10] Other examples for added value generated by location information are fleet management in logistics applications and conferencing systems. A number of online games also use geographic coordinates to build communities with specific interests, e.g. a treasure hunt that combines virtual scenarios and places in the real world.

Service	**Positioning**	**Proximity Detection**	**Location Distribution**	**Application Logic**
Finding Non-Portable Resources	+	-	+	+
Finding Persons	+	-	+	o
Resource Tracking	+	-	+	o
Conferencing Support	+	o	o	+
Presence Applications	+	-	+	+
Security Alert	+	-	+	+
Maintenance Information	o	o	-	+
Patient Information	-	+	-	+
Location Information in Emergency Services	+	-	o	+
Push Services	o	o	o	+

Table 2.1: Application building blocks (source: [OKB+05])

A systematic evaluation of current positioning technologies and the use of location information to facilitate the setup of multimedia conferences has been documented by Ott et al. in [OKB+05]. The research project *Geo-Based Services Enabling Cooperation* (GEOCOOP) has

[10] See <http://www.access.gpo.gov/nara/cfr/waisidx_03/47cfr20_03.html>.

identified several building blocks of location-based services, i.e. self-contained modules that describe distinct aspects of the technical infrastructure required to create location-aware applications.

The classification of building blocks given in [OKB+05] reflects the close relationship between location-based services and presence systems: The authors identify the positioning system as the first building block, comprising devices that generate low-level sensor data describing the location of a person or device within a specific reference system (e.g. geographic coordinates or descriptions of civic locations such as postal addresses). The information generated by these sensors can be distributed either in a local environment or published beyond organizational boundaries using a standardized protocol for distribution of presence information. The distribution infrastructure is defined as a specific building block as the actual dissemination protocol may be adapted to the needs of the particular application. The specific application logic is regarded as the third building block that defines the semantics of location information and the processing of the data received from the sensors.

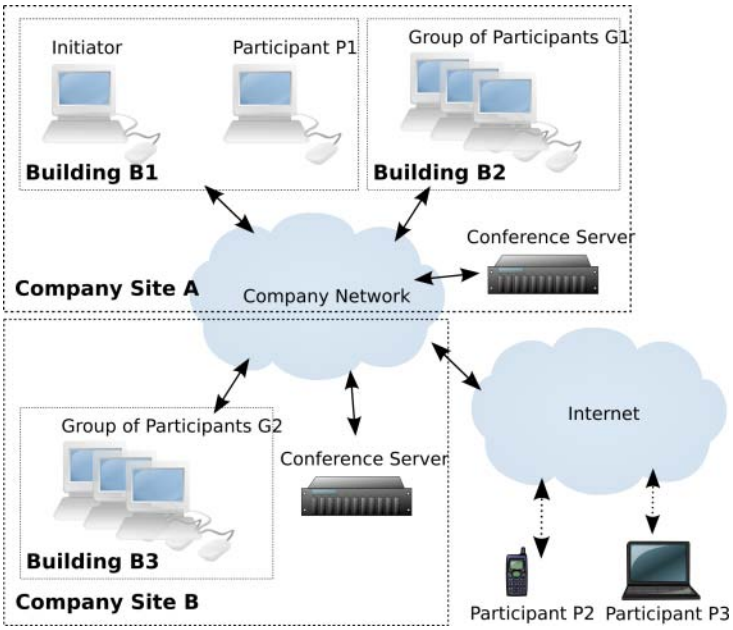

Figure 2.2: GEOCOOP demo scenario (source: [OKB+05])

An important result of the GEOCOOP project is illustrated by Table 2.1 that was taken from [OKB+05]. The rows in this table represent applications that use one or more of the building blocks denoted in the column headers. A plus sign (+) within a table cell marks a building block as required for the respective service, while "○" denotes optional use and "–" indicates that the building block cannot be used with the service at all.

Even without going into the details of the applications listed in Table 2.1, it is obvious that location information plays a significant role for specific kinds of presence applications. To demonstrate the benefits of this technology for enterprise communication, an existing conferencing server has been enhanced by functions for location-aware session creation. The core

component besides the *Asterisk*[11] conferencing server was a SIP-based presence service for distribution of location information.

The basic architecture of the GEOCOOP service targets at a large company with several organizational units at different geographical locations. A company network is used to interconnect company sites as shown in Figure 2.2. In addition, VPN connections from the Internet may be supported to let employees on travel participate in teleconferences as well.

The conferencing service provides a specialized conference planner that interacts with a conference server and a local SIP client to initiate ad-hoc meetings or schedule formal conferences to be held in the future. The graphical user interface of the planning tool allows for presentation of pre-defined maps of buildings to locate prospective participants and see their current presence status as depicted in Figure 2.3. In addition, the conference service allows for allocation of resources such as meeting rooms and presentation equipment. The conference then can be held as a face-to-face meeting or as a hybrid meeting with remote participants using the network for audio/visual communication.

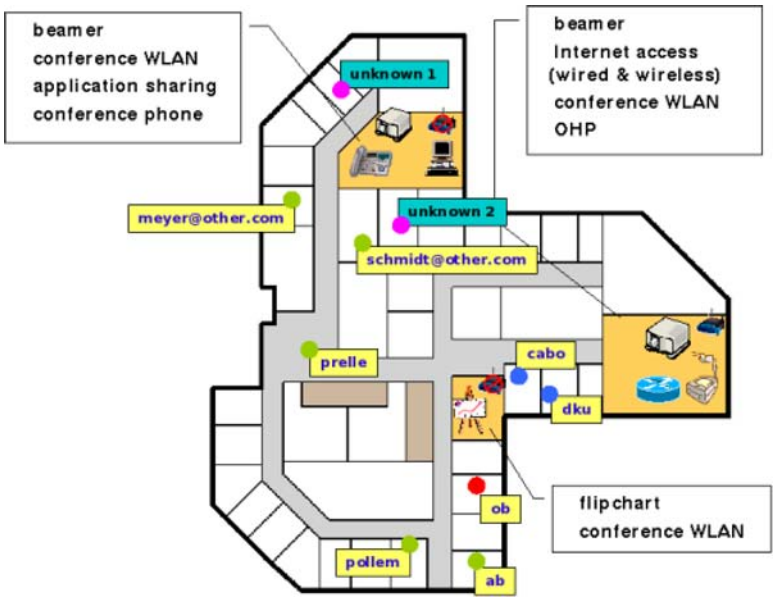

Figure 2.3: GUI-based conference initiation (source: [OKB+05])

In the GEOCOOP system, the presence service is used for dissemination of location information and abstract status descriptions as well as conference invitations. To interact with this specialized conference service, specific software agents are required on the users' local systems. Thus, the SIP-messages for conference invitation can be augmented with session-specific parameters such as resource plans, policies, or network parameters for accessing the local network in the meeting room. Users who want to accept a conference invitation then may choose to attend the conference in a dedicated meeting room or via the teleconferencing service. The GEOCOOP service thus is designed to support conference initiation and does not enforce a specific policy. Automated decisions of the system such as the teardown of a teleconference are controlled by customizable policies that are configured by the conference initiator.

[11] Available at <http://www.asterisk.org>.

2.5 On User Acceptance of CSCW Systems

Accompanying the development of groupware and CSCW systems in the 1980s and 1990s, several studies have been performed about user acceptance of presence technologies (see e.g. [MBS+91, NLM+97, Rou99]). The following list shows some important results that have influenced the design of modern presence systems:

- The quality of audio-visual links to remote sites is worse than physical presence, therefore causing less interaction between local and remote parties to take place. Studying the CAVECAT system which was similar to RAVE, the authors of [MBS+91] found that local individuals are "more present" than distant colleagues when the signal quality of their single audio/video link deteriorates.

- Awareness of persons in distributed groups can be achieved by publishing abstract descriptions instead of live video. The level of abstraction depends on the granularity of information that is necessary to represent the watched person's availability status.

  Abstract presence descriptions additionally are equipped with meta-data describing the context in which the presence data has been collected or published. Typical meta-data items are sample time, user name, type of presence document etc.

- Presence applications should support automatic generation of status descriptions to avoid manual updates. Simplistic approaches like common instant messaging applications or the presence system for telephone users described in [MiSm00] show that users rarely update their availability information. Instead, sensors could be used to determine a user's current activity and set appropriate status flags. Examples for simple but effective sensors are screen-savers, determination of idle time (e.g. `utmpx` in UNIX-systems), or login information. In addition, the call-status of accessible phones, calendar data, meeting schedules, etc. could be used.

  As seen above, modern office equipment and home appliances can provide extensive data to describe a user's current occupation. The more sensors are available the more accurate information can be synthesized from their data. Numerous research projects have proposed mathematical functions dealing with incomplete or contradicting data. A detailed discussion of this approach can be found in Chapter 4.

  Another interesting approach to estimate if a user may be interrupted in his current activity is described in [HFA+03]: Hudson et al. develop statistical models to predict the best occasion for approaching another user. Their simulation shows good results especially for determination of bad situations, i.e. when the other party better is not disturbed.

  Hudson et al. also show that the quality of sensor-data always depends on the habits and preferences of the particular user being observed by that sensor. When the authors prepared their study on the interruptibility of users in specific workday situations, they have found that the test person's doors where open most of the time even if visitors were not solicited to enter. The quality of presence indicators thus depends not only on the sensor itself but also on the user's preferences. A sensor-based presence system therefore must be highly customizable in order to operate properly without user interaction.

- Integration of available communication services in a homogeneous application interface. As described in [DoBl92], the Portholes system facilitates this by presenting several buttons to establish a specific communication channel with another user. Example services include video-conferencing ("office sharing"), one-way video, asynchronous text-messaging etc. The interface can be extended by new buttons that are bound to user-defined functions. A function is called by the Portholes system whenever its associated button is pressed, and is expected to initiate a communication relationship with the remote user.

- Event history can be used to determine the current presence status of users. As Terveen et al. motivate in [TMA+02], personal history is a valuable indicator of current and upcoming events. Users often refer to their own history when talking about their activities and their intent. When trying to automate determination of presence status, back-references to event history could provide indications on recurring patterns in the user's behavior.

- Privacy protection in presence systems has been discussed since the first groupware systems with awareness mechanisms have come into existence. Clearly, presence applications may automatically reveal sensitive data to unauthorized persons if authorization functions are configured poorly or have not been implemented at all. Therefore, modern presence protocols support authentication and encryption by design. However, applications still have to supply a robust concept for identity management and must provide intuitive mechanisms for configuration of security functions.

  The usability of secure presence applications includes not only requirements on the interface design to protect users against configuration mistakes that lead to security breaches. In addition, systems must provide functions for retrieving watcher information (*reciprocity* of subscriptions), and to control access to one's presence data. The latter includes manual as well as automatic blocking of requests (either a priori or when a subscription is active).

In addition to presence awareness and instant notification of received messages, Rennecker and Goodwin in [ReGo03] have identified the following features specfic to instant messaging systems that are contributing to this application's popularity and at the same time have consequences for productivity and workplace organization:

- *Within-medium polychronic communication*:

  Discussion threads with different people may exist at the same time, independent of each other.

- *Silent interactivity*:

  The contents of text-based interaction with remote peers cannot be overheard by people who are physically present. Users of IM systems therefore can manage multiple informal communication threads at the same time while usually the audio/visual communication channel is limited to one conversation at a time.

- *Ephemeral transcripts*:

  The authors postulate that the contents of a messaging thread should not persist after the messaging application has been closed. While this requirement may help protecting the privacy of incautious use, it gives no additional security against eavesdropping on the wire. Moreover, logging of transcripts for documentation of work-related communication will be the normal case for office use.

As the factors listed above directly influence the usability and acceptance of a presence system, our design is geared towards the measures suggested by the cited studies. The general requirements for our design have been deduced from an analysis of existing groupware systems and their technical basis, the results of which are presented in Chapter 3. To make this discussion more concrete, the following section introduces a typical use case that illustrates the benefits of automatic presence services.

2.6 A Use Case for User-Controlled Presence Aggregation

The introduction to presence services in this chapter has denoted various facets of presence information and depicted numerous application scenarios that can benefit from presence services. For example, presence systems could facilitate the monitoring of appliances or help planning and setup of conferences as described in Section 2.4.2. In our research work, we have focused on presence as an *enabler for interpersonal real-time communication*. The envisioned service provides information on the availability and willingness of human users to participate in a conversation, and facilitates the negotiation of interactive media channels between peers during call setup. The goal is to support users in their communication habits rather than forcing them to obey a specific communication pattern or use communication media that are not appropriate for the conversation topic.

In this section, we present a simple use case that illustrates the presence service from a user's perspective. The user is assumed to work in a large company that has excellent Internet connectivity and has replaced all PSTN phones with modern SIP-based IP phones. The user shares his office with a co-worker, and both use the same IP phone, but with a dedicated "line" for each of them. The user in addition has a networked home with a desktop computer and possibly some "intelligent" devices that observe the presence of the user's bluetooth phone or notebook computer. These two devices are typically used when at work or on business travel, hence indicating work context rather than being at home with the family. As a result, three distinct presence zones can be identified as visualized in Figure 2.4.

The figure shows a common setup for this scenario: A public SIP server located somewhere in the Internet acts as a proxy where SIP requests for this user arrive. This server also receives information on the availability of certain communication media supported by the user's devices. For example, the hardware IP phones register with the public SIP server and thus are always online. Other devices such as a WLAN access point are used to detect personal devices of the user and do not offer communication services.

Some devices can occur in more than one presence zone as indicated for the user's notebook computer and his mobile phone. These offer communication services for a specific presence zone while associated with this particular zone. A probability value given with the presence information for this device indicates the exactness of this information. For example, if the user

has an IM client installed only on his notebook computer, an interactive text channel may be announced for the presence zone where the notebook computer is located. As this channel is usable for communication when the user is near the notebook computer and is willing to join a chat session, the probability might be set to a value less than 100 % if it is unclear whether or not the user is around.

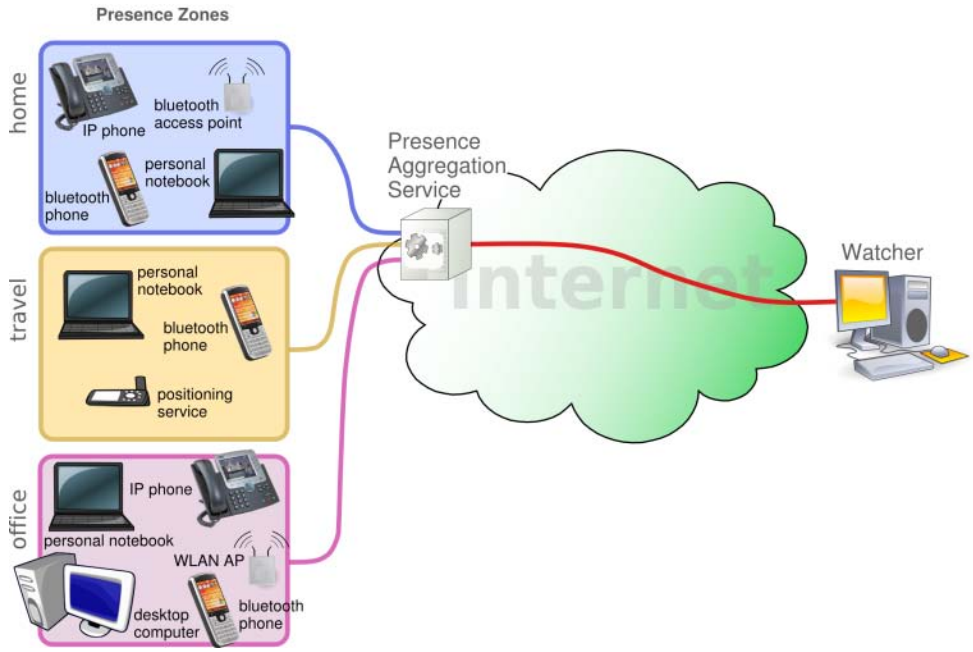

Figure 2.4: Presence zones for use case scenario

The channel abstraction we are using in this thesis is important for negotiation of appropriate communication media. Figure 2.5 shows the channels possibly offered by the office presence zone with their media description. We assume that the personal notebook is used for instant messaging but not for audio communication. The announced media type therefore lists only *interactive text* as the channel description. The hardware IP phone offers interactive audio communication, and the desktop computer is equipped for video conferencing. The mobile phone has been configured to divert incoming calls to the voice mailbox, so only one-way communication without user interaction is possible. In addition to voicemail, the mobile phone accepts text messages using the short message service (SMS).

The published channel descriptions from the three presence zones are aggregated by the public presence aggregation server that manages the subscriptions from external watchers. A watcher who wants to begin a conversation with the respective user may refer to the aggregated channel descriptions shown in Figure 2.6.

In this example, the user is at home and does not accept incoming phone calls from external watchers (i.e. persons who are neither family members nor friends). The aggregated view on the available media channels therefore lists the interactive audio and video channels to be in the state `CLOSED`, i.e. not available. Communication is possible only via interactive text messaging or asynchronously by talking to the voice mailbox or sending a text-based short message.

Based on the information of supported communication media, the watcher's presence application can display a dialog that shows the current status of each channel together with a

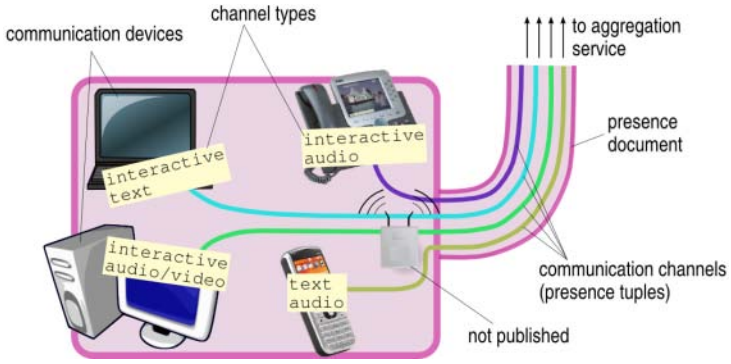

Figure 2.5: Publication of distinct media channel descriptions

short description or an icon representing the channel type and whether or not this is an interactive channel. Now, a watcher who wants to begin a conversation with the subscribed user can choose between the available channels and the option to wait until a specific media is available again. If the watcher requests a specific channel that is currently not available, his presence application might track the status of this channel and try to contact the subscribed user as soon as the channel becomes available. The aggregation service thus can be used to enable multimedia communication similar to the call completion feature in the telephone network, where a trigger can be installed after a failed call attempt to establish the call as soon as the called party is available again.

2.7 Summary

In this chapter, we have discussed the use of personal information in networked environments to determine a user's availability for interpersonal communication. A brief overview of early groupware systems visualized the influence that technical evolution had on the role of awareness platforms. In particular, the ongoing merge of different network technologies—most notably of wireless access networks—based on the Internet Protocol turned out to have a great impact on users' demand to publish availability information. The increasing number of personal devices that are capable of asynchronous messaging (especially in terms of ad-hoc text messages) fostered the opinion that instant messaging and presence will be one of the most valuable business applications for the next years.

To get a better understanding of this application area, we have examined the use of the term *presence* in literature, and gave a concise definition of its use within this thesis. To avoid confusion with the widely adopted use of that term in the Internet today, our definition is compatible with [RFC2778], an IETF standard defining a model for presence and instant messaging.

The next step was to determine the key functions and drawbacks brought to a communication system by including presence-support. A close look on two projects where awareness of users and services is a major feature helped to identify the expectations users associate with a a good presence service. With *Portholes* and *GEOCOOP*, two existing approaches for user presence systems have been presented in detail. Portholes, being one of the first prominent systems for *computer-supported cooperative work* (CSCW). has introduced a number of features that have become fundamental concepts of modern groupware systems as well. The trade-off

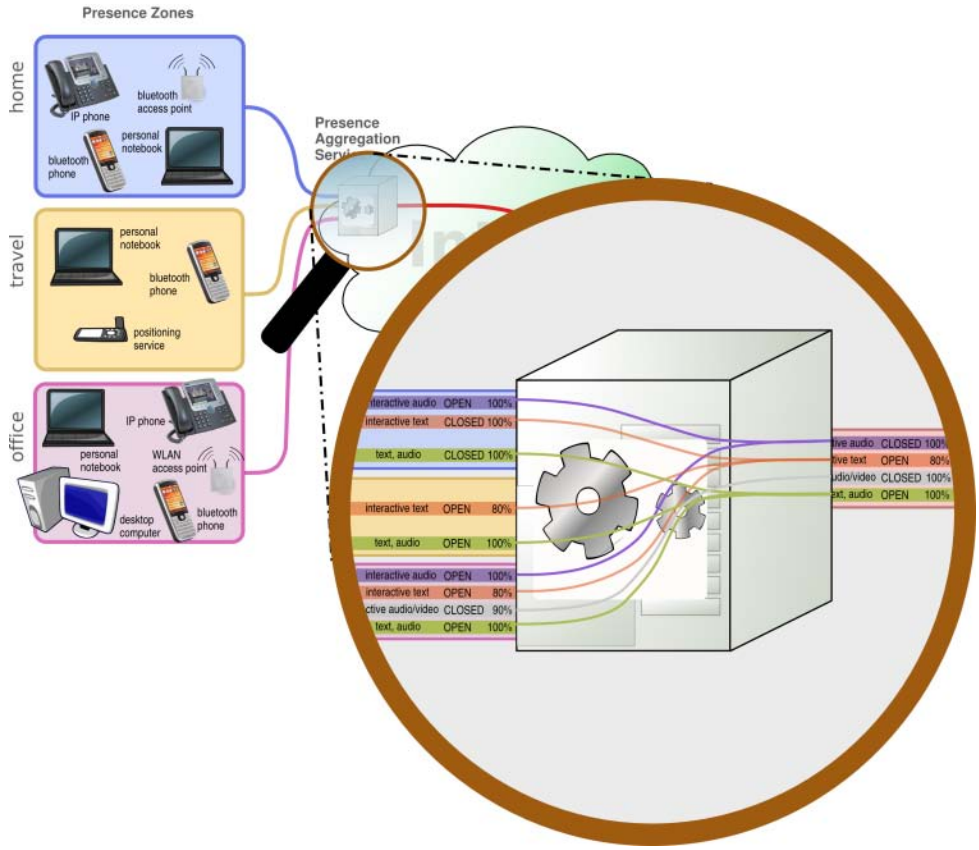

Figure 2.6: Aggregation of multiple channel descriptions

between privacy protection and functionality has been solved in a very pragmatic way for the specific environment of the Xerox PARC and EuroPARC research teams, and an initial impulse for a broad discussion of this topic was given.

The GEOCOOP project has focused on the use of presence information—and location information in particular—to enhance the instantiation of multimedia conferences. The project investigated the impact of different positioning technologies as a basis for location-aware conferencing and demonstrated the benefit of location information during the conference setup phase. Significant effort has also been put into usability aspects and the enforcement of privacy policies.

After close examination of these examples, we have synthesized a typical use case for a wide-area communication service. The use case exemplifies various expectations in a system like this as well as various forms of interaction. Based on these observations, we will deduce requirements on a presence-aware communication system in the next chapter both from a user's perspective and from the technical side.

Chapter 3

Requirements Analysis

In the previous chapter, we have identified use cases for wide-area communication services based on typical application scenarios of presence-aware communication and have summarized the factors known to affect the user acceptance of these systems. Taking our analysis as a background, the major issues of groupware systems in heterogeneous wide-area networks are discussed in this chapter. Our special focus is on the impact on user awareness within these applications and the deterioration of perceived service quality for users working with them. The following section, Section 3.1, discusses the limiting factors from a social-technical perspective that we have identified from the documented issues with existing groupware systems. Based on our results, we propose a set of guidelines for the design of presence applications that should be followed to enhance the user acceptance of computer supported interpersonal communication.

Given the usability requirements, we have deduced functional requirements for a presence service that is de-coupled from any specific groupware application. These requirements are discussed in Section 3.2, assuming that the system serves as a generic building block for virtual collaboration and is not restricted to multimedia conferencing services in CSCW applications. A discussion of suitable large-scale distribution services in Section 3.3 then identifies the *Session Initiation Protocol* (SIP) as a good infrastructure for the dissemination of presence information. The security issues that arise from the publication of personal presence information are discussed in Section 3.4. Finally, Section 3.5 concludes the chapter with a brief summary of its contents.

3.1 Guidelines for Presence Application Design

From a user's perspective, the social aspect of an application that facilitates interpersonal communication is more challenging for system design than technical restrictions are. User acceptance therefore is primarily founded in a sound interface and a functional design. Technical brilliance, in contrast, is not a requirement in itself, but may be necessary to achieve the goals. Looking at the most successful computer systems around, many examples can be found that reinforce this statement. (Though some systems certainly provide a usable interface in conjunction with leading-edge technology such as the *Mobile Interactive Spaces* shown in Section 4.1.3.)

One important aspect of presence services is the trade-off between information and interruption: users want to become aware of other user's activities without being disturbed by system-

generated notifications of status changes. Even worse, in [Nie03], Nielsen predicts a significant decrease of productivity due to interruption of tasks caused by instant messaging systems. Although the exchanged messages are very short and do not take much time to type, the context switch between actual task and the message's topic can take up to ten or fifteen minutes. Re-establishing work context after various interruptions would eat up most of the productive time a user has. The author hence demands non-intrusive awareness tools that do not attract the immediate attention of the user. [Nie03]

In this section, we summarize the results of research on interface design for awareness systems and messaging applications. The goal is to characterize the inherent trade-off between minimizing the information overload (what Nielsen has dubbed "information pollution" [Nie03]) and self-determined use of the data provided by a presence system. The achievements of several years of psychological and linguistic research showing patterns of interpersonal communication are mentioned to motivate our approach to describe a user's presence status. However, as this thesis is about a middleware to convey this information between distant hosts, no detailed guidelines for user interface design will be given. Instead, we characterize the application's functionality together with its intended use. The focus of our discussion is laid on the maximum amount of content information the presence service must provide to enable these functions; end-user applications that use this service usually have a user interface displaying only a small set of presence information. Upon explicit request—initiated e.g. by clicking into the application's window or pressing a keyboard shortcut—more information is revealed and additional functions may be offered over an extended user interface.

Depending on the actual design of the presence service and the protocols being used to implement the functions postulated in this section, additional issues must be addressed, especially regarding the robustness against failures and abuse. A detailed discussion of the options and threats for system design can be found in Section 3.2.

3.1.1 Minimizing Information Overload

Although there are no meaningful statistics to quantify the impact of asynchronous update notifications generated by a presence service as Nielsen has quoted for instant messaging, there is no doubt that a productivity loss can occur if user interfaces are too intrusive. On the other side, Dourish/Bly and Herbsleb/Grinter report of instant messaging to have improved communication between co-workers in a software development team, hence resulting in a performance gain. [DoBl92, HeGr99] Statistics on contents and purpose of instant messaging are contradictory, depending on technical skill and social context of user groups (see also the findings of Isaacs et al. in [IWW+02c]). The existing trade-off between disruptiveness and timeliness of user information thus is always task specific and cannot be solved in general.

User interfaces therefore are traditionally designed in an iterative fashion, i.e. layout improvements are evaluated in subsequent field studies performed by users with average technical skills. Recent proposals (e.g. by Chewar [Che03]) aim on a more formalistic framework to define and evaluate scenarios for performance tests of user interfaces. The effects of update notifications on visual displays have been addressed in particular in [HKP+03, CCH01, DaKr04, Che03]. The impact of task interruptions in general is documented by Speier et al. in [SVV97], a description of several experiments to examine user's ability to complete tasks of different complexity after being disturbed.

3.1.1.1 Limiting the Notification Rate

Concerns of "information overload" have been documented since digital communication has become omni-present in large organizations in the early 1980s. Nearly two decades before bulk e-mail and unsolicited commercial e-mail revealed the weakness of the global SMTP-based message exchange regarding integrity checking and prosecution of malicious use, Denning in [Den82] has addressed the upcoming problem of "electronic junk". Hiltz and Turoff in this context demanded the mitigation of information overload by filtering functions that enhance computer-mediated communication instead just replicating traditional interpersonal communication. [HiTu85]

A valuable result of the efforts in usability research cited before is the observation that presence information should be displayed in a fashion that does not attract the user's immediate attention. A presence service can support this goal by providing structured content information that allows for selective filtering, prioritization and throttling of update notifications as visualized in Figure 3.1. In the following sections, we discuss the advantages and disadvantages of these three techniques when applied by a presence source, i.e. either an originator of that information or an intermediary that aggregates status descriptions from multiple sources.

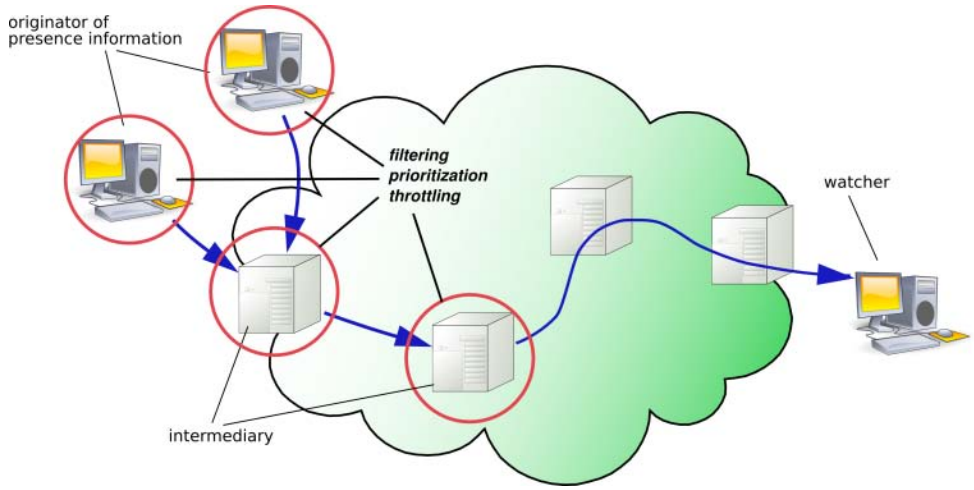

Figure 3.1: Server-based limitation of the update notification rate

Filtering

The description of information items contained in a presence status notification should be explicit enough to allow filtering of unwanted events. This includes unique identification of presence descriptions for various communication channels offered by the publishing entity as well as an expression language that allows attribute-based filtering of information items, e.g. to exclude information about unwanted or not supported types of communication. Filtering in general can be implemented at one or more points along the message flow: at the sender, at the receiver, or at an intermediary within the network.

To provide event filtering at the sender or at an intermediary, the presence service must define a specification language to express filter functions. Furthermore, protocol elements to

install and remove filters from these network nodes are required. In addition, presence status updates must have specific characteristics to facilitate effective filtering. First, it must be possible to distinguish information items contained in update notifications either using the message structure or by explicit identifiers. Second, each presence information item must consist of attribute-value pairs built from primitive data types that allow for simple filter expressions according to the actual data type. For example, numeric data types should enable range checks and arithmetic operations while boolean types allow combination of predicates. Complex data types may be constructed from primitive types using syntactic conventions, e.g nesting of elements in XML.

An advantage of event filters provided by the network compared to receiver-based filtering at the watcher's host is the limiting of of update notifications sent through the network. Especially for thin clients that have low processing power or where network connectivity is affected negatively either because of low bandwidth, frequent outages or high costs (factors that may occur e.g. in wireless cellular networks), a sender-based filtering would be appreciated to minimize the amount of traffic.

Prioritization

Another option to minimize the load of information to be processed by a presence subscriber is to annotate information items with a priority value. The watcher's presence application then could be configured to filter out presence events that fall below a given threshold, or generate immediate alerts if an event with a high priority occurred.

A user should be able to define custom alerts for events with high priority. For example, if the user is about to visit a remote peer physically before a certain deadline has passed, he could initiate an audible alert to be played and a popup window with a modal dialog opening as soon as this particular contact appears available. A similar solution with limited expressiveness is described by Cadiz et al. in [CVJ+02]; the tool's user interface is integrated with the Microsoft Windows Desktop. An extensible sidebar allows for a customized display of several events to track user activities, web page updates (e.g. weather reports, stock tickers), and incoming messages. By default, the sidebar elements display the current status of the watched object, i.e. a minimized version of a dynamically changing graph, an abstract description of a user's presence status, or just some digits indicating the current stock exchange rate. Any of these events can be combined with an alert specified via a graphical dialog interface.

Event throttling

Filtering and prioritization are useful to suppress transmission of unwanted information and for selective display of items that are of user interest. Both techniques require user interaction to determine what kind of information the user wants to see. As this does not protect the receiver from being overloaded with update messages, a mechanism for controlling the sender's notification rate on a per-message basis is to be provided.

Similar to well-known strategies for traffic-shaping at the IP layer [RFC1633, RFC2475] the presence service is allowed to transmit only a given average of presence information within a specific period of time. For example, if the threshold was set to 1 kilobit per second, an update notification of 80 bytes average size could be sent every 0.64 seconds, while notifications of 480

bytes must wait for 3.84 seconds until the next message is allowed.[12] This is in particular useful for text-based protocols such as the *Extensible Messaging and Presence Protocol* (XMPP, see Section 3.3.3.1) where protocol PDUs and content data are represented in XML and thus can have significant length if not compressed. Especially for wireless access networks with a high bandwidth-delay product the throttling can help saving costs without significant degradation of service quality.

Looking at throttling in cellular networks, the need for automatic rate adaptation becomes obvious: At first, the quality of data transmission may vary (due to changing cell utilization or spurious connectivity loss due to roaming) resulting in variable bandwidth. Moreover, compression rates may depend on the content actually being sent, hence the net size of the presence information being transmitted on the wireless link is not predictable. Since both factors influence the throughput of the wireless link, the throttling parameter has to be reconsidered from time to time.

In some cases, it may be convenient for an application to limit only the rate for certain types of events, e.g. if high-volume sensor data is distributed over the Internet. Throttling therefore should be combined with filter expressions to select the event notifications a specific throttle should be applied to.

3.1.1.2 Aggregation of Notifications

Another reasonable approach to minimize the information overload is to provide a user with a simplified version of the originating data. While techniques like filtering and throttling operate on the source data without modification, *information aggregation* can be used to transform a set of input data to a higher abstraction level. This process's result then is presented to the user in an appropriate way as previously discussed.

The aggregation serves the purpose of synthesizing high-level information on user presence from system-specific data like raw output from motion detectors, temperature sensors, screensaver status etc. With enough background information, this low-level data can be used as indication for the presence or absence of certain users. However, as remote subscribers typically lack this background, the unprocessed sensor output would be useless for them. Generating a more abstract view on that data thus means to lose detailed information about the status of specific sensors in favor of getting an interpretation of this data in its specific context. As the latter usually is more adequate for distributed awareness systems, information aggregation is an important aspect of an automated presence service.

Sensor fusion

One approach to combine syntactically unrelated data from multiple sources is through mathematical operations. Once having collected a set of numeric sensor data that represents a closed system's state at a specific point of time, a mathematical function can be applied in order to generate an aggregated view upon that system. This technique, called *sensor fusion*, has several advantages: First, the formalistic approach allows fast processing of high-volume sensor data in a deterministic fashion, only depending on current and past input data. When speed is not a big concern, the processing step might also apply learning algorithms known

[12] Note that these figures are rough estimations for PDU sizes of binary presence protocols and uncompressed HTTP-style protocols using UTF-8 encoding. Header overhead for TCP and IP is not counted here.

from neural networks to improve the result's quality by interpreting the data. An example for a system that uses learning algorithms for recognizing the physical context of a device (lying on a table, being carried by hand or in a suitcase etc.) is given by Schmidt et al. in [SAT+99] (cf. Section 4.1.2).

A disadvantage of sensor fusion is the high complexity for users to understand the mathematical functions. Moreover, when inference algorithms are used to predict sensor values, an initial data set must be created for the learning function to work upon. For sensors giving hints about a user's presence status, this particular user has to generate representative input reflecting habits and preferences regarding availability and willingness for communication. In [SAT+99], the authors briefly show the initial setup phase for a sensor array. Though the learning process described therein is very fast, a set of pre-defined contexts must be provided. Every context is defined with a set of Prolog rules that map sensor values to a more abstract representation.

As systems that automatically determine a user's presence status from given sensor data cannot guarantee correctness of the computed result, they must provide a function to override that value. The new status value should in turn be fed into the learning algorithm in order to improve automatic detection. Inconsistencies can occur if this override mechanism is used to arbitrarily set status values unrelated to the sensor data. The semantics of those overrides is not defined properly as long as no general presence rule can be derived. As a consequence, the quality of future aggregation steps may deteriorate.

The option for users to interfere with the sensor fusion shows the inherent drawback of automated systems, i.e. to map given input parameters to an appropriate output value, complete mathematical descriptions of the input and output domains are required to facilitate the automatic construction of an abstract concept for that application domain (the training phase of learning algorithms). If, e.g., an unknown—and possibly infinite—vocabulary shall be used to express user status, automatic reasoning is not a good candidate.

Level of Abstraction

In the past, several research groups have examined the level of detail presence information should provide to be efficiently processed by human users. Dabbish et al. discuss in [DaKr03, DaKr04] the results of their empirical study on the impact the degree of abstraction has on the prediction of user interruptibility. In their setting, a group of users was occupied with a specific task affording a certain level of attention (a computer game in this case). A second group of users was provided with information on the workload of the individual members in the first group in various forms, representing different abstraction levels. Members of the second group then had to choose an opportunity to contact specific persons from the first group based on the workload information they have been provided with.

In brief, users who had to contact another person had a better feeling when it was appropriate to interrupt her the more abstract the information about the remote party's workload was. In particular, obtaining a replication of the peer's desktop did not turn out to be useful as it was too difficult to determine the owner's interruptibility from the raw data.

Various studies on interface design for awareness systems also have found that a high level of abstraction is suited best for representations of presence information. Systematic research of multi-modal interfaces for *human-computer interaction* (HCI) evolved from the late 1990s and has delivered valuable results some of which can facilitate proper design of user interfaces for awareness systems. Descriptions of empirical studies can be found in literature ranging from

guidelines on structure and placement of graphical objects on a window manager's desktop interface to tests with innovative output devices creating haptic stimuli like heat or motion.

One of the most interesting displays supporting peripheral awareness of user activity is available in the AROMA (*abstract representation of presence supporting mutual awareness*) system described in [PeSo97, Ped98]. The authors describe their prototype to follow the two design principles of "radical abstraction" and "non-intentionality" (see [PeSo97]): Captured sensor data in this system is mapped to an abstract event like sound modulation or changing speed of a moving object. Depending on the activity level in the observed area (e.g. an office, a single workplace, or a conference room), the corresponding sound or movement will change accordingly.

Though there were several unconventional displays proposed for AROMA, the most surprising is possibly an electro-mechanical merry-go-round representing user activity at a distant location with its rotation speed. To avoid rapid changes of motor speed due to activity bursts, a decent hysteresis has been added. Thus, changes between inactivity and activity are smoothened and extend over a longer time. This leads to a memory-effect that allows recent activity to be observed for some time afterwards.

The major finding from studies of the AROMA system was probably the appropriateness of symbolic representations to indicate user activity. Tangible interfaces and haptic displays turned out to be as good as audio/video interfaces for peripheral awareness, sometimes even better. An important point made by Pedersen is that many people tend to read too much meaning into symbolic representations for abstract events: In [Ped98], she gives the example of moving objects on a video-screen to indicate activity. Depending on their number and their speed, people tended to read additional meaning into the displayed data. For example, when only few objects were present indicating low activity, these objects have been misinterpreted as specific individuals to be present in the observed room. In contrast, a large number of crowded objects was taken as an abstract indication of a high activity as intended.

A second observation from Pedersen regards the number of semantic mappings between sensor data and symbolic displays. If this number exceeds a certain limit (four, in this case), the display requires intellectual reading instead of only peripheral awareness. As a consequence, the information conveyed by our presence service must be abstract enough for support of peripheral information but must contain sufficient details to avoid misinterpretation by users. This means that a core vocabulary with strict semantics must be defined to express the presence status of individual users. Starting from a fixed set of status descriptors (e.g. `available`, `absent`, `busy`, `in-call`, etc.), state transitions and accompanying events must be specified to avoid implicit and semantically fuzzy vocabularies known from existing presence applications (see Chapter 4).

3.1.1.3 Prediction of Availability

Traditional presence services—as several other information systems, too—suffer from the fact that effort and benefit for information producers are not balanced: Providing detailed presence information in a timely fashion results in high workload at the publisher's side while there is almost no use in publishing this information. The consumer usually gets a high benefit from this information at minimal cost. Because of this asymmetry, the quality of provided information goes down if no additional incentives are offered for manual updates as empirical studies with groupware systems show (cf. [Mac90]).

An alternative of increasing the benefit for information providers is to lower the cost for updates. In the past, there has been much research on interface design to simplify the process of data collection, synthesizing information and publishing the result. After years of research on graphical user interfaces for desktop computers and mobile equipment with reasonable results regarding user acceptance the focus recently has shifted to multi-modal interfaces providing task-specific facilities for human-computer interaction such as tactile devices. Though there is great potential for further improvements in this area, the need for user interaction cannot be eliminated completely with this approach. Hence, several research communities have begun to exploit the feasibility of automatic inference of a user's presence status (see [TBH+04] for a brief overview).

From intuition, it is clear that inferring the presence status of an individual is a hard problem for both, humans and computers. While sensors can detect absence of people, it is not easy to determine whether a person is available for communication even if present. Worse, although the user's current workload may be guessed from sensor data a classification of activities as typically provided by human users is nearly impossible. Hence, the presence language's vocabulary for automatic detection is restricted to a fixed set of distinct values. A mapping from sampled sensor data to the presence vocabulary is based on an extensive analysis of past activities using statistical methods. The result usually is annotated with a numeric probability value that indicates the accuracy of that estimation. A detailed description of log file analysis is given among others in [BTS+02, HFA+03].

To identify representative activity patterns in large log files, data mining technologies can be applied instead of manual analysis. Prototype implementations using machine learning algorithms have been developed in several research projects e.g. by Microsoft [HKK+02] and at Sun Laboratories [TaBe03]. Fogarty et al. in [FHL04d] present an evaluation method that allows to determine the quality of a given mapping between sensor data and status forecast. An interesting result of their empirical study is that human prediction of status and automatic inference are nearly equivalent, i.e. about 80 % accuracy compared to the self-reports of watched individuals.

Fogarty's study on robustness of statistical models has two implications that can affect the design of a presence service. First, the best results are achieved if automatic inference is restricted to determination of workload as an indicator for non-interruptibility of the particular user. The second observation is that even human watchers cannot exactly determine whether a person is busy from just watching her activities. Obviously, there is a difference between subjective self-reports and objective monitoring. In other words, activities seem to exist whose workload cannot be measured in a typical office environment. Users therefore should have control over their personal status record being published by a presence service to be able to override values where automatic detection failed or was too optimistic. In conjunction with presence aggregation, these users may also combine the results from multiple presence sources, recursively applying automatic inference, or give explicit rules to synthesize the status value to be published.

Another issue with automated presence inference as described before is raised by the unification of sensor data: As the sensor network's output is mapped to a single value taken from a very small domain, it serves as an overall presence indicator for the particular user. If a subscriber knew only this value she would be aware that the desired person is available for communication but no contact information is available. As the following section will show, the selection of appropriate channels is a fundamental requirement for interpersonal communication. Al-

though the channels to be used could be negotiated after the first contact was made, this poses an unreasonable burden on the initiating side as the probability of call-completion decreases. A superior method is to publish detailed records on the availability of distinct communication channels since a user might be capable of answering quick questions via instant messaging but does not want to engage in a video-conference. With explicit information about available devices or communication services, a caller can select an appropriate medium for contacting the remote user from the offered set of communication channels. Thus, the presence service provides support for the negotiation of a preferred communication channel. The following section shows that this selection of a communication medium is crucial for interpersonal communication, as a meaningful conversation always requires a *common ground* to be established between communicating parties.

3.1.2 Patterns of Interpersonal Communication

As we have motivated before, interpersonal communication always requires a channel to exchange information between the involved persons. Besides this technical infrastructure, both parties have to establish a *common ground* for their conversation, i.e. they determine an intellectual context necessary to fill utterances with semantics. [McMo94] A large number of technical solutions has been proposed to support real-time communication over great distance. Some of these systems just try to replicate conditions that characterize face-to-face communication, others provide additional services for a more effective conversation.

In this section, we show how presence services can be used to negotiate communication channels that constitute the technical infrastructure to establish the common ground between communicating parties. To enable this negotiation, one party must offer a set of supported media it is willing to use for interpersonal communication. If another user wants to communicate, he has to select one or more appropriate media and send an invitation to establish a conversation. Section 3.1.2.1 argues that this pattern is an essential aspect of any meaningful conversation, and hence must be taken into account when discussing new forms of communication support. The process of communication channel selection from a given media set is described in Section 3.1.2.2. Section 3.1.2.3 concludes the discussion with a brief summary of the important role played by a presence service for negotiating communication channels that are needed to establish common ground.

3.1.2.1 Establishing Common Ground

From communication theory, we know that technical installations only provide rather incomplete replacements of face-to-face communication. [McMo94, HoSt92, AAS03, MIH+03, CoMa03] The problem of this approach is the substitution of real-world concepts like three-dimensional views, high-resolution video, gaze awareness etc. with low-quality imitations. Therefore, some researchers have argued that computer-mediated communication should augment physical communication channels rather than trying to imitate them. [HoSt92, MIH+03] As Axelsson et al. found out, patterns of interpersonal communication can differ depending on the communication channels' characteristics. [AAS03] More important, users are able to utilize these channels to yield a more effective information exchange.

Presence services are good examples for enhancements of traditional communication channels: Providing information on the availability and willingness of users at distant locations to

participate in a conversation, a presence service can give abstract hints about the status of a user to be contacted without trying to imitate physical awareness. The more focused the service the better is its quality. For example, publishing detailed information on a user's location does not indicate whether or not the user is willing to accept incoming phone calls (as he might be in a meeting or at a restroom). The availability to visit him is also not defined clearly: Even when in his office, the user might be in a consultation or be busy with other work.

Instead of publishing a detailed description of user activities or user location the presence system should distribute data that serves a specific purpose, in particular to indicate availability for communication. Examples for this kind of presence service are the widely deployed instant messaging systems like AOL Instant Messenger, or Microsoft's Windows Messenger. Status information conveyed in these systems denotes whether or not the publishing user is available for communication using that particular tool. In particular, if the user does not type anything on his keyboard for a while his status will reflect this in some way, e.g. indicating absence. However, as the user is still in the office, he might be available via other communication channels like phone or personal visit.

Considering the scenario depicted above, it is important to notice that individuals who want to initiate a conversation are able to arrange with the technical limitations, either because they have observed specific activity patterns of the watched person (e.g. being available for phone calls before 5 pm even if the presence system shows him to be away) or because additional information sources are available (e.g. a recently written email was received). [HiBe03] Thus, it seems to be natural for humans to deal with uncertain knowledge and to estimate the probability that a communication attempt succeeds. A presence service can support this behavior by presenting explicit information on the various communication options the watcher has, rather than showing only a single presence value as common in most presence clients.

3.1.2.2 Media Selection

When negotiating parameters for the communication session, not only technical settings like audio codec or video frame rate have to be considered: before any setup commands are exchanged the calling user has to select one or more communication channels to be used for information exchange with the called party. For example, a quick greeting at the morning might be sent via an instant message, with no immediate response anticipated. For sensible discussions, in contrast, the participants may prefer a bidirectional audio-visual channel imitating a real-world face-to-face meeting. As found by McCarthy and Monk in [McMo94], the selection of communication channels depends on the topic to be talked about. Thus, channels could be classified by their quantitative and qualitative dimensions. The decision of what channels to use for interaction therefore is a fundamental part of establishing the *common ground* for interpersonal communication. [McMo94, AAS03]

Now, if the party who initiates a conversation is in charge of selecting the appropriate media it would be inapt to override this decision without need. One reason for doing so might be the limited capabilities of the receiver system. If, e.g. a mobile user only has a pager device incoming audio calls are declined automatically simultaneously indicating the calling party's subscriber number on the pager's display. The called user then has the choice to call back or to ignore the attempt to contact him. Although technical limitations caused the communication to fail, control was always by the involved human users, not the technical system. This should be true even if the called party's communication environment were capable of accepting the call;

neither deflection nor acceptance should be performed automatically only if the user previously has instructed the application to do so.

While this behavior is widely accepted in user interface design, systems for multimedia communication often do not honor the basic rationale behind this, namely to keep the user in control of media selection. Systems that support *vertical integration* or *seamless communication* usually offer automatic fall-back to alternate communication channels if the initial call setup request cannot be fulfilled. Any media not supported by the remote party then is replaced by an equivalent media, still trying to establish the communication relationship. As long as original channel and its replacement share the same quantitative and qualitative class exchanging them without user interception is not critical. However, as soon as the replacement's capabilities are different from the original—either inferior or superior—it should be used only after both participants have agreed in using this channel instead.[13]

Note that it often makes little sense to provide an alternative channel with capabilities that are different from those of the originally requested channel, because the caller would judge it as not appropriate for the particular topic to be talked about. For example, users might refrain from talking to a voice mailbox when their message is too personal and they do not want it to be stored permanently. Or the caller wants to have immediate feedback as might be in case of a manager who is about to dismiss the callee.

As you can see, there is a trade-off between the caller's intention to establish a communication channel with specific characteristics at a certain time and the callee's capabilities that might not be able (or willing) to provide an appropriate channel. While in real life, a manager might force his employee in a face-to-face conversation things are more difficult for teleconferencing: At least for synchronous communication relationships as requested here, the callee has to agree with the establishment of a common media channel. If declined, no communication between both parties is possible.

This asymmetry between real-world and virtual environments can be overcome by introducing role-based access control in teleconferencing systems which in turn would need a very complex identity management to be really secure. As without this access control mechanism, the unrestricted call facility would be too obtrusive, it has been completely abandoned from most of the widely deployed call setup protocols.

3.1.2.3 The Role of Presence Services

A presence service can provide useful information for a caller to facilitate channel selection. Unfortunately, in most of the existing presence applications, there is no distinction between a user and the devices or services associated with him. Hence, to determine user presence status any device or service must be subscribed and evaluated independently. As a subscriber has to know all contact addresses of the entities to subscribe, incomplete and inconsistent information may be the consequence of this architecture. For a presence system that supports multiple channels by design, a well-defined mapping from user identities to owned devices and services can be given. A subscriber then just subscribes to a specific user's presence status to get notified about the status of every contact point—or *channel*—exposed by the remote user. The notification service thus distributes a complete view on a user's communication facilities,

[13] Usually this additional negotiation step will be expressed by both peers as local policy in their corresponding configuration files. This way, no additional user interaction would be necessary.

probably annotated with additional information on channel priority, accuracy of given data, or free-form notes to explain the respective status value (e.g. longer absence during holiday could be commented with a note indicating the return date).

Having the subscribed entity assemble the published data (in contrast to the subscriber addressing every channel explicitly) has the additional advantage of sender-generated *views* on the status information. The sender of presence notifications then may decide independently for each subscriber what the notification should look like and how much information to reveal. The aggregation process also can take into consideration individual subscribers' identities, e.g. to create a high-level public view on the status information and a detailed private view containing sensitive information about current activities, or additional channels e.g. for the private cellular phone.

3.1.3 Summary

In the previous sections, we have discussed the usability trade-offs we found during our analysis of literature about existing groupware systems. In our opinion, these trade-offs have to be addressed when implementing a presence-based application. Although the functionality of a presence service is orthogonal to application design, it seems appropriate to visualize how decisions made when designing the technical infrastructure of the presence service will affect the design options available for a presence application. For example, most of the instant messaging applications found in the Internet allow for a single presence tuple to be published. Although the subscription usually is directed to a URI uniquely identifying a human user, the status attribute corresponds to a device or a communication channel, respectively.

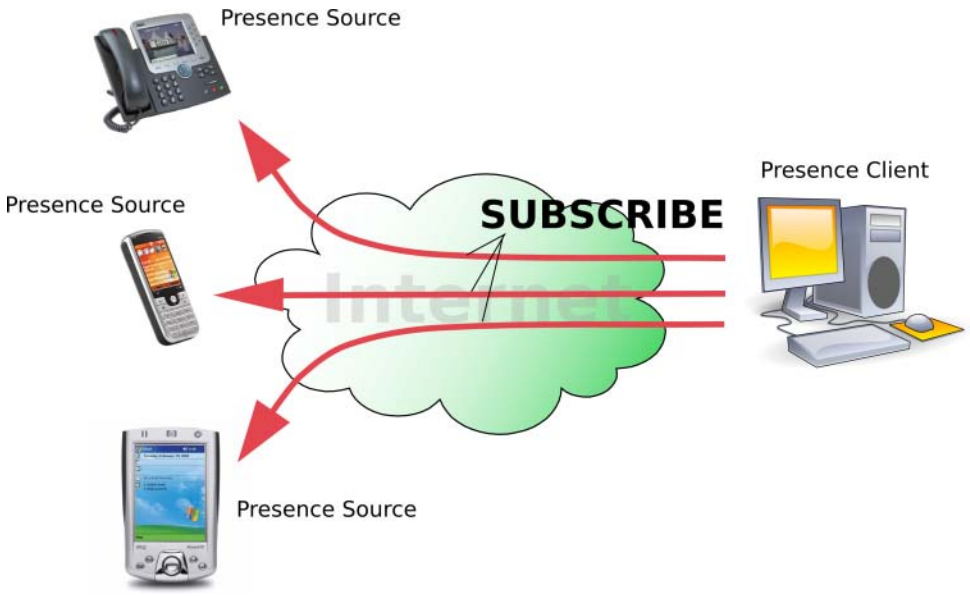

Figure 3.2: A presence client subscribing to multiple devices

Then, two things can happen when a user has more devices to be reached at: the presence status of all devices are aggregated and published in a single tuple, or a watcher has to subscribe the devices instead of the user (see Figure 3.2). In a presence aggregation service, this

is handled by an intermediary that translates a user-specific subscription into device-specific subscriptions and collects the presence attributes of incoming notifications into a large presence document that is forwarded to the watcher. As a result, the watcher's application is faced with multiple presence tuples corresponding to a single user (as this is the entity the human user of the presence service is interested in). The way how presence tuples are combined must depend on the personal preferences of the publishing user and his devices. The process for presence aggregation thus has to be designed on the basis of scenarios that reflect the anticipated use of the service by specific user groups. In this section, we have summarized the results from the discussion on usability and communication patterns and synthesized the following guidelines for application design:

- *Define a standard vocabulary with a flexible extension mechanism.*

  A fixed set of language elements with clear semantics simplifies computer-supported communication control with respect to the participants' preferences and capabilities and enables automatic inference of status values as explained in the preceding section. The language should be expressive enough to provide detailed status information for every communication channel offered by the person who is publishing her presence data. Possible status values might be `away`, `busy`, `in call`, among others. A specific vocabulary to describe a user's presence status in certain situations has been standardized by the IETF as [RFC4480] (cf. Section 4.2.2.2).

  In addition to the detailed descriptions of every channel, an aggregated value indicating the availability and willingness of the user to take part in a communication relationship should be contained in his presence status record.

  To give a more detailed description of a communication channels' status, a mechanism should be provided to extend presence tuples with additional attributes. To enable fast conformity checks, use of a type system is preferred, providing at least the basic types boolean, string and numbers with a fixed precision. Complex types are optional and can be realized by combination of components that have a basic type. To avoid confusion, the type system should build on existing standards.

- *Provide support for uncertain status values.*

  As mentioned before, automatic detection of the user's workload yields a numeric value indicating the probability of being available for communication instead of a symbolic value from a discrete vocabulary. Obtaining symbolic values can be done easily using fuzzy logic. However, as both, aggregation functions and end-user applications might benefit from the more of details known about the described entity, the numeric parameter could be included in the published status record as well.

  As other attributes of a presence status description probably are also determined by some sort of heuristics (e.g. estimations based on sensor-data), the presence language should provide a generic mechanism to annotate these values with a numeric indication of their accuracy or probability, respectively. Thus, the set of status attributes can be evaluated in combination, with single values being more significant than others. Local policies of an interpreting application (e.g. for aggregation of multiple sources of presence data) could define additional thresholds for specific presence attributes to be disregarded in further calculations. Together with the extensible type model, the attribution of values

with probability measures gives a strong mechanism for uncertain reasoning on arbitrary data sets as shown in Section 3.1.1.3.

- *Use a consistent presence data model.*

  In existing awareness applications and presence information formats, the semantics of the published data is not defined explicitly. In these systems, users can subscribe to contact addresses of other users' communication tools instead of their individual presence status. However, as the description of existing groupware systems in Section 2.4 already shows, the major goal of user presence is to facilitate interpersonal communication. Thus, the subscription of a specific communication service is reasonable but not sufficient. Subscriptions therefore should be user-based, i.e. status records represent the availability of a user in conjunction with a detailed description of the channels offered for communication.

  As this description provides the recipient of status notifications with a full overview of facilities to get into contact with the publishing individual, sorted by priority or personal preference, the number of successful communication attempts—both technical and social—will increase. Moreover, aggregation services can act upon the explicit channel set, e.g. to create watcher-specific views or to combine with the output from multiple presence sources.

- *Let users keep control of their communication relationships.*

  Technical limitations especially for distributed applications in wireless networks can heavily impact user-acceptance if spurious outages occur or throughput goes down without any obvious reason. The situation gets worse if negotiation of communication channels is fully automated and cannot be modified to fit a hostile environment. Users therefore should have full control over the channels they select for a conversation. This also includes the automatic presentation of details about channel sets from distant users to be able to choose appropriate media for interpersonal communication.

- *Generate alternate views specific to pre-defined subscriber groups.*

  To a certain degree, the quality of presence data depends on the disclosure of data about current or future activities. As some of these data might be sensitive and therefore to be seen only by a pre-defined group of well-known subscribers but not by the rest of the world, different views must be generated from the original data, one for each group. In conjunction with identity management and secure transmission, every subscriber can be provided with a custom view on the publishing entity's presence status without revealing too much sensitive information. The presence service might arrange for trusted intermediaries for subscription-handling and view generation according to a pre-loaded user-specific configuration.

- *Define an expression language for flexible filtering and throttling of notifications.*

  To avoid overloading of watchers with low-level presence data, technical measures should be provided to set an upper limit on the amount of data sent to such endpoints. The easiest way to achieve this is to control the rate of outgoing notifications depending on their size and frequency. Besides that, both, sender and receiver might allow for installation of boolean predicates to decide whether or not a notification should be processed at all.

Both approaches need an expression language to specify constraints on notifications. An outgoing message that does not fulfill a given constraint is silently dropped. Thus, a watcher is in control of what kind of messages are requested from a presence server. To query presence attributes as well as system-specific parameters provided by the runtime-environment, the specification language must support boolean expressions as well as arithmetic operations and functions that operate on character strings.

- *Allow for partial status updates.*

  To save bandwidth and device storage, the presence service should allow for status update notifications to contain only the differences between two states. Therefore, if only few channels have changed the presence system does not need to distribute the respective user's complete status description. Instead, a specific notation is used to indicate which status attributes are being updated.

Note that this list of guidelines is not meant to be exhaustive nor mandatory for designers of a presence service. Its sole purpose is to summarize the preceding discussion on design issues and present its synthesis in a form that can be referenced easily. As will be shown in the remainder of this document, numerous issues will arise when building a real-world presence aggregation service. As anticipated, the most pressing questions occur for assuring a secure and scalable service. Both aspects will be discussed in the following section, where these guidelines are refined into a more specific set of functional requirements for an Internet-scale presence aggregation service.

3.2 Functional Requirements

Following the suggestions from the previous section and the findings of the research project PASST [OKB+04, BOK05], we now describe the functional requirements for a large-scale presence aggregation service. The goal of this section is to discuss the options to implement the previously requested functions under the aspects extensibility, scalability, and security. Ease of use and maintenance cost play a less important role though these factors must not be ignored.

The main part of this section will be an informal description of a service architecture for wide-area distribution of presence data according to habits and preferences of the publishing individuals. The system should facilitate the syntactic and semantic aggregation of sensor data that describes single aspects of a user's presence status to enhance interpersonal communication as previously pointed out. Second, the aggregated data must be published to subscribed watchers with respect to customized access permissions. The goal is to enable what is called *social presence* in literature, i.e. distant users being aware of another. Since manual updates of presence status have proven to be inadequate, a sensor-based approach for automatic inference of current activity is recommended. To combine low-level sensor data with abstract presence information, a highly customizable, distributed presence aggregation mechanism is necessary. The functional requirements discussed in this section therefore address the following aspects of presence information management:

- *Distribution of local presence status*: Exchange of fine-grained information on the current activities of a specific user in a local environment such as the user's desk area. Personal

devices and services within a single trust domain publish low-level data to describe their current status. This data may be synthesized and made available externally by a dedicated aggregation engine.

- *Presence status aggregation*: The transformation of low-level presence data from multiple sources into a more abstract view. The resulting status description should list possible contact addresses of a user together with an abstract classification of the media types offered for that contact. In particular, the vocabulary used for this purpose should differentiate between audio, video and text communication as well as additional services such as application sharing.

- *Publication of presence information*: Subscription management and notification of watchers if the presence status they have subscribed for has changed.

- *Integration with a multimedia communication service*: Providing an application interface to facilitate initiation of media sessions based on presence information.

- *Configuration interfaces for user-specific aggregation rules*: Usability of interfaces for user control of the aggregation process.

As the main aspect of a flexible wide-area presence service is the aggregation of presence information from multiple sources, the following requirements assume the existence of a generic presence model as a common content format for the transport protocols used to disseminate the presence information over the Internet. This presence model is required to be independent of specific deployment scenarios (such as the limitation to enterprise networks or the assumption of a specific network topology as is common in cellular networks) and of the nature of information sources. Instead, the model should support the management and distribution of presence information from various data sources, including low-level sensor data as well as abstract data such as SIP registrations or availability information from instant messaging applications. To ensure utmost interoperability with existing and future applications, the presence service should use standardized technologies wherever possible, especially for distribution of presence data.

Figure 3.3 again shows the overall architecture for the presence aggregation system we have presented in Chapter 1. The central element of this architecture is the presence server which is responsible for aggregation and re-distribution of the abstract presence information that has been generated from individual presence zones (shown on the left). Every presence zone constitutes a *personal presence environment* with a single *personal presence server* (PPS) that is responsible for aggregation of low-level presence data from personal devices or distinct sensors.

Another important aspect of this architecture is the public presence infrastructure. It is responsible for distribution of presence information to user agents that indicated their interest in the particular user's (presence) status. Depending on the exact topology, multiple presence protocols could be used, e.g. SIP and XMPP (cf. Section 3.3). Presence servers and protocol gateways can provide additional functions taking advantage of the presence status information. Examples for these functions are routing of incoming calls (e.g. to implement a follow-me service) or filtering of notifications based on the watcher's identity.

User agents are required to interpret the received presence information and render it appropriately to the watcher. Though most existing clients respect the usability guidelines given in

Section 3.1, the content format for representing status information used by the underlying protocols is too limited to achieve the desired effect. Therefore, if new data fields are added to the presence information format clients must be adjusted accordingly to benefit from the changes. In particular, the probability value we have postulated earlier means a valuable information to the watcher and thus should be displayed together with the corresponding status description. As a consequence, any extension to the presence format may require adjustments of user agents as well.

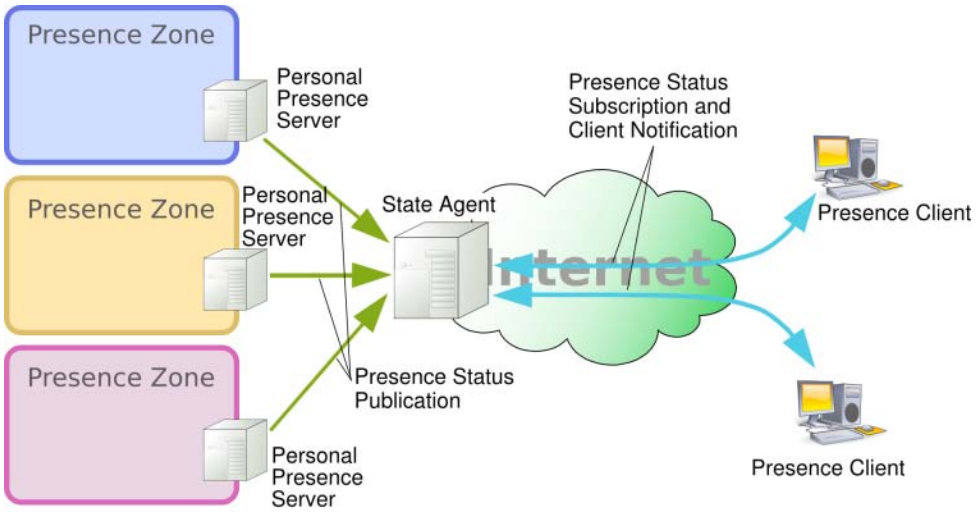

Figure 3.3: Status aggregation for multiple presence zones

Note that the service architecture depicted in Figure 3.3 shows a small part of the entire system specific to a single user. A scalable protocol for dissemination of presence information requires a distributed server architecture with each server being responsible for a small fraction of all (presence) users in the Internet, typically associated with a single administrative domain. Users therefore must have globally unique names from an infinite name space (at least conceptually). Every user in turn may have multiple personal presence environments, e.g. an office environment and a home environment. Within these environments, user-specific presence information is collected, aggregated and then published to interested watchers. Sensors in the local environment therefore are not globally visible and need only local addresses. The presence aggregation process then is responsible for generating a consistent view on the user's presence status before this information is forwarded.

The *public presence server* that handles subscriptions from external watchers, must check if these watchers are permitted to receive the information they have requested. For example, a user might want family members and co-workers to receive his presence information with a different level of detail. To be able to do so, different authorization classes can be specified for the group of *family members* an the group of *co-workers*. The presence service must implement an access control concept and should provide means of distributing different presence information to different watchers. For certain watcher classes, access to presence information may also be blocked completely.

Based on this architecture, the functional requirements have been clustered in five categories that are discussed in the following subsections. First, the coordination of local sources of pres-

ence information and the data distribution within the local environment are analyzed. After that, we take a look on the data format used to describe the presence status of an entity, and discuss how presence documents in this format have to be processed by an aggregation engine. The fourth part of our requirements list is about the rule-based language for controlling the aggregation process. Finally, the provisioning of scripts to the aggregation engine is discussed.

3.2.1 Local Presence Environment

As seen before, the local presence environment must allow for easy integration of presence sources. A component-based approach together with a flexible addressing-scheme is envisioned to ensure interoperability between distinct modules, especially if provided by different vendors. As sensors may be shared between several users, it must be possible to detect the presence or absence of sensors in the local presence environment and determine their grade of utilization by the corresponding user. For example, a user could use a conference phone in a meeting room shared with several other users. The information that a conference call has been initiated from this phone now can be used as activity indication for any user in that particular room. The phone therefore should appear as a dynamic presence source for any user while in the room.

The local environment therefore must provide light-weight communication between components, role-specific addressing and self-description of components as well as an authorization framework with multiple trust domains. In particular, the following requirements have to be addressed:

- *Ad-hoc communication*

  Components within a trust domain should be able to communicate with each other without explicit configuration of endpoint addresses or registration at central entities. The communication infrastructure must detect new components and report disappeared components during runtime.

  Presence sources such as sensors and external protocol engines (e.g. SIP user agents) and any publishing engine such as the personal presence server will be implemented as one or more components using this platform for internal communication within the local presence environment. The dynamic nature of this environment and the possibly high frequency of status messages requires a light-weight communication infrastructure with low management overhead.

- *Self-description of presence sources*

  Presence sources must provide an abstract description of their type and capabilities on request. The local presence server hence can detect presence sources in the local environment and determine user-specific processing rules depending on a source's actual type. This allows a presence aggregator to apply processing rules without the need for explicit configuration for every presence source, especially when new sensors have been added.

- *Application-specific command sets for local communication*

  An extensible set of application-specific commands for controlling presence sources and for conveying presence information from sources to aggregators should be provided to facilitate creation of new components. The content format being used to express sensor

data must support the degradation of samples, i.e. the sampled value is annotated with a decay function that specifies its decreasing exactness over time.

To avoid duplication of effort and ensure interoperability of presence sources, a standards-compliant representation of presence values based on the IETF *presence information data format* (PIDF) [RFC3863] must be used.

- *Robust and secure communication*

  To protect sensitive data, the local communication infrastructure must provide authentication of components and encrypted communication for groups of components owned by a single user. To enhance robustness of the local communication and protect the entire system from outages induced by crashed components, components should send keep-alive messages with a unique identification on a regular basis. The frequency of these soft-state registrations must be adjusted automatically according to the number of active components to avoid congestion of the communication channel with management overhead.

3.2.2 A Data Format for Presence Aggregation

The presence data format will be used to represent presence information in the different stages of the presence aggregation and distribution process. One fundamental format and processing model will be used for all stages of the presence aggregation and distribution process. As a result, the presence aggregation process can be viewed as an iterative process where each aggregation step receives input from one or more presence sources and aggregates this information in order to obtain a semantically richer (or more precise) presence information about a given user.

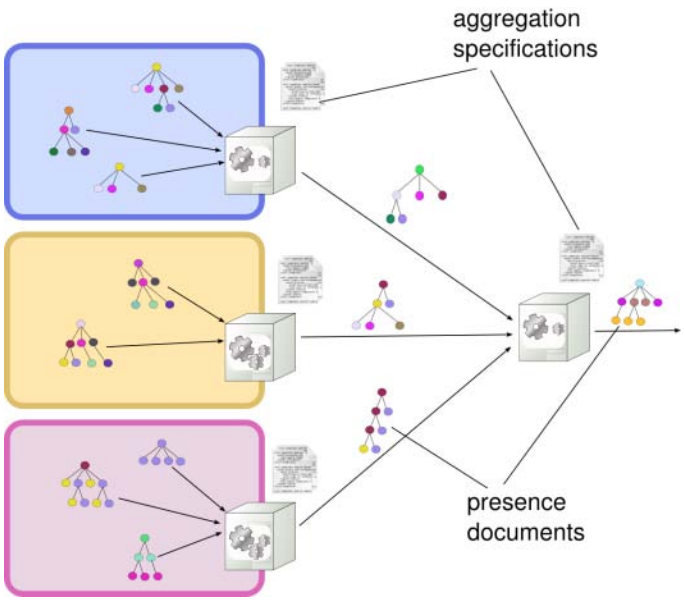

Figure 3.4: Using a common presence data format for multi-step aggregation

Figure 3.4 depicts the fundamental processing model: a presence aggregator receives presence information from multiple sources that is represented in a common format. Input and the output of the aggregation process use the same data format hence enabling multiple aggregation steps to be performed iteratively. The aggregation process is controlled either by a set of default rules or a script provided by the user whose presence information is being processed (see Section 3.2.3). The data format therefore must provide sufficient type information and processing rules to enable automatic processing based on the default rule set.

A presence document describes one or more media channels offered by the user. Each channel description consists of various attributes indicating its capabilities and the current status. Static attributes can be used to specify a unique name for each channel and the media types the channel is capable of.

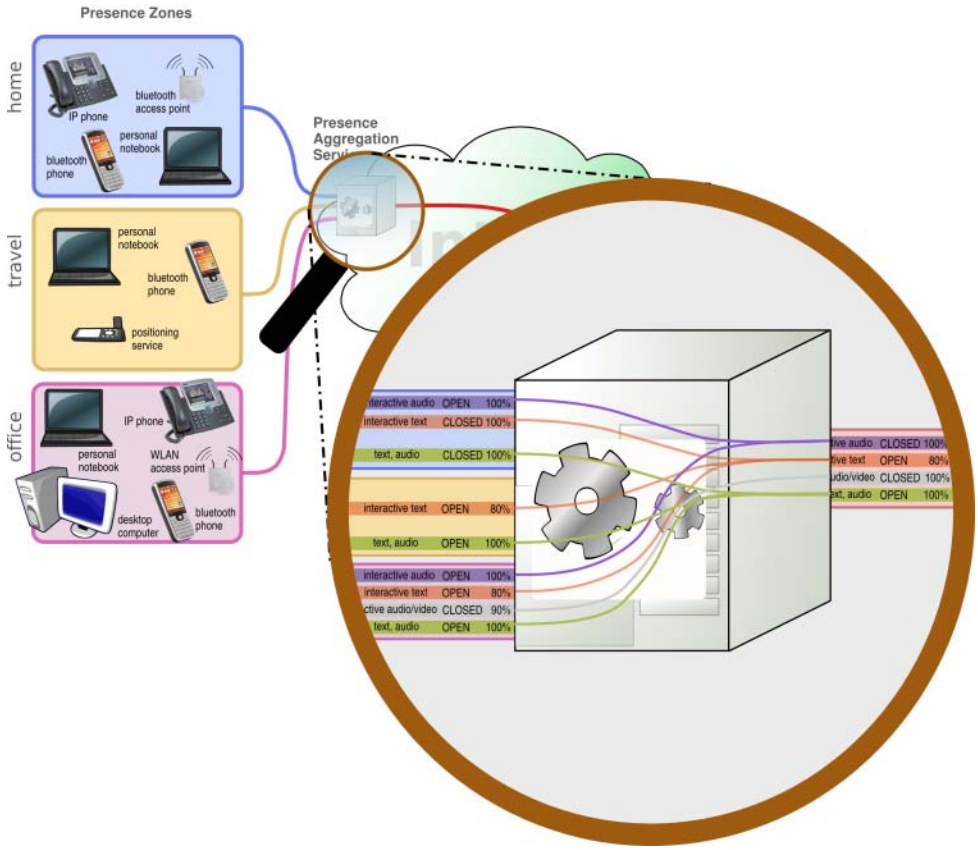

Figure 3.5: Aggregating multiple channel descriptions

The channel concept is an abstraction of device-specific approaches offered by several presence protocols today. The presence vocabulary therefore must provide means for specifying the channel type (e.g. “audio”, “voice”, “video”, or “text”) and type-specific facets of the presence status (e.g. a voice channel may be marked as `in call` if it represents a telephone).

The aggregation of presence documents comprising multiple channels is depicted in Figure 3.5. Here, the aggregation engine generates a presence status description from an input document comprising several channel descriptions. The output of the aggregation process may

contain any number of channel descriptions, including zero (in this case, no presence status notification will be sent).

Access of subscribed watchers to presence information may be restricted on a per-channel basis. The presence document format must support the concept of *authorization classes* a channel may be associated with. Pre-defined authorization classes allow for definition of user groups that are granted access to specific parts of a presence document. New classes may be defined either in a presence document or within an aggregation specification.

In summary, the presence data format must provide enough information to describe the status of specific devices represented as abstract communication channels. Additional attributes are required to control the aggregation process, especially to denote the aging of concrete presence values. To avoid sensitive data to be disclosed to unauthorized watchers, the concept of different authorization classes is to be supported as well.

3.2.3 The Presence Aggregation Process

As depicted in Figure 3.4, the presence aggregation process takes input from one or multiple presence sources and (optional) processing instructions that govern the processing step. The processing instructions are represented in a custom *Presence Aggregation Language* (PAL).

In general, the aggregation process is a specific form of sensor fusion where multiple input parameters are transformed into a new status description. The way the fusion takes place is dependent on the basic data types and processing rules. These rules are defined implicitly by the aggregation engine or stated explicitly in a user-specific aggregation specification, called a *PAL script*.

As mentioned in the previous section, the channel abstraction is a fundamental concept for presence aggregation. Channels are described as sets of attributes each of which has a specific type, value, and optional information on the degradation of this value's exactness over time. This leverages the notion that parts of the information contained in a presence status description may become less relevant with an increasing age. Each information is thus associated with a timestamp (when the information has been sampled or generated) and with a decay function that specifies how to decrease the relevance with respect to the age. Different classes of decay functions should be supported to take into consideration various models of degradation:

- no decay (constant),
- linear decay, and
- logarithmic decay.

For the classes *linear decay* and *logarithmic decay*, additional parameters such as factors and logarithmic base values can be specified. In addition, thresholds can be specified that are used to determine when the information has expired and should no longer be considered.

3.2.4 An Aggregation Language for Presence Information

Although information system design has been a popular research topic for several decades and its fundamental concepts are well understood, real-world systems typically have specific constraints that must be taken into account when these technologies are applied to that particular

context. This section covers the major aspects of formal languages that describe the conversion of collections of structured information items between different levels of abstraction. In particular, the design of our specification language for algorithmic aggregation of presence information documents is discussed with respect to several generic building blocks used for similar tasks.

Before discussing detailed aspects of the presence aggregation language, it is important to recall the non-goals of this language, i.e. what it is not supposed to be: First, no formal verification of aggregation results is required. Since presence information documents may be very dynamic, with their vocabulary not necessarily known a-priori, correctness proofs are a tedious task and cannot be performed off-line. Hence, formal specification techniques like *higher order logic* (HOL) or *CSP* do not seem appropriate solutions. Second, users should not be forced to understand the concrete syntax of presence information documents as would be necessary for document transformations on the syntactic basis (e.g. using XML transformations [XSLT]). Instead, the aggregation language shall define an abstract data model that facilitates aggregation specifications to use terms of the application context.

The language to be designed instead should not require specific knowledge in software development, though it must cover a broad range of application scenarios including very specialized setups where event-sources are spread over several administrative domains, have different notification rates and abstraction levels. Moreover, a single user may have various aggregation specifications that depend on each other placed at different presence servers. To meet the security requirements mentioned in Section 3.4, distributed aggregation specification must carry enough information to authenticate watchers and check their permissions to access specific information items.

Some valuable advices regarding the usability of specification languages are given by Mackay in [Mac90] who examined the user interface of *Information Lens* [MGL+92], a system aimed on improvement of organizational knowledge by processing personal e-mail folders, similar to procmail[14]. According to Mackay, configurable software must match the following criteria (cf. [Mac90]):

- *Reflectiveness*

  Adaptive software must be reflective as it has to give users feedback on the effect of modifications they have made. As software customization typically is an iterative process, the system has to provide mechanisms for users to become aware of their personal usage patterns and to understand the determinism between changed input parameters and produced results.

  This requirement is related closely to *robustness* of configuration files: Local changes of a software system or its processing rules should have only local effects that can be attributed to a particular change. If this behavior is not given, adaptations cannot be done in an iterative fashion and thus validation of changes is hardly possible. [Mac90]

- *Usability*

  Complex software systems must be able to run out the box, without non-trivial configuration steps being required before first startup. As most systems' flexibility is made primarily of their large number of customization options, a trade-off between usability

[14] See <http://www.procmail.org>.

and flexibility exists. In general, this is solved by providing a set of default rules good enough to enable proper operation even if not modified.

Default rule-sets should be modular, self-contained and independent. Otherwise, the robustness criterion would be missed. For example, the XML document transformation language *XSLT* defined in [XSLT] provides a set of default rules any of which can be replaced by a customized rule without interfering with the other default rules (given that the customization itself does not prohibit their invocation).

The more complex a system the more interdependencies of rules exist. A popular example is the mail server sendmail[15] which is equipped with a large number of processing rules. The high quality of these rules manifests in the fact that numerous sendmail servers run for years without change even of a single rule. The drawback of this excellent work is the interdependence of rules: Many operators do not change the default configuration as they are afraid of unforeseen behavior of their mail server.

- *Cooperative*

  In [Mac90] the author documents the process of sharing customized configuration files in large organizations. According to her findings, software customization is done in large parts by personnel with high skills in software development. This is true especially for new programs being deployed within that organization.

  An important observation is that after having customized the software according to their own needs, users often publish the configuration files in order to share their knowledge with others. Therefore, complex software systems should support this process of information distribution, e.g. using modular configuration files in a standardized format. Functions that can be generalized to solve not only a single problem but a class of similar problems could be collected in libraries that can be referenced in a configuration file.

  A prominent example for a highly configurable software system with a modular customization mechanism is the editor Emacs[16]. It comes with a considerable default configuration that can be adapted to anyone's needs using a variant of the *Lisp* programming language. The main configuration file can be split into multiple files using the inclusion mechanism of Emacs Lisp. Override mechanisms allow for adaptation of existing modules. Moreover, a plethora of libraries with useful functions is published under the terms of a free software license. Thus, users who are not familiar with writing Lisp code can easily extend their editor by simply including those libraries.

Although Mackay's paper was written in 1990 and is documenting findings in an environment that is quite different from today's information society, the findings cited above still seem to be true. Strong indicators are the given examples and—probably more important—the existence of the Open-Source Software community that produces high-quality software mostly on a volunteer-basis.

Taking into account the results of the cited study, the presence aggregation language should be powerful enough to fit the needs of high-skilled users, i.e. useful features need not be sacrificed only for the sake of simplicity. To facilitate use of the language even by untrained users,

[15] See <http://www.sendmail.org>.

[16] Available at <http://www.gnu.org/software/emacs/>.

an appropriate interface that gives a simplified view (and probably less functionality) is to be provided. This could be achieved by a graphical user interface together with a library of abstract building blocks for common functions. More elaborate functions that have been created manually are hidden as library objects as well. As an example for a graphical interface used as abstraction layer for a more complex transformation specification can be found in [Hüt01] and [Ber01].

Given these requirements, we envision the aggregation language to re-use primitives from existing programming languages in combination with the specific concepts of presence aggregation such as channels and their associated attributes. Although the language is not required to be a general-purpose programming language or to be Turing-complete, some elements of typical script languages could be useful to simplify specification development and enhance the readability of PAL scripts. Examples for these additional features are object notation for complex data types, functional abstraction, and regular expressions. Beyond that, the language is required to provide the following features:

- *Boolean algebra*: PAL must allow for describing boolean expressions and provide predicates that allow for evaluating attribute values and use the result in boolean expressions.
- *Conditional statements*: Conditional statements allow for selecting "branches" of channel declarations depending on the value of boolean expressions and thus are important aspect of PAL.
- *Mathematical functions*: Mathematical functions are provided for constructing mathematical expressions that compute values, e.g., taking attribute values of input channels into account.
- *Attribute selectors*: Attributes given in a presence document must be accessible from within a PAL script. Regular expressions may be offered to specify attribute selectors without fully qualifying the requested attributes.
- *Container types*: Multiple presence values can be stored in containers together with operations on these containers to access their elements and to apply functions on a collection of attributes.
- *History support*: Presence information for a specific channel may not only depend on current status values but also on historic data. Therefore, aggregation functions should be allowed to calculate the current status based on time series of attribute values. For example, the indication of activity may be derived from idle time sensors using the average durations of inactivity.

The invocation of PAL scripts is controlled by the aggregation engine which is responsible for providing correct input parameters and handling the scripts' output values. Several events may trigger script execution, e.g. the receipt of a status update notification for the corresponding presentity or a system-generated event such as the expiry of user-defined timers. Watcher-based filtering of notifications and event throttling may be controlled by PAL scripts as well.

Finally, the aggregation process should be backwards-compatible with existing clients such as older versions of *Microsoft Windows Messenger* that use vendor-specific variants of the PIDF content format. To benefit from the wide deployment of these applications, the aggregation engine should be able to convert between the most common presence formats.

3.2.5 User-Specific Configuration

Once a user has created a PAL script that defines rules for aggregation of presence information and the publication of the aggregation result to subscribed watchers, this script must be installed and activated on the user's presence server. From the existing protocols for configuration management, we advocate for an interactive Web interface that provides an upload function for script files as well as management features to inspect, modify, re-order or delete existing scripts.

As the presence server can be located anywhere in the public Internet, the HTTP connection must be secured to avoid eavesdropping and tampering with a presentity's configuration. Encryption is required because the aggregation rules may disclose e.g. the existence of certain presence sources, or too restrictive publication policies could compromise the principal. In addition, the modification of aggregation rules is permitted only to authorized users, as rogue aggregation scripts could reveal sensitive information published by certain presence sources. Uploads therefore must be authenticated, and the integrity of provided scripts is checked.

3.3 Distribution of Presence Information

The discussion of social and technical issues of systems to support interpersonal communication in the previous sections of this chapter has led to a number of guidelines and functional requirements for designing a presence service that addresses these issues better than existing groupware systems do. A technical prerequisite for this is an infrastructure for distribution of presence information and initiation of real-time multimedia communication. In this section, we discuss various technologies that can be used for this purpose, with a specific focus on presence information delivery. We begin with a brief introduction to protocols and architectures for Internet-scale dissemination of presence documents in Section 3.3.1. The section refers to the World Wide Web as a reference architecture for a scalable wide-area information system. After that, we discuss the standardized *Session Initiation Protocol* (SIP) in Section 3.3.2 as a flexible protocol that enables not only push-based event notification but also the setup and management of real-time multimedia communication sessions. In Section 3.3.3, we evaluate our choice against various alternative approaches that are specialized on the distribution of presence information.

3.3.1 Internet-scale Event Notification

An important aspect of a global presence service is the scalable distribution of published presence data to the subscribed watchers. While existing groupware systems typically aim at centrally managed organizations with a coordinated network infrastructure, global presence systems must interface with a large number of different network architectures, operating systems, and administrative domains. Hence Rifkin and Khare identify *system interoperability* as one of the most critical factors for Internet-scale applications, as these applications must work across trust boundaries and bridge between different semantical levels of information ("*ontology boundaries*") using a common application vocabulary. [RiKh98b]

In this section, we discuss the basic design options that are available to construct an Internet-scale presence service. The requirements we have identified in the previous section impose a number of constraints on the overall system architecture that fit into the concept of *Represen-*

tational State Transfer (REST) defined by Fielding and Taylor [FiTa00a, Fie00b, FiTa02] as a modern interpretation of the World Wide Web architecture. In particular, [FiTa02] imposes the following requirements on a global information system:

- Internet-wide communication

  An open system must allow for "anarchic scalability" [FiTa02] caused by the lack of a centralized instance that controls the evolution. In addition, packet loss, different latency and delay times in networks prohibit the establishment of consistent state between all nodes with a simple request/response protocol like HTTP.

  If messages traverse multiple trust boundaries, security may deviate as well in the absence of end-to-end encryption and authentication. The lack of a global key infrastructure also prohibits establishment of trust relationships with arbitrary peers in an open system, i.e. a trade-off exists between scalability and security.

- Independent development

  As stated above, there is no centralized instance controlling the evolution of large, open systems like the Internet. As a consequence, development of new applications is hardly coordinated. To retain interoperability, changes to the system are always gradually, causing improvements to be backwards compatible in most cases.

Appreciating its scalability and lack of centralized control, various presence applications use the *Hypertext Transfer Protocol* (HTTP) or extensions thereof to convey status descriptions. Examples for HTTP-based presence services include the server-based NESSIE awareness environment [Pri99] as well as highly interactive presence applications such as *meebo*[17] or *KoolIM*[18]. Although using different techniques for generating updates of the displayed presence status, all of these applications have in common the active fetching of presence information.[19]

These examples already show that HTTP has always been a reasonable transport mechanism for presence documents from the early days of the World Wide Web. When Rifkin and Khare published their survey [RiKh98b] on event notification systems in 1998, they have set on economic network effects to leverage a single protocol for event notifications, as seen in the case of HTTP as enabler of the *World Wide Web*. Hence, they have voted for a text-based protocol that facilitates asynchronous messaging using a syntax similar to HTTP.

The protocol envisioned by Rifkin and Khare should have provided functions necessary for advertisement and subscription of event types, notification management, and policies. Clients in this model poll a local proxy server for events that are announced in a global directory formed by the distributed proxy infrastructure. Figure 3.6 illustrates a possible architecture for this client-initiated notification model.

The proxies in this model are organized like a content network (CN), i.e. an overlay network on the application layer according to the terminology of [RFC3466]. Acting as *surrogates*, proxies receive requests from local clients that are interested in the status of other participating entities. When a request cannot be satisfied by the surrogate it is being forwarded to the *origin*

[17] See <http://meebo.com>.

[18] See <http://koolim.com>.

[19] Note that the NESSIE system also provides a push-based protocol that enables active notification of status updates by the NESSIE server.

server for the requested resource. As presence status information documents by definition are highly dynamic, any request will be forwarded unless the surrogate itself is the origin server for the requested resource or it has cached a recent copy of that document. In Figure 3.6, the client protocol provides two operations:

- `FETCH`

  The `FETCH` method can be used to retrieve information about the specified resource's current presence status. The receiving proxy might return a cached copy of the requested presence information data if still valid. Otherwise, it forwards the request to the origin server that is responsible for that particular resource. When processing a `FETCH` request the proxy may store routing information to contact the origin server directly for subsequent requests. If no such information exists, the overlay network's intrinsic forwarding mechanism is being used.

  Once the presence information is known, it is returned in the body part of the response. A negative response with an empty body is sent to the client if the requested resource could not be accessed (e.g. because it is not available or the client is not authorized to access that data) or the operation has timed out.

- `PUBLISH`

  To update its current presence status the `PUBLISH` request is used. The request body contains an appropriate representation—either partial or complete—of the presence status information. `PUBLISH` requests always have to be acknowledged by the origin server to avoid inconsistent status information caused by requests that were lost or have been reordered.

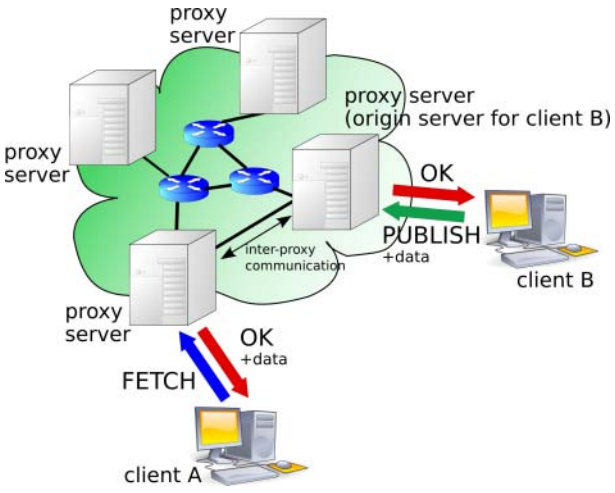

Figure 3.6: Distribution architecture for pull-based event notification

To minimize protocol interaction for subsequent messages, the protocol outlined here may instantiate an implicit subscription after a `FETCH` request has arrived at a proxy and the authorization requirements are met. To make this subscription explicit, an additional protocol

method, say SUBSCRIBE, could be used as well. This kind of request then indicates interest in the presence status of the specified resource. The proxy may forward the request to other servers as appropriate, e.g. to establish a subscription at the corresponding origin server to get notified of status changes. The response sent to the client indicates whether or not the subscription was honored by the content network. This is useful especially when denial-of-service attacks are an issue (see Section 3.4.2) as FETCH requests without previous subscription could be declined to speed up operation.

Following the REST model, the methods FETCH and PUBLISH could be mapped to the HTTP methods GET and POST to create an HTTP-based presence service. A client then must poll the presence server to fetch the latest status updates. As HTTP does not provide a push-service for content distribution, the SUBSCRIBE method would have the same semantics as FETCH.

As the polling of servers for status updates seems to be inadequate for an Internet-scale presence aggregation service, we have investigated alternative protocols that offer persistent subscriptions and push-based dissemination of status updates. A standardized presence distribution protocol that meets these requirements is the *Session Initiation Protocol* (SIP), discussed in the next section. In addition to the transport of presence documents according to the abstract definitions of FETCH and SUBSCRIBE, the protocol supports instant messaging and signaling of real-time multimedia sessions.

3.3.2 The Session Initiation Protocol (SIP)

The *Session Initiation Protocol* (SIP) has been developed primarily for setup and termination of interactive multimedia sessions over the Internet. Its core functionality defined in [RFC3261] not only provides a mechanism for creating and terminating media sessions, but also defines an application-layer routing mechanism as the basis for exchanging messages between SIP protocol entities. Scalability is achieved by soft-state registrations of SIP endpoints (*user agents*) and DNS-based resolution of SIP addresses (typically URIs with scheme sip) as well as transaction-based communication that ensures efficient forwarding of SIP messages.

As the protocol has been designed as a basis for a broad range of applications within the area of interpersonal real-time communication, it provides a generic mechanism for reliable transport of MIME content and related meta-data that can be used independently of the functions for session-control. In this regard, SIP is akin to the HTTP REST model (cf. Section 3.3.1) where the result of an operation depends on several parameters such as the request method, the request URI, possibly some additional meta-data (e.g. cache modifiers) and the current status of the requested resource. Unlike the World Wide Web architecture, SIP does not explicitly separate clients and servers. Any node in a SIP network therefore acts as a server and listens for incoming SIP requests. Following the World Wide Web terminology, SIP nodes that forward SIP requests according to some specific routing scheme are called *proxies*, while the term *user agent* (UA) denotes endpoints that process incoming requests according to their method's definition. A user agent is logically split into two parts, the *user agent client* (UAC) that initiates SIP transactions, and the *user agent server* (UAS) that processes SIP requests sent by a UAC to initiate a SIP transaction.

In this section, we give a brief overview of the basic SIP functionality when used as a protocol for event notification, and show how presence aggregation can be handled within a SIP network. We begin with a high-level introduction to the network architecture of SIP in

Section 3.3.2.1, followed by a description of the protocol's application-layer message routing mechanism in Section 3.3.2.2. Section 3.3.2.3 then shows the generic event notification mechanism for SIP that has been defined in [RFC3265] and its use for dissemination of presence information according to [RFC3856, RFC3859]. The specific task of presence aggregation is addressed in Section 3.3.2.4. A short summary concludes this section with an evaluation of SIP against the functional requirements we have stated in Section 3.2.

3.3.2.1 The Basic SIP Architecture

The basic architecture of a SIP network is illustrated in Figure 3.7. In this figure, the user agent on the left has registered with the SIP proxy for the domain "`example.org`". This registration creates a binding with the sending user agent, i.e. the abstract name "`bob@example.org`" is associated with its corresponding device address. When another user (here shown at the right) sends a message to `bob@example.org`, it will pass zero or more SIP proxies, and eventually arrive at the proxy where the target user is registered. The proxy then looks up the binding for the registered user and forwards the request to the associated device address, or returns an error response if no such binding exists.

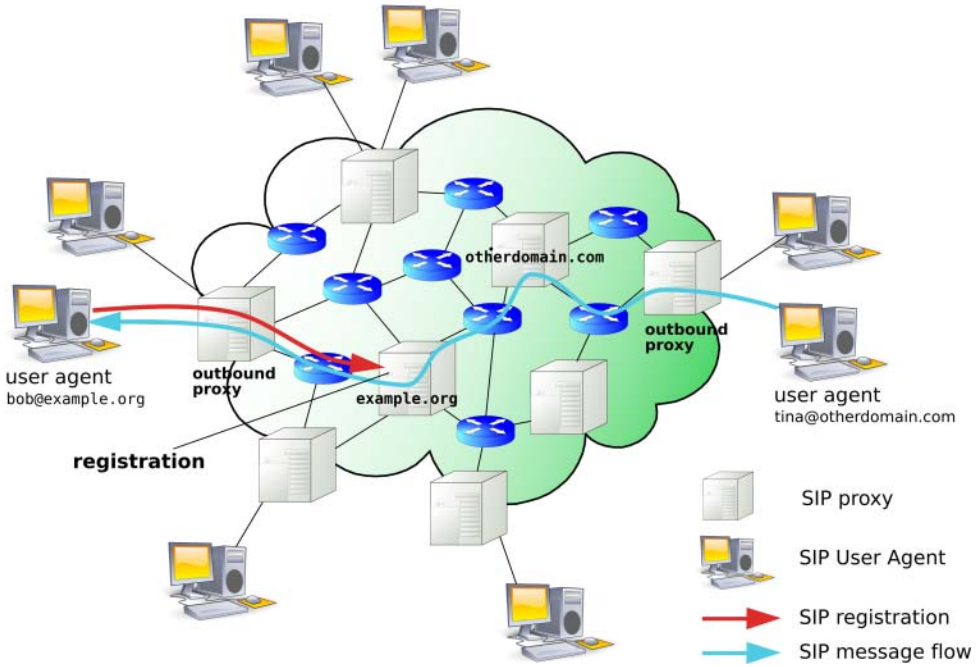

Figure 3.7: Basic SIP architecture

The arrows in this picture denote complete SIP transactions, i.e. operation-specific requests with exactly one final response completing the transaction. While being forwarded, the path of each request is recorded in a specific message header, `Via`. Once the receiving entity has generated a response, it is passed along the reversed `Via` path and thus traverses the same proxies as the corresponding request. The subsequent adding of `Via` headers on the request path is shown in Figure 3.8. On the return path, the topmost header is dropped as it arrives at the proxy denoted by this specific header.

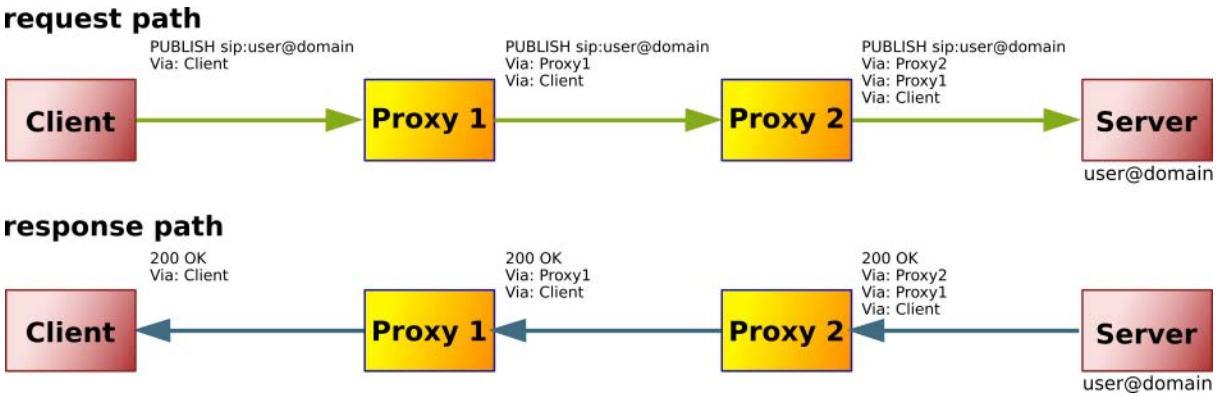

Figure 3.8: Adding `Via` headers in a SIP transaction

With a response traveling upstream along the path created by a forwarded request, a proxy knows when the transaction ends and hence can destroy any data it has stored locally to process this transaction. In particular, this is used to handle timeouts that may occur in a congested network or when user input is required to handle requests at their final destination. The resource allocation of a *transaction-stateful proxy* therefore only depends on the number of transactions to be handled in parallel. A state agent (i.e. a presence server that manages subscriptions on behalf of a user) in contrast must provide additional resources for storing subscription data and optionally for caching status descriptions. A subscription for a particular presence record hence creates a *dialog* between the state agent and the initiator of the subscription. Every status update notification then is sent within the context of this dialog, identified by specific mandatory parameters of the SIP request and their corresponding final response.

To keep track of the needed resources and to avoid stale subscriptions, each subscription has a limited life-time. If not refreshed in time, the state agent will remove the subscription from its internal storage. If necessary, subscriptions even may be preempted by the state agent to save resources.

3.3.2.2 Application-layer Message Routing

To determine the next hop where a SIP request must be routed to, user agent clients and proxies examine the *request URI* contained in the start line of this request. Usually, the request URI contains a request target of the form `user@domain`, where the domain part can be used to identify the SIP server that handles incoming messages for the administrative domain of the receiver. In our example, the receiver's administrative domain is `example.org`, and the request takes an additional hop through the proxy for the sender's administrative domain, `otherdomain.com`. Both peers of this message flow additionally use a dedicated *outbound proxy* to handle outgoing messages and thus do not have to care about the routing of messages through the SIP network.

To determine the actual address of the destination server, the sender only needs to resolve the domain label as described in [RFC3263]. The use of DNS `SRV` and `NAPTR` resource records leverages decentralization far beyond the distributed architecture of the World Wide Web. In particular, HTTP only allows for immediate name resolution (i.e. `A` or `AAAA` record lookups) while `SRV` resource records additionally carry the port address and an implicit protocol specifier

used to query the SRV record. For example, a zone file for the domain example.org could have entries shown in Example 3.3.1.

Example 3.3.1: A DNS SRV resource record for SIP over TCP

```
_sip._tcp    IN SRV  0 100 5060 proxy.example.org
proxy        IN A  134.102.218.199
```

Here, the TCP-based SIP service is provided by the server proxy on port 5060. As there is only one entry for this service, the additional numeric parameters specifying the priority and weight of this particular entry relative to other entries can be set to any numeric value. Given this, a lookup for _sip._tcp.example.org (e.g. using the UNIX command **host -t SRV _sip._tcp.example.org**) would yield the string proxy.example.org which in turn can be resolved to the IP address 134.102.218.199.

If a more flexible mapping of resource identifiers to transport addresses is necessary, the *Dynamic Delegation Discovery System* (DDDS, see [RFC3401] and following) can be used. To locate a SIP server, the destination domain of a request URI would be queried for a *NAPTR* DNS resource record [RFC3403] as shown in Example 3.3.2. The service descriptors "SIPS+D2T", "SIP+D2T", and "SIP+D2U" list the services offered for example.org, i.e. SIP over TCP and SIP over UDP as well as SIP over TLS.

Example 3.3.2: DNS NAPTR resource records for SIP domain resolution

```
example.org.
  IN NAPTR 50 50 "s" "SIPS+D2T" ""  _sips._tcp.somecompany.com.
  IN NAPTR 50 50 "s" "SIP+D2T"  ""  _sip._tcp.somecompany.com.
  IN NAPTR 50 50 "s" "SIP+D2U"  ""  _sip._udp.somecompany.com.
```

The example given here only contains terminal rules, i.e. no recursive NAPTR lookup is necessary. According to the procedures for locating SIP servers defined in [RFC3263], a SIP-specific NAPTR query yields a string to be used as input to a DNS SRV resource record lookup. Within the NAPTR entry this is indicated by the flag "s".

DNS-based routing of requests and the symmetric response-path created by insertion of Via headers, together with the concepts of transactions, comprise the *transaction-layer*, a minimal core of SIP-based communication. Scalability of the protocol is ensured by the decentralization of SIP servers across the global domain space. Based on this core protocol, the IETF has defined application layer protocols such as the event notification mechanism that is discussed in the following section.

3.3.2.3 SIP Event Notification

In the previous sections, we have summarized the functionality of the transaction-layer that is responsible for efficient message routing through the SIP network. As SIP messages can contain almost any type of content, the protocol has become popular especially as transport mechanism

for applications that enable interactive real-time communication. One of the applications that have been defined on top of the basic SIP protocol is the event notification service of [RFC3265] used for conveying presence information through the global Internet.

The SIP-based presence service works as depicted in Figure 3.9. The user on the right has passed a subscription request (using the SIP method `SUBSCRIBE`) for the presence status of `alice@example.net` to the global SIP network where it is processed by a *state agent* that handles subscriptions for that particular user. While this subscription is active, the client will be notified by the state agent via SIP `NOTIFY` whenever a presence-related event for Alice occurs. As depicted here, a status change can be induced by sending a `PUBLISH` request to the state agent that is responsible for the presence status of the resource `alice@example.net`.

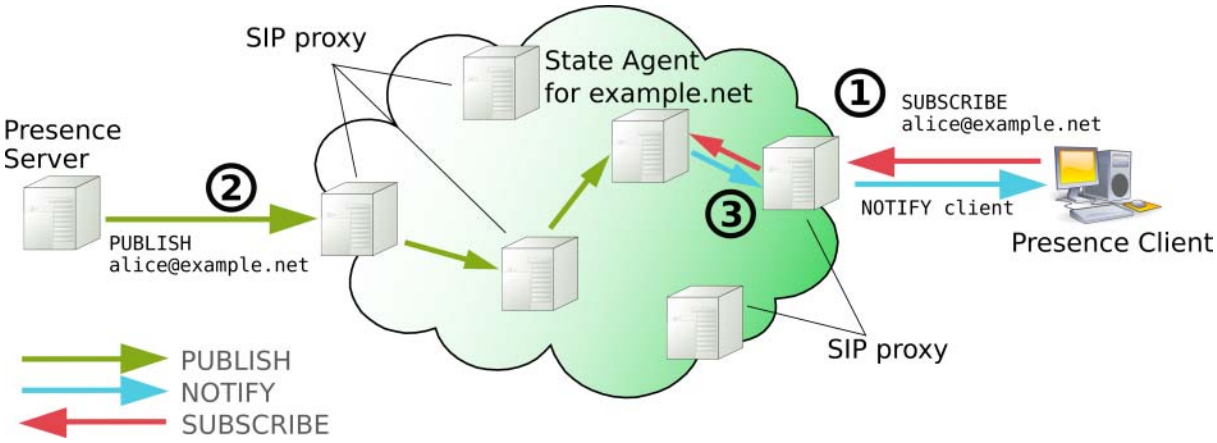

Figure 3.9: Push-based communication in a SIP network

Events in SIP are identified by the URI of the resource which the event refers to, and the *type* of that respective event. Several event types have been defined for SIP, comprising so-called *event packages*—independent modules to be used in distinct application contexts. The *presence event package* for SIP is defined in [RFC3856] in conformance with the fundamental requirements of [RFC3859]. It uses the *presence information data format* (PIDF) that has been standardized by the IETF as a core vocabulary for protocol-agnostic specification of presence-related information. The PIDF vocabulary is discussed in more detail in Section 4.2.2 together with specific extensions that allow for more expressive presence information.

To learn that somebody has subscribed one's presence state, the SIP event framework provides a *template package* to represent watcher information [RFC3857, RFC3858]. Using this, a presence agent can subscribe its own presence status to get notifications about other subscriptions. In addition to the event type, the external watchers are interested in, the subscription for one's watcher information must use the suffix "`.winfo`" with this event type. Thus, if Alice was interested in the subscribers of her presence status she could subscribe to the resource `alice@example.net` with the event type `presence.winfo`. Notifications sent in response to this subscription then use the content type `application/watcherinfo+xml` as defined in [RFC3857], instead of a presence-specific document type. Example 3.3.3 gives an example for a watcher information document that has been sent to Alice.

An important aspect of the watcher information is the indicated status. Here, two watchers have subscribed to Alice's presence record. Bob's subscription already has been approved and

thus has the status *active*, while John's request is still pending. When a subscription is active, update notifications will be sent to the specific watcher automatically by the state agent. Pending subscriptions, in contrast are not yet authorized and therefore will not get notified when Alice's presence status changes.

Example 3.3.3: Watcher information for Alice's presence record

```
<?xml version="1.0"?>
<watcherinfo xmlns="urn:ietf:params:xml:ns:watcherinfo" version="0" state="full">
   <watcher-list resource="sip:alice@example.net" package="presence">
     <watcher id="10000" event="approved"  status="active">sip:bob@example.net</watcher>
     <watcher id="10012" event="subscribe" status="pending">sip:john@otherdomain.com</watcher>
   </watcher-list>
</watcherinfo>
```

Even though the authorization of watchers is outside the scope of SIP, the watcher information package provides the required functionality for a simple authorization handshake between the state agent and the user whose presence status is being subscribed. Given that a user is always allowed to subscribe the watcher information associated with his own presence status, he will get notified whenever a subscription request was received by the state agent. When receiving a watcher list with entries that are marked as *pending*, the user's presence application will query the user if this subscription is to be accepted. If yes, the state agent's configuration will be updated to reflect the user's decision. To change the configuration, the state agent must offer a suitable interface, e.g. via HTTP or the *XML Configuration Access Protocol* [XCAP].

As soon as the subscription has been accepted, a `NOTIFY` message with a description of the current status of the subscribed event will be sent to the watcher. Thus, any authorized subscription will be answered with an initial status description, even if the requested duration was zero seconds to request an immediate termination of that subscription. `SUBSCRIBE` requests with an expiration time of zero seconds therefore can be used to implement the one-time `FETCH` operation we have defined in Section 3.3.1.

3.3.2.4 Aggregation of Events

In the previous section, we have described the basic event notification model of the Session Initiation Protocol. In this model, every status change signaled to the state agent will cause a notification message to every subscribed watcher. As a consequence, multiple presence notifications would be generated for presentities whose status is synthesized from multiple sources. Therefore, the state agent must aggregate the received event publications and generate a consistent view on the user's presence status. An example for this scenario is given in Figure 3.10.

In this figure, three different network nodes publish presence information for the resource `alice@example.net`. Two watchers have subscribed to this resource. A third watcher has subscribed to some other resource whose status is managed by the state agent for the domain `example.net`.

When a status update for Alice is published by either of the presence sources, the state agent must combine this status description with stored status descriptions from other sources, and publish the result to the subscribed watchers as indicated in Figure 3.11.

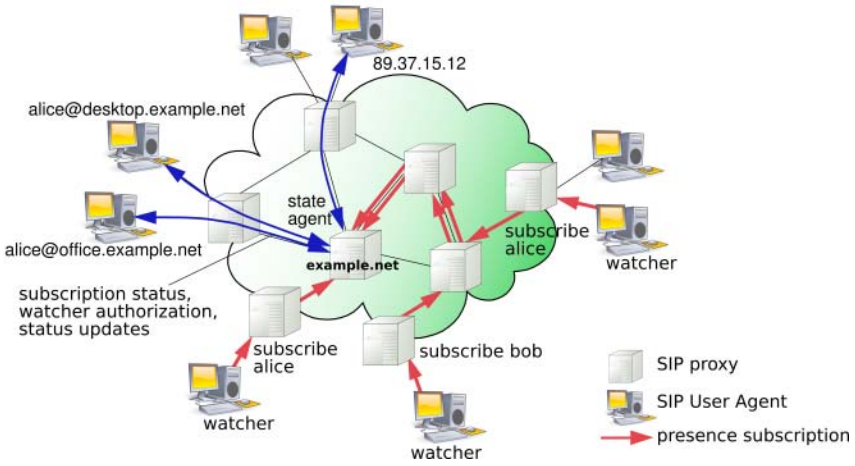

Figure 3.10: Using multiple presence sources

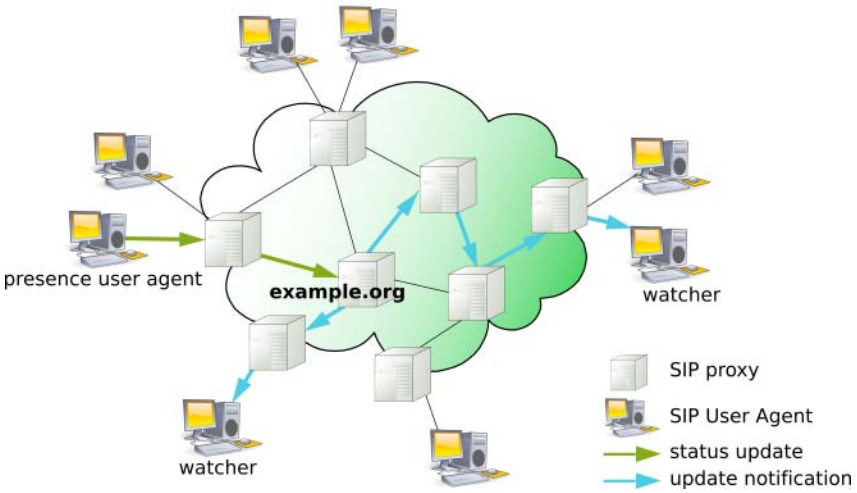

Figure 3.11: Dissemination of status changes

The composition of status descriptions from multiple sources requires additional processing of the received presence information. In general, the following options exist for composing presence documents:

- *Syntax-based composition*

  For syntax-based composition, the distinct presence tuples of all documents are copied into a new document. As the identifier of presence tuples is required to be unique only within the sequence of presence documents of a single source, additional processing might be necessary to ensure uniqueness of tuple identifiers within the new status description (e.g. prefixing identifiers with a unique identifier for each source). This composition technique is illustrated in Figure 3.12.

- *Semantic composition*

  If more control is desired, the composition must be performed with respect to the semantics of the presence data, i.e. the state agent must have additional knowledge about the meaning of presence tuples contained in the input documents. Semantic composition is useful especially for aggregation of abstract status descriptions published by presence servers in a local presence environment as described in Section 3.2. Today, neither standardized SIP mechanisms nor research efforts for semantic composition of documents at state agents exist.

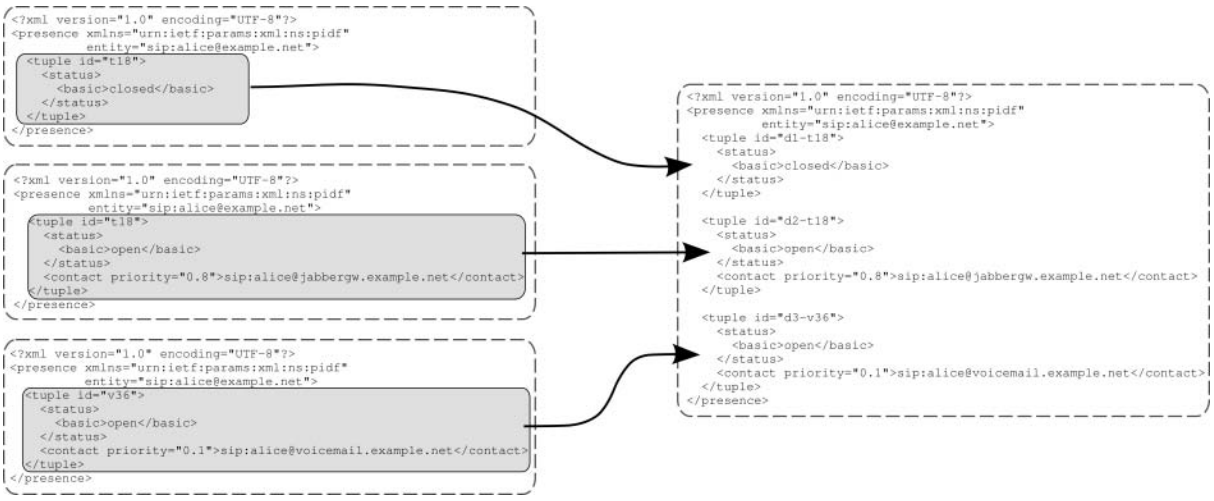

Figure 3.12: Syntax-based composition of presence documents

As semantic composition of presence information is outside the scope of SIP, no direct support exists in this protocol. To use it with our presence aggregation service to provide a more flexible composition technique, a common data-model for the aggregation engine and the contributing sources must be provided as well as an aggregation language to specify how information items published by different presence sources should be composed.

3.3.2.5 Summary and Evaluation

In this section, we have given an overview of the Session Initiation Protocol (SIP), with a specific focus on the distribution of presence information. The DNS-based application-layer routing of SIP constitutes an overlay network that can be used for efficient transmission of presence information. The limited lifetime of SIP messages and transactions facilitate the implementation of SIP servers whose memory consumption is proportional to the number of incoming requests. The soft-state subscription mechanism of [RFC3265] in addition helps avoiding old dialog state to be kept indefinitely on the server.

A major advantage of SIP compared to HTTP is the support for push-based communication between SIP nodes, i.e. it enables instant notification of status changes to subscribed watchers. In summary, SIP provides the following advantages over HTTP-based approaches when used as a protocol for conveying presence information through the public Internet:

- *Support for external triggers*

  Watchers are notified immediately when the status of a subscribed resource has changed. In addition, the current status can be fetched in a one-time operation to facilitate bootstrapping of new watchers.

- *Immediate status updates*

  Notifications in the push-based architecture of SIP are sent instantly whenever a status update occurs, i.e. the only delay between the status change and its observation by watchers is caused by the time the information dissemination takes.

- *Reliable transmission of status updates*

  As watchers are notified of status updates immediately after their occurrence, every change is communicated. Information loss can happen only if the frequency of status changes is higher than the maximum frequency for sending notification messages. For example, if the time for message transmission and processing takes 500 ms, the status must not change more frequently, as the receiver would not be able to reduce the notification backlog.

- *Event throttling*

  Servers can control the load that outgoing status update notifications generate. While the pull-based mechanism only allows for implicit flow control by deferring responses, the push-mechanism enables throttling of event notifications. If, e.g. frequent status changes would overload the sender's message processing capabilities, several status updates could be accumulated into a single notification message.

- *Simultaneous sending of update messages to multiple watchers*

  Several watchers can be notified of status changes simultaneously to ensure a consistent view of the overall system status (given that the average round trip time at the transport level is largely the same for every subscribed watcher).

- *Clear semantics of transmitted information*

  The presence profile defined in [RFC3859] together with the presence document format specified in [RFC3863] define concise semantics for the information items contained in a presence status description. Thus, presence information can be sent to and received from other presence domains, even if SIP is not used to convey the presence data internally.

  Other presence protocols such as the *Extensible Messaging and Presence Protocol* (XMPP, see Section 3.3.3.1) use custom formats to express presence information hence making interconnection of different presence domains more complex. Even worse, upcoming solutions in the World Wide Web (c.f. Section 3.3.1) use language-specific object notations such as the *JavaScript Object Notation* (JSON[20]), a format to convey serialized JavaScript objects. Mapping these document formats to each other then requires specific parsers and a well-defined mapping function to ensure interoperability of presence systems from different vendors.

[20] See <http://json.org>.

In addition to these features that qualify SIP as a protocol for efficient dissemination of presence information, we have argued that SIP can be used to enable presence aggregation at intermediaries on the messaging path. In particular, presence sources and state agents can be equipped with user-provisioned rules for semantic composition of presence documents from multiple sources. The aggregation result then can be forwarded to subscribed watchers according to the processing rules of the respective state agent.

Finally, SIP is superior to various other presence protocols as it provides a tight integration of setup, management and termination of multimedia sessions. Being used as the primary call-signaling protocol for IP-based telephony services, SIP is widely deployed and thus provides an appropriate basis for presence-aware communication services as postulated in Section 3.1.

3.3.3 Alternative Distribution Architectures

After we have discussed various options for the distribution of presence information through the global Internet in Section 3.3.1, the Session Initiation Protocol (SIP) was identified as a robust and scalable protocol for managing presence subscriptions and conveying presence documents in event notification messages. SIP provides a set of operations with clear semantics and well-defined state transitions in both, the initiator and the responder of a transactions. The protocol's addressing scheme allows for decentralized management of object identities in a scalable, heterogeneous network infrastructure. Thus, the Session Initiation Protocol meets all requirements we have identified in Section 3.3.1 as crucial aspects of Internet-scale presence services.

In the past, there has been a large number of proposals for wide-area presence distribution protocols that use different architectures from those of the World Wide Web or the Session Initiation Protocol. In the following section, we comment on two of the most important distribution architectures and correlate them to SIP. We begin with the *Extensible Messaging and Presence Protocol* (XMPP), an XML-based mechanism for conveying instant messages and presence information. XMPP is based on the proprietary *Jabber* protocol for instant messaging and presence with an architecture and a data model that differ from corresponding SIP concepts.

After the comparison of SIP and XMPP, we discuss *content-addressable networks*, a distribution architecture that is not tied to the centralized DNS name resolution mechanism and hence is less vulnerable to outages of the global routing infrastructure. In this discussion, we focus on the distribution of presence information, especially in comparison with SIP. A complete description of all features offered by the respective protocols is deemed out-of-scope for this document.

3.3.3.1 The Extensible Messaging and Presence Protocol (XMPP)

The *Extensible Messaging and Presence Protocol* (XMPP) has been standardized by the IETF as a successor of the proprietary Jabber[21] messaging protocol. XMPP [RFC3920] is designed explicitly as a lightweight protocol for presence-aware instant messaging applications. Therefore, it has native support for presence and provides basic mechanisms for securing XMPP communication streams using *Transport Layer Security* (TLS) and a specific *Simple Authentication and Security Layer* (SASL) profile.

[21] The homepage of the non-profit Jabber Software Foundation can be found at <http://www.jabber.org>.

Using TCP as underlying transport protocol, XMPP only provides unicast communication, either between a client and a server, or between two servers. The server-to-server communication facilitates the creation of an overlay network of interconnected XMPP servers for dissemination of presence data to connected clients. The resulting XMPP mesh network is visualized in Figure 3.13.

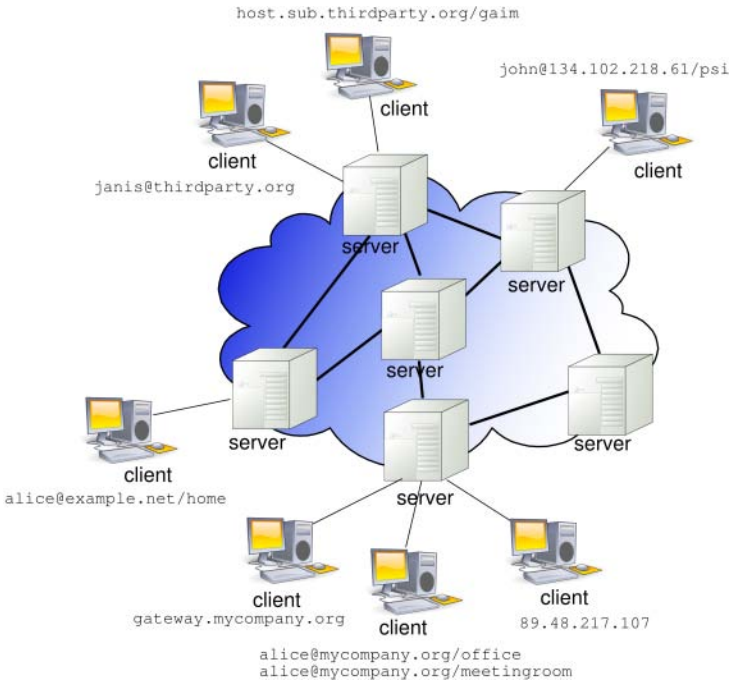

Figure 3.13: XMPP overlay network

As depicted here, the XMPP core network is made up of XMPP servers, i.e. relays that are responsible for the interconnection of distinct administrative domains and may act as gateways to foreign presence protocols. XMPP servers thus take the role of SIP proxies, whereas messaging streams are terminated or originated by XMPP servers, while SIP proxies act as a forwarding hop that preserve the end-to-end relationship between communicating clients.

XMPP clients connect to an XMPP server that is responsible for their domain which is referenced in a unique identifier—the *Jabber ID*.—that was assigned to the client. Besides a domain name, a Jabber ID may contain a node identifier (e.g. a user name) and a resource name to distinguish several devices owned by the particular user. Examples for different formats of Jabber IDs are given in Figure 3.13. Note that valid Jabber IDs may consist of the domain part only, as this is sufficient to ensure routing of XMPP protocol elements. These consist of XML fragments—called *stanzas*—that are carried within a (typically) TCP-based XML session. The session starts with an XML declaration followed by an XML start tag for the document element `stream` in the namespace `http://etherx.jabber.org/streams`.

The first step in any XMPP session is the negotiation of security parameters using the *Simple Authentication and Security Layer* (SASL) protocol [RFC2222]. Because of the server-based relaying of protocol elements, XMPP streams can be secured on a hop-by-hop basis only, typically using *Transport Layer Security* (TLS) between any two hops.

The XMPP message flow between a client and a server is illustrated in Figure 3.14. First, the client initiates a connection and sends the opening sequence to the server. If the server accepts this XMPP session setup request, it responds with a similar opening sequence, containing an attribute `id` with a unique stream identifier for that server. After the following SASL negotiation, the client sends a `presence` stanza to publish the client's current presence status. As there is no attribute `to` contained in the start tag of the element `presence`, the status change will be published by the server to any subscribed watcher, shown here as `john@otherdomain.com` (assuming that John has previously subscribed to Alice's presence status). At some point, the streams are closed and the TCP connection is released.

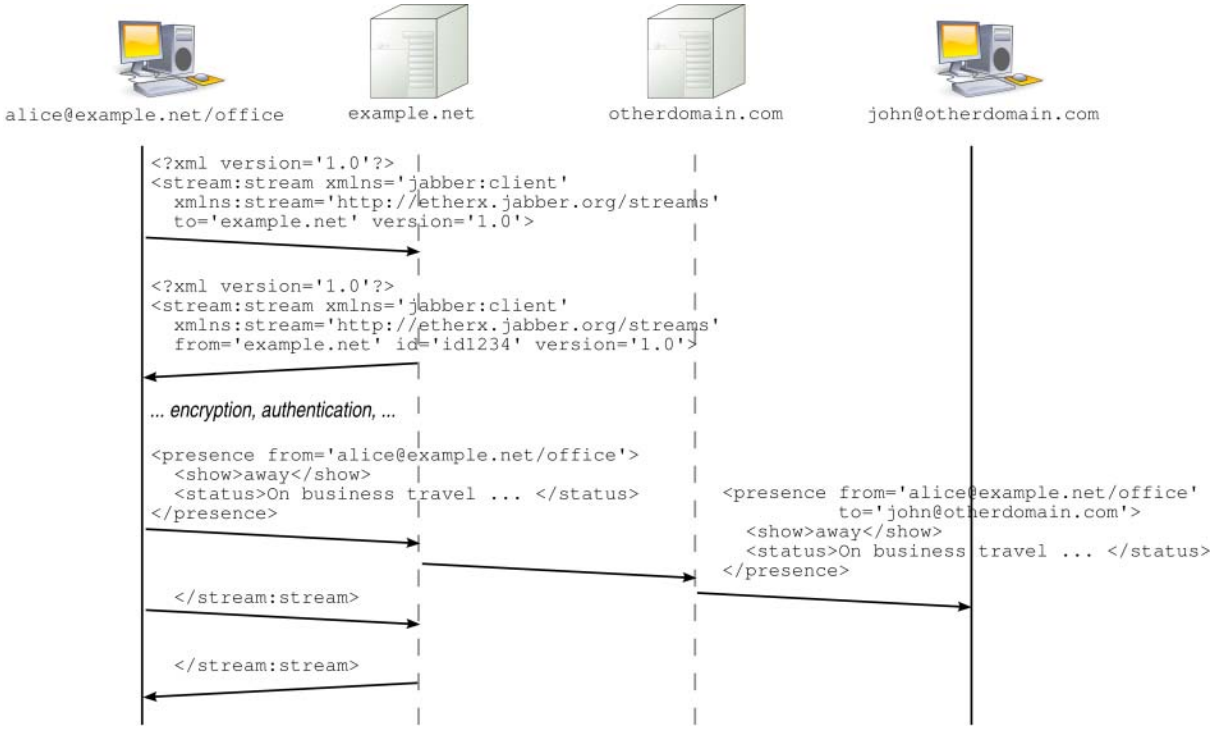

Figure 3.14: Example XMPP protocol flow

Using the presence vocabulary of the core protocol, only hop-by-hop security is available. To achieve end-to-end encryption and signing of presence information, the data must be converted to a PIDF document. This document then can be signed and encrypted, and is carried as opaque content of an `e2e` element within the `presence` stanza as specified in [RFC3923]. A detailed discussion of XMPP's custom presence format as well as the encapsulation of PIDF documents is given in Section 4.2.3.

In summary, the *Extensible Messaging and Presence Protocol* provides a connection-oriented distribution mechanism for XML-based presence documents. The dissemination of presence information is based on explicit subscriptions to named XMPP entities. Distinct resources of a presentity can be identified using a specific part of the particular resource's unique Jabber ID. Aggregation of presence information is possible at each trusted XMPP server instance, unless the presence information is carried as signed and encrypted PIDF document.

Since XMPP has been designed as a protocol for instant messaging and presence, is provides a vocabulary for expressing a user's presence status as well as routing mechanisms for

simple ad-hoc text messages, and the management of presence subscriptions. Several extension proposals (so-called "JEPs") exist, that provide enhanced vocabularies for presence, media negotiation, and setup and termination of real-time multimedia communications. The multimedia session signaling has been proposed in [JEP-0166], and may turn out to be an important amendment for leveraging XMPP as a full-fledged call signaling mechanism. With the *Jabber Enhancement Proposal 0060: Publish-Subscribe* [JEP-0060], a powerful mechanism for presence aggregation has been announced. A brief discussion of this module is again contained in Section 4.2.3.

3.3.3.2 Content-Addressable Networks (CAN)

As SIP and XMPP build on a network architecture similar to the World Wide Web, both protocols use application-layer message routing based on the global DNS infrastructure. In the past few years, distributed architectures have evolved that are independent of any centralized network infrastructure such as DNS and thus can be used for ad-hoc messaging and presence even if no Internet access is available. With *content-addressable networks*, architectures for decentralized computing with minimal or even no centralized components have become reasonable alternatives to traditional client/server architectures. Content-addressable networks are a specialization of *Peer-to-Peer* (P2P) networks that use deterministic mathematical hash functions to distribute the address index over the participating nodes.

P2P networks in general consist of nodes with—at least conceptually—equal roles. Any node in a P2P network is able to initiate requests, i.e. to act as a client, and to process received requests as a server component. Depending on the exact architecture, nodes may even forward messages to their actual destination. Early P2P protocols relied on a tree-based forwarding mechanism similar to distance-vector multicast routing, causing a large protocol overhead. Content-addressable networks are P2P networks that require a minimum number of key lookups from the *distributed hash table* (DHT) to find the corresponding value that is stored with the given key. [RFH+01, Rat02] These optimized P2P networks therefore combine server localization and message forwarding, achieving a worst-case complexity of $O(\log_n)$ hops between any pair of nodes. [SMK+01] distributed hash tables used in these variants of P2P networks thus provide an interesting design option for routing of SIP messages as shown in [RyWo03, SiSc04b]

As shown by Schollmeier et al. in [SGF02], the completely decentralized routing infrastructure of P2P networks ensures increased scalability and robustness regarding traditional server-based messaging systems. Although the network's topology and routing architecture is hidden from the application layer, the lack of centralized infrastructure components, especially the hierarchical Domain Name System, and unpredictable loss of connectivity require careful design of protocols and implementations. As the P2P technology is typically used in overlay networks of the public Internet, these issues are rarely addressed in literature.

P2P technology in some way resembles fundamental principles of the Internet architecture which have been sacrificed when building networks and services that rely on centralized servers or require the presence of intermediaries, examples of which are Web-Services and network address translators. In particular, P2P uses a dumb network, with the application intelligence being distributed over the participating nodes. The nodes in addition share a common fate as the routing depends on all (or at least a large number) of collaborating entities. And, due to the absence of intermediaries, all communication between nodes has end-to-end significance.

As a result, P2P networks eliminate numerous issues that are known to affect scalability and robustness but also complicate secure communication between individual users.

The previous considerations make content-addressable networks look ideal for large-scale distribution of presence information. A major downside is the high protocol overhead caused by the dynamic network topology. In particular, routing of messages and group security could render P2P networks unusable for applications with high message rates. Existing applications such as *Skype*[22] therefore use a hybrid architecture with a central entity that coordinates access to the system. There is only little official information on the Skype protocol available, so we can give only a short illustration of the product's core functionality. Technical details mentioned here are only speculative, as found from the re-engineering effort that has been done by Baset/Schulzrinne, found in [BaSc04]. Skype entities use a central server only for bootstrapping, specifically for user authentication. Once a node has detected a reasonable number of online peers, its routing table is filled with their addresses, and the node becomes a natural part of the network's distributed routing architecture.

The Skype architecture is a hybrid P2P network to guarantee availability of a sufficient number of nodes that have persistent Internet connectivity and that are not behind a firewall or address/port translators (NATs), respectively. That way, these entities—called *super nodes*—can provide NAT or firewall traversal services to any other node within the Skype network. The super nodes are considered to be essential for routing in the Skype network as any Skype client needs to contact at least one super node at startup. To assure scalability, a super node is responsible for a cluster of client nodes, the exact size of which is not known. The set of reachable super nodes is also used to guarantee robustness of the Skype system, as any client maintains a list of alternative nodes to be contacted when the active super node for that client is not available any more.

Although only little information exists on the Skype protocol, the observations made by Baset and Schulzrinne show that P2P networks can be used to provide fully distributed, scalable and robust messaging systems with support for user presence. A disadvantage of those systems is the tight trust relationship between participating nodes, as a client must rely on other clients to route messages correctly. In addition, presence aggregation is difficult because of the dynamic routing tables. Whenever a node leaves or a new node joins the P2P overlay, the distribute hash table is re-arranged to adapt to the new situation. Presence aggregation, in contrast, relies on a static message path traversed by presence documents. Along this path, user-provided aggregation specifications can change the document contents or create watcher-specific views on a particular document. In a content-addressable network, aggregation specifications would have to be retrieved from dedicated servers on demand, hence generating additional messaging overhead whenever the distributed hash table has changed.

3.3.3.3 Summary

In this section, we have compared presence information distribution using the Session Initiation Protocol with two alternative approaches that represent typical architectures found in recent literature on presence services. The Extensible Messaging and Presence Protocol (XMPP) has become popular for instant messaging, as it is a lightweight protocol with a small vocabulary and thus easy to implement. XMPP benefits from its predecessor Jabber which is widely de-

[22] See <http://skype.com>.

ployed and can be scaled from a single-server application for small user bases up to a mesh of interconnected servers for large organizations or wide-area communication even at Internet-scale.

As a reaction to the forthcoming standardization of SIP-based instant messaging and presence services, the XMPP/Jabber community has authored a number of "enhancement proposals" to add new features to the XMPP core protocol. Among these features are the setup and termination of real-time multimedia sessions and aggregation of presence documents as well as the encapsulated transport of PIDF documents. (In Section 4.2, we will discuss the content handling of presence aggregation services in detail, including the options provided by XMPP.)

As presence is especially useful in ad-hoc networks where no central routing infrastructure or name resolution is available, several research projects have investigated the adaptation of existing presence protocols such as SIP to peer-to-peer architectures. In particular, content-addressable networks that provide optimizations for retrieval of addressed nodes in $O(\log_n)$ time have been used to implement presence-enabled instant messaging clients.

Comparing XMPP and content-addressable networks with SIP, we found that neither technology was superior to the others. While XMPP is widely deployed for instant messaging and presence, it has not yet gained the importance of SIP by telephony service providers. As a consequence, only few providers offer gateways to establish phone calls with the public switched telephony network (PSTN) via XMPP.

A similar observation was made for peer-to-peer (P2P) technologies. The most popular representative for P2P-based interpersonal communication is the proprietary *Skype* telephony service. Although interfacing to the PSTN is possible with this service, no integration with other communication services in the Internet is available. Moreover, the highly distributed routing architecture of content-addressable networks makes a presence aggregation service very difficult to implement.

Summarizing these observations, we have decided to use the Session Initiation Protocol for Internet-scale distribution of aggregated presence information. The protocol fits perfectly into the Internet multimedia conferencing architecture proposed by Handley et al. in [HCB+99], as it provides open interfaces for integration of other communication services. As we have shown in Section 3.3.2, the SIP architecture does provide a suitable platform for creating a presence aggregation service.

3.4 Security

As a presence service reveals user-specific information, e.g. about activities, habits, communication devices, or calendar information, it can be abused for surveillance or profiling of that particular user. Many people thus are reluctant to use a public presence service even if the service provider ensures privacy protection for the user-specific data to persons that have been explicitly accepted as authorized receiver of the presence information. In the previous sections, we have argued that users get more confident in a socio-technical system as they learn to control the system's complexity. As a consequence, we have proposed an aggregation service where users can influence the publication of their personal presence information at any presence server within their domain.

While the ability to control the presence service is crucial for its acceptance by users, technical measures must be taken to ensure the system's integrity throughout its use. In particular,

care must be taken to protect sensitive data from unauthorized access and inhibit publication of forged presence data. We discuss the security goals for Internet-scale presence services in Section 3.4.1, and point out the common threats to this kind of protocol in Section 3.4.2.

3.4.1 Goals

Being a distributed information system that is based on an untrusted and insecure transport channel, a presence service must ensure confidentiality and integrity of the transmitted presence information. In addition to these fundamental requirements of Internet-based applications, the *fair information practices* defined in [RFC3694] for the *Geopriv* protocol also list the following goals for processing sensitive personal data. We have commented each goal according to the requirements of our presence aggregation service:

- *Openness*

  Users should be notified when a subscription to their presence status has been established. Ideally, the subscriber allows the subscription of his presence status as well (*reciprocity*).

- *Individual participation*

  Users must be provided with a means to control the process of presence aggregation and distribution of aggregation results.

- *Collection limitation*

  Presence servers as well as end-systems that process presence information should store personal information only temporarily for the purpose of enabling communication. The collection of data to construct user profiles should be prevented.

- *Accountability*

  Subscribers must be accountable for adhering to these practices.

Most of these goals cannot be achieved by technical measures alone. Instead, users of the presence service have to be educated to be cautious with their personal data, and to use the privacy features of the system. Specifically, if authorization-based filtering is offered by the presence server, a decent authorization specification should be constructed to limit the amount of information revealed to barely known subscribers. An example authorization specification is presented in the following section, together with a discussion of typical threats to presence services.

3.4.2 Threat Analysis

The threat analysis for a presence aggregation service at first takes into consideration the typical security risks for any distributed system, i.e. eavesdropping, man-in-the-middle attacks, forging identities, denial of service. Server systems and protocols thus have to be protected against disclosure of sensitive data, identity theft, and faked presence messages. The following list shows the technical countermeasures that are usually provided to prevent the mentioned attacks.

- *Data encryption*

  To protect a system or a protocol against eavesdropping, messages and stored data must be disguised by a cryptographic function that is reversible only with specific knowledge or brute force.

- *Authentication*

  The identity of communicating parties can be assured by presenting a cryptographic puzzle that can be solved only with specific knowledge such as a password or a private key.

- *Message integrity*

  To prevent a system against injection of forged messages, a digest is calculated over the message and signed with a cryptographic function.

The security mechanisms listed here are well-understood regarding their strengths and weaknesses, and have been successfully deployed in various distributed applications. Unfortunately, several types of attacks exist that cannot be defeated entirely with these cryptographic mechanisms. A denial-of-service attack, e.g., might exploit the multiplier effects of subscribe/notify systems to bring down the entire presence service. To do so, a large number of forged notifications is sent to a presence agent, which then sends a status notification to each watcher who is subscribed for the respective user's presence status. The larger the average size of the watcher list, the heavier is the impact of a single forged `PUBLISH` message. For example, an update of a status record that is subscribed from 20 different users will cause 20 notifications to be sent through the network.

The push-based notification service may also tempt marketing experts to send unsolicited notification messages with commercial advertisements to presence-aware clients. This form of mass marketing is known from the SMTP-based message exchange where recent surveys [Fal03, Fal05] found that the level of so-called "spam" messages per month has increased from 40 % in 2003 to 73 % in 2004. Therefore, it seems reasonable, that mass marketing will explore new territories as soon as they become available and are adopted by a sufficient large community of users.

To foster user acceptance of Internet-scale presence services, these technical issues have to be addressed in combination with the socio-technical issues that have been discussed at the beginning of this chapter. In particular, a secure distribution service must not only ensure data protection, but it must also provide flexible policies for watcher authorization and information filtering. The following key questions can be used as a guidance for the design:

- *Who is given access to the data?*

  Watchers must be explicitly authorized to be notified of a user's presence status, after positive identification of the particular watcher has taken place.

- *What data is published?*

  Depending on the access permissions a watcher has, a presence server can filter out presence information or distort sensitive data prior to sending a status update notification that describes the current presence status.

- *How is the data distributed?*

  Sensitive data always must be transmitted over a secure channel to avoid eavesdropping or tampering with the published data.

The third question again refers to the secure transport of presence information through the network and is covered by the security mechanisms of the Session Initiation Protocol. Unfortunately, a major obstacle exists for ensuring privacy in the public SIP network. Most commercial providers of telephony services currently do not offer termination of secure SIP signaling sessions using TLS or S/MIME because of the additional resources that are necessary for cryptographic calculations. In addition, the network setup typically inhibits end-to-end security as intermediaries such as Session Border Controllers are used to protect the edges of each provider's network.

If the missing encryption is tolerated, users can benefit from the authentication of watchers that is done by commercial telephony providers and its trusted peering partners. The resulting *Web of trust* then facilitates identification of users from foreign domains as they have authenticated with their home proxy.

In order to answer the first and the second question of the previous list, the operator of the presence service must define an according policy addressing the authorization of watchers and the filtering functions that are applied before any personal data is sent through the network. The authorization of watchers to access a presentity's presence status—or parts thereof—is based on the watchers' identities, i.e. their unique URIs. In today's presence systems, watcher authorization is usually implemented as a handshake between the person or software agent that subscribes to a presence source, and the user whose data is published by this presence source. When queried to grant access to his presence data, the user might reject the request, and no subscription will be created. If the subscription is accepted, the requesting user will get notified from any status change that occurs until the subscription is removed.

One disadvantage of this authorization mechanism is its coarse granularity: The user can only choose between revealing all details of his presence status, or rejecting any information. Therefore, it is necessary to filter the information sent to authorized watchers depending on their identities. Every presence attribute within a status description then will be annotated with a filtering expression to be checked against the URIs of the subscribed watchers. Before sending a document to a particular watcher, every attribute is removed that the receiver is not permitted to see. In Figure 3.15, this watcher-based filtering is illustrated by a presence source publishing a status description that contains presence attributes annotated with watcher-authorization information. The presence server interprets these annotations and sends out two different versions of this document, one to each watcher.

Common practice is the grouping of watcher URIs to authorization classes that can be referenced in an authorization specification. Instead of annotation presence attributes with complex filtering expressions, the authorization information then can be given as a class name that is defined in an external authorization specification as depicted in Figure 3.15. An example authorization specification that can be used with our presence aggregation service is given in Example 3.4.1.

Here, the five authorization classes *private*, *friends*, *family*, *protected*, and *public* are defined. In this specification language, every authorization class represents a set of regular expressions that are tested against the URIs of subscribers. To facilitate creation of authorization classes, the language allows for combination of classes using set operations and references.

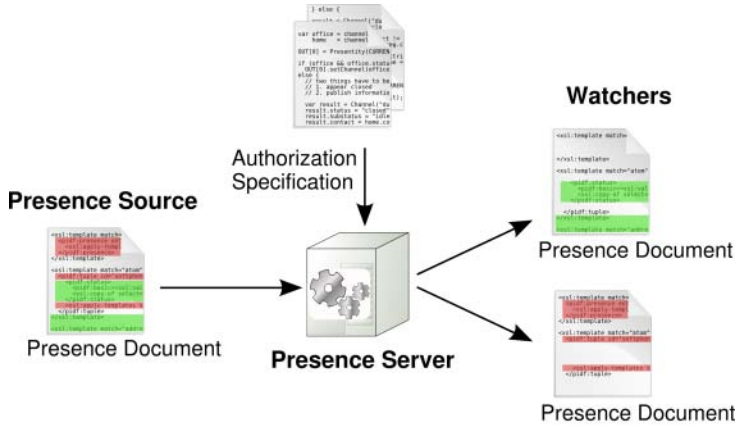

Figure 3.15: Watcher-specific filtering of presence attributes

The authorization class `public`, e.g., is constructed from any URI except those that are also member of the class `protected`.

Having defined this set of named authorization classes, presence documents can refer to these classes by their name. As discussed in Section 5.3, we have decided to extend the standardized presence information data format by a new element type `authclass` that carries the name of the corresponding authorization class for a presence tuple. This technique is also used in several other languages for specifying authorization classes such as the rule-based filtering language that is discussed in Section 4.3.2.

3.5 Summary

In this chapter, we have discussed our analysis of limitations of existing groupware systems that have been documented in literature. Today, CSCW systems try to hide differences between participants regarding time-zone, connectivity, or location. In Internet-scale systems some of the underlying assumptions that originate from enterprise communication systems are not valid any more. The design goals for global cooperation therefore deviate from those of the local approach.

After discussion of the issues we have identified, a set of guidelines for Internet-scale presence applications has been proposed. The rationale is to give control of the communication service back to the users and let them manage their media sessions on their own. The presence service then is conceived as technical infrastructure to support users without constricting their activities. Automation of tasks is controlled by user-specific profiles defining the preferences for specific decisions.

To enable evaluation of existing systems and possibly design a new type of presence service, the abstract guidelines for user-friendly presence applications have been refined into a list of functional requirements. After that, we have examined protocols and architectures for dissemination of presence information, with a specific focus on user-controlled presence aggregation. As a result, the Session Initiation Protocol (SIP) has been identified as a scalable and sufficiently flexible protocol that meets the functional requirements imposed on an Internet-scale presence aggregation service and integrates well with multimedia session setup and management.

Example 3.4.1: Specification of authorization classes

```
<authdef>
  <authclass name="private">
    <authelem>alice@example\.net</authelem>
  </authclass>

  <authclass name="friends">
    <authelem>Bob@otherdomain\.com</authelem>
    <authelem>[^@]*@mysportsclub\.com</authelem>
    <authelem>jill@example\.net</authelem>
  </authclass>

  <authclass name="family">
    <authref name="private" />
    <authelem>philip@example\.net</authelem>
    <authelem>edward@example\.net</authelem>
    <authelem>anne@example\.net</authelem>
  </authclass>

  <authclass name="protected">
    <authref name="friends" />
    <authref name="family" />
  </authclass>

  <authclass name="public">
    <set-difference>
      <authref name="protected" />
      <authelem>.*</authelem>
    </set-difference>
  </authclass>
</authdef>
```

From the discussion of the security goals for the presence aggregation service, we have found that the trade-off between privacy of personal information and the need to publish parts of this information to describe the current presence status has an effect on the security measures to be provided by a presence system. An essential feature of every presence service hence is the support of watcher-specific presence publication. The presence tuples contained in a source document are annotated with identifiers of custom authorization classes. These authorization classes comprise the URIs of persons or entities that are allowed to subscribe to a specific subset of the respective user's presence status. This way, users can control who is permitted to subscribe to their presence status, and can specify the subset of information that is published to each watcher.

Chapter 4

Related Work

The previous chapter has presented our analysis of functional and non-functional requirements for presence-aware communication services. As a major result, we found that the entire process of aggregating presence information and publishing results must be in control of the user who owns the data sources, and watchers should be aware of this fact. Presence-aware communication has been a popular research topic in the past, so numerous activities exist where we can build upon. A large number of these projects target on the automated acquisition of parameters that define a context, vocabularies for context representation, transport of presence data and on facilitating text-based ad-hoc communication such as instant messaging.

In this chapter, we present a detailed overview of the activities which had a significant influence in this area of research and which we considered relevant for our work, with a particular focus on the requirements stated in Chapter 3. Our analysis of results from related projects has focused primarily on the following topics:

- Generation of raw presence data,
- aggregation of raw presence data from multiple sources,
- representation of abstract presence information, and
- wide-area distribution of presence information.

In most systems, the generation of presence data is closely related to the aggregation process, and vice versa, while the data representation and distribution is—at least to some degree—decoupled from the other aspects. Therefore, we treat data generation and processing as a single topic in our discussion, apart from the representation and distribution of presence data, each of which is discussed in its own section.

Furthermore, we have found that most of the existing work on presence aggregation falls into the three categories *presence distribution services and event notification systems*, *context-awareness*, and *future context prediction*. The first category, *presence distribution services and event notification systems* primarily targets at the efficient dissemination of presence information in wide-area networks such as the Internet, while the other categories focus on the generation, representation and processing of this information. As the information distribution already has been discussed as part of our requirements analysis in Section 3.3, we are focusing now on the detection of context to enhance interactive communication in Section 4.1. An

important aspect which is orthogonal to the presence data generation and processing, is the protocol-agnostic representation of abstract presence information we will discuss in Section 4.2. Existing languages for controlling the process of data aggregation are discussed in Section 4.3. A brief evaluation of our analysis is given at the end of each subsection. Section 4.4 concludes this chapter with a summary of our findings from an investigation of the existing approaches to presence aggregation.

4.1 Context-Awareness

According to our definition given in Chapter 2, presence information denotes the availability and willingness of a user to communicate with other peers, including the description of preferred media for conversation. This definition is closely related to the notion of *context-awareness* in systems that process device-specific status information at a server within the network in order to provide an abstract view on the owner's location, activities or behavior. Context-aware applications are able to adapt their behavior and possibly their presentation dynamically to changed conditions of the environment. For example, a notebook computer may dim the screen's backlight in the darkness, while turning up brightness in sunshine conditions. The context information can be transferred between applications either explicitly by sending a complete status description, or implicitly through events that update the local event state of each receiver.

In this section, we specified context-awareness systems are analyzed for their handling of abstract presence information in order to establish a conversation with that particular user. In a very basic presence system, a single sensor could be used to determine if a particular person is in a given context. The audio-visual link between two sites of the Xerox company described in Chapter 2 and the door sensor of the Portholes system are examples of such a system. As these early presence systems only support a small set of sensors with well-defined semantics, no generic aggregation mechanism is needed. The more complex the technical infrastructure gets and the more generic applications become that use this infrastructure, the more flexible and scalable mechanisms for aggregation of sensor data must be provided. Cohen et al. in [CPT+01] come to the conclusion that these aspects are the key challenges for aggregation systems. In addition to efficient data distribution and composition of data from multiple sources, the authors have identified automatic detection of data sources, dynamic security policies, and a concise programming model to be important characteristics of data aggregation services. The programmable composition of data sources should occur in a "time-dependent manner" [CPT+01] to increase the flexibility of high-level operations on available data traces.

In this section, we discuss a number of systems and frameworks that address these challenges in several ways. Many of these systems use *content-based networking* as underlying distribution infrastructure, i.e. routing based on structured message content as described in [CaWo03]. First, we discuss in Section 4.1.1 the *Solar* project that defines data aggregation as a filter function located in the nodes of the distribution environment. Applications can subscribe to a specific context that is an implicit result of the aggregation process. As an example for automatic context determination using Solar, a related project is described that builds on the Solar system to detect if a person attends a meeting and thus is not available for interactive communication.

An example for powerful context determination using sensor-fusion based on mathematical models is provided by the European research project *TEA*, we discuss in Section 4.1.2. A

similar approach is followed by the *GeoBots* project we discuss in Section 4.1.3. Here, the major focus is on providing information to mobile users who enter or leave a specific spatial context, i.e. announce network services, send commercial advertisements, etc. To do so, context information is made explicit and is carried through the network using a proprietary extension to the Session Initiation Protocol. A similar approach is followed by the framework for Context-aware Communication Services of TU Darmstadt that we describe in Section 4.1.4.

4.1.1 Solar

The *Solar* system described in [ChKo02, Che04] provides a *content-based network* (cf. [CaWo03]) with support for context-aware adaption of application behavior. An important aspect of Solar is the understanding that the raw data collected from sensors is too limited and too inaccurate to allow for direct inference of application context. The system therefore provides operations to filter, transform, and aggregate raw data into more abstract information such as user presence.

The composition of data source is based on the filter-and-pipe pattern, where filters map a set of input ports to a set of output ports. Data sources in this model are filters that have no input ports, and sinks are filters without any output port. The connection of filters, sources and sinks is modeled by pipes, ensuring that only compatible types of ports are connected to each other. An aggregation specification thus can be treated as directed acyclic graph with sensors being the source nodes and applications being the data sinks. The graph is serialized in XML, with additional annotations for controlling aspects of filter processing such as rate control for the underlying application-level multicast routing mechanism.

To define contexts, Solar uses the *Intentional Naming System* (INS) described by Adjie-Winoto in [ASB+99]. Based on this technical infrastructure, contexts are defined as structures containing key/value pairs that are announced by data sources. Subscriptions to contexts are made by *queries* that are constructed like context definitions with less specific data fields. Both, context definitions and context queries use a nested structure that is suited for localization of data sources along the Solar sytem's multicast-based *dissemination path*.

The following example visualizes the naming of data sources (sensors) that are identified with a specific context. As context definitions, the names comprise sets of key/value pairs. Each key may contain either a literal value or another set of key/value pairs. For example, a sensor might be described as follows:

```
[ sensor      = "temperature",
  scale       = "Celsius",
  granularity = "room"
  location    = [ building = "Pearson",
                  room     = "102",
                  address  = [ city   = "London",
                               street = "Gower Street",
                               zip    = "WC1E 6BT"
                             ]
                ]
]
```

A query for this sensor now would contain the key/value pairs that are required to be in the search result, while omitting the details of a specific sensor. To get all sensors within a specific office of the Pearson building, the following search expression might be used:

```
[ location = [ building = "Pearson", room = "102" ]]
```

Contexts are described in the same way. A person may e.g. have an attribute `location`, containing key/value pairs for the building, the room, and possibly the building's postal address. Since persons usually move around, this location information may change over time. This dynamic change is reflected by operators that can be used in resource names. Given an operator `$locator`, a specific sensor at a person's current location while in the Pearson building may be described in the following way:[23]

```
[ location = [ building = "Pearson", room=$locator:room ]]
```

An example for the use of Solar for context determination is given by Wang et al. in [WCK04]. Two types of sensors are used in an office to detect if a meeting is currently in progress. The detection is based on motion sensors and pressure sensors that are mounted to every chair around a meeting room table within that particular office. The data delivered by sensors in a regular interval is aggregated through a function specified by a Solar operator graph that combines the current status of both types of sensors.

The *Combiner* records the current motion status and pressure status as abstract boolean events, resulting in four possible status combinations that must be mapped to a meeting status. While the two combinations (motion and pressure detected, neither motion nor pressure detected) can be interpreted easily, more intelligence is needed if only one type of sensor has recorded an event while the other failed to do so (e.g. because an empty chair has been moved or a sensor did not respond correctly). The combination function therefore changes the meeting state only after a short hysteresis that allows for new sensor values to be recorded. As the pressure sensor has a reporting interval of 120 seconds [WCK04], longer meetings may occur as a sequence of short meetings if the time span for status updates is too short.

The meeting detector example shows that the Solar system requires much background knowledge about the context to be detected and the characteristics of the sensors being used. The aggregation logic is entirely part of the using application, hence no explicit support for user-provisioned aggregation rules is available. The Solar system in addition does not provide support for degrading of sensor values over time, nor does the integration with a wide-area multimedia communication service benefit from INS distribution infrastructure.

4.1.2 Technology for Enabling Awareness (TEA)

Similar to the Solar project, the project TEA[24] has focused on the publication of abstract context-descriptions to leverage context-aware applications in wide-area networks. A key component of this system is the aggregation of discrete presence *cues* into context information as described by Schmidt et al. in [SAT+99]. The cues result from applying mathematical functions to low-level sensor data. They can have numeric as well as symbolic values, depending on

[23] This example is adapted to the description of the Solar naming model in [Che04] and [ChKo02].

[24] Short for *Technology for Enabling Awareness*. This international research project was funded by the European Union's research program *Esprit* in the subprogram "IT for Mobility" from 5/1998 to 4/2000. Some results are available at `<http://www.teco.edu/tea/>`.

the specific application-scenario. Input parameters are the discrete output values from a single sensor over a given period of time hence providing a basic history log.

The sensors used for collecting environmental data were designed to be carried with the devices they describe. For example, a mobile phone has been equipped with several sensors to detect if it is being moved. [SAT+99] Another prototype system has been connected to a PDA simulating a wearable computer that will be integrated into users' clothes. The demonstrator with its sensor equipment is shown in [LAL01].

An interesting aspect of the TEA system is its support of so-called *cues*, an abstraction layer that allows for pre-processing of raw sensor data before the data is fed into an aggregation function. This way, sensor output may be calibrated, or traces of this sensor from the past may be processed as well. The determination of the actual context then depends on a set of rules that map combinations of cue values to a context name and a value that indicates the certainty of this result. For simplicity, the authors of [SAT+99] use exclusive context, i.e. contexts without any overlaps. The following rule recognizes a mobile phone held in the hand of a user (taken from [SAT+99]):

```
Hand(t) :- standard_deviation(accelX,t) > Dx,
           standard_deviation(accelY,t) > Dy,
           average(light,t)> L.
```

Here, `Dx` and `Dy` denote pre-defined threshold values for the device accelaration, and `L` denotes a pre-defined threshold value for the light conditions. If the cues derived from the current values of the accelaration sensor (`accelX` and `accelY`) and the light sensor (`light`) exceed all of these thresholds, the predicate `Hand` yields `true`.

The given example shows that simple contexts such as the placement of a mobile phone can be defined easily by intuitive Prolog statements. The problem here is the inference of facts that have a significant influence on the particular context, specifically the selection of appropriate values for the constants `Dx`, `Dy` and `L`. To simplify this task, an algorithm has been provided to create a self-organizing *Kohonen map* from a collection of sensor data. While being run with the time-dependent sensor data as its input, the Kohonen algorithm clusters the sensor data in a three dimensional vector space, where specific contexts may be identified.

Being a powerful mechanism for automated definition of contexts, the automated learning algorithm also has some substantial drawbacks:

- First of all, the initial setup of a custom aggregation function requires a previous simulation of activities and behavior characterizing distinct context scenarios to create the Kohonen map. For example, if three distinct contexts *office*, *home* and *travel* with their intuitive semantics are to be defined, sensors have to be set up at least in the office and at home, and the neural network has to be trained accordingly. As described in [SAT+99] and [LAL01], different types of sensors can be used to mimic human perception, specifically to recognize changes of the noise level illumination in the current environment, detect motion, and record device-specific changes like beginning and ending of phone calls, manual input on a computer keyboard etc. During the training phase, the sensor data is collected for off-line analysis. The analysis helps to identify characteristic situations and facilitates context definition using Kohonen maps. In general, this is a tedious task that should be performed preferably by technical personal only.

- Second, the fine-tuning can be difficult for contexts with similar characteristics, e.g. if casual working in a home office shall be treated differently from spending the evening at home. Sometimes, increasing the number of sensors can help solving this problem, but this is typically not an option. Instead, aggregation functions must be defined that evaluate conditions not covered by sensors, e.g. complex user interactions.
- Finally, the inherent fuzziness of the aggregation function, even after extensive training. This non-determinism prohibits the system from being used with sensitive applications like the arming of a security system as these cannot be controlled effectively by heuristics.

Having pointed out possible disadvantages of the self-organizing neural networks, we will finish our discussion of the TEA system with a quick look on the scripting component. It allows for association of actions with the event of context changes. Users can upload scripts into the system which are executed e.g. when the user enters a specific context, leaves that particular context, or when a given context is active for a specific time span. While a context is active, repetitive script execution is also possible, with an interval granularity in the range of milliseconds.

According to [SAT+99], TEA provides a set of pre-defined primitives for creating application scripts. An important aspect of the scripting language are conditional actions that are performed only while in a specific context or if a context transition occurs. The conditions have an associated value that denotes its *certainty*, i.e. there is a certain probability that the context was inferred correctly. These *triggers* provide a flexible mechanism to associate context changes with application-specific actions but do not provide mechanisms for further aggregation or distribution of context information.

In summary, the TEA project offers an interesting approach to automated context determination in a standalone application (i.e. without a mechanism for wide-area distribution of context information). Offline analysis of sensor logs together with a powerful learning algorithm facilitate the identification of relevant contexts that can be distinguished at the application level. To use this mechanism, an extensive learning phase must be conducted, with a simulation of all contexts that are required by the application. Once the relevant contexts have been defined, Prolog rules must be developed to determine these contexts from low-level sensor values. This step requires high technical skills and a good knowledge of the sensors' characteristics.

The TEA project is discussed here for its novel approach to determine the technical parameters that are necessary for identification of distinct contexts. As the TEA system does not provide any support for distribution of context information or multi-step aggregation thereof, we did not consider it as an infrastructure to leverage interactive multimedia communication.

4.1.3 Mobile Interactive Space

A service architecture and communication protocols for the integration of context determination with interactive multimedia communications has been the natural consequence of the research on groupware systems in the early 1990s. With the advent of high-bandwidth mobile networks of the third generation (most notably UMTS in Europe), these efforts have been extended to mobile users as well. In [Kan01], Kanter describes the *Mobile Interactive Space*, a service architecture that targets primarily on mobile devices, but can also be applied to fixed line networks.

The key objective of Kanter's work is to provide a platform that facilitates automatic matching of service requests with corresponding service announcements. For example, a user who

carries his notebook computer through a foreign town may have triggered a search for a wireless LAN hotspot that offers legal Internet access where he can check his email. A service announcement sent by the Internet service provider will tell the connection parameters and possibly the business terms at which the service can be used. Eventually, the notebook computer will automatically establish a connection to the user's home environment through this service provider's infrastructure.

Following the spirit of early groupware systems, Kanter has defined the *Mobile Interactive Space* to be a media space where users as well as their technical equipment and infrastructure are represented by avatars that can establish intention-based ad-hoc communication with each other. To do so, the system architecture provides context determination from sensor data, a proprietary protocol for dissemination of context data, and a mechanism for automated discovery and negotiation of services. In addition, the mobile clients may use the network infrastructure for persistent storage of their context data (called *Active Context Memory*). Thus, the context information is available to other entities while the context owner is offline or roaming around. The dynamic characteristics of an entity within the Mobile Interactive Space are described in the *Mobile Service Knowledge*, a set of rules that reflect the relationships of all objects in the world of this particular media space. The following example shows the definition of a service in the Mobile Service Knowledge (taken from [Kan01]; the author uses Prolog for defining the application logic):

```
relation @ "<http://psi.verkstadt.net/msk/relation>".
relation(access) ::
  type(relation) &
    -> init     :- that <: xsReq &
    <- xsReq    :- certifies.init(this) &
    <- grant(X) :- netIPaddress(X) &
    <- deny(X)  :- closelink(X)
}.
```

Here, the service is identified by an HTTP URI that follows a specific naming convention for the Mobile Service Knowledge (MSK). The relation *access* matches an observed access request `xsReq` with the status change that occurs when this request is handled by a service provider. If the request is granted, a new IP address is assigned to the requesting client. Otherwise, the link will be closed immediately.

Given a set of rules in the Mobile Service Knowledge, application interaction is defined as the mapping of service requests to service offers. A service request is part of the context of a user's application within the Mobile Interactive Space. The announcement of services and the transmission of requests are modeled as context transfer over a specific *Extensible Service Protocol* (XSP) Kanter has defined for this purpose. The protocol that must be implemented by clients and servers of the Mobile Interactive Space enables *discovery* of XSP-enabled entities via IP multicast, *routing of events* within the overlay network provided by XSP-enabled entities, and *service negotiation*.

As the architecture of Mobile Interactive Spaces is targeted primarily on the awareness of resources, no explicit support for user presence exists beyond the user's *physical presence* recorded in the Active Context Memory as information about his mobile device. As the Extensible Service Protocol uses the global SIP infrastructure for naming of entities, message routing, and server location, the Mobile Interactive Space provides native support for user presence based on [RFC3856]. Although this information could be used as context information about a particular XSP entity, the system provides no explicit mechanisms for user-controlled

aggregation of this information beyond the ontology-matching algorithm of the Mobile Service Knowledge.

In summary, our analysis of the Mobile Interactive Spaces shows that Kanter provides an interesting approach to context-aware communication based on the Session Initiation Protocol. As the model explicitly targets at devices and users in mobile networks, the physical presence of users is a key concept of this model. While the projects Solar and TEA have focused on the automatic determination of context, this research effort has investigated a mechanism for ontology-based matching of context information. The matching is based on mobile code that is conveyed through a proprietary protocol, XSP, that provides discovery and negotiation of services in a global SIP-based network.

Looking at the requirements stated in Chapter 3, the Mobile Interactive Spaces reveal a number of disadvantages. First, any entity within a Mobile Interactive Space requires support of the complex Extensible Service Protocol that extends the Session Initiation Protocol to carry context descriptions through the network. Second, no dedicated mechanism for user-provisioned aggregation specifications is provided. Any aggregation must be specified in terms of automated context matching, i.e. based on the system-provided ontology. Usability of this approach suffers especially from the required knowledge of the existing rule set as well as the complex evaluation of Prolog-based specifications.

The high complexity of context description also increases the risk of security breaches as rogue code from the Mobile Service Knowledge may be executed at the local node. Even if the code base was trusted, chances are that a code fragment contains semantic errors that avoid proper matching of context descriptions. In this case, a request might fail even if correct rules were available.

4.1.4 Context-aware Communication Services

A research project to enhance interactive real-time multimedia communication by using context information has been undertaken by the TU Darmstadt, Germany, in parallel to the PASST project this thesis is based upon. According to [Gör05], presence information can be used to control the establishment of interactive communication channels between peers. To do so, the author has extended the *Call Processing Language* (CPL) [RFC2824, RFC3880] by specific operators to retrieve context-specific information from a dedicated storage server and perform routing decisions based on this information. An example for a context-specific CPL primitive is given in Example 4.1.1 (taken from [Gör05]).

Example 4.1.1: An extended CPL primitive for context-specific call routing

```
<context-switch field=2label">
  <context contains="meeting">
    <reject status="reject"/>
  </context>
  <context is="travel">
    <location url="sip:john@example.org"> <proxy/> </location>
  </context>
  <otherwise>
    <sub ref="voicemail"/>
  </otherwise>
</context-switch>
```

In this example, the basic primitives `reject`, `location`, `proxy`, and `sub` are embedded in a context-specific conditional clause, identified by the element `context-switch`. Basically, this clause specifies that incoming calls should be rejected when the context of the called user is related to a meeting. If on travel, the call is proxied to another registered SIP user, e.g. a secretary or an account used with a mobile device (hence implementing a follow-me service). In any other case, the call is redirected to a sub-specification that handles voicemail.

The most interesting aspect of the Context-aware Communication Services is its intrinsic support for aggregation of low-level sensor data using fuzzy mechanisms. Sensors therefore provide not only their current value together with a measurement unit and a maximum sample frequency, but also denote the *dependability* of generated sensor values and their *decay* over time as discussed in Section 3.2.3. Example 4.1.2 depicts an example of a sensor description containing these extended attributes (taken from [Gör05]).

Example 4.1.2: Description of a temperature sensor in PIDF-CE

```
<sensor id="2">
  <auth-class>Work</auth-class>
  <time-date>2004-06-15T12:15:33Z</time-date>
  <decay-function>Exponent</decay-function>
  <value>22</value>
  <unit>Celsius </unit>
  <type>Temperature</type>
  <id>2</id>
  <owner>Room 305</owner>
  <dependability>91%</dependability>
  <frequence>1Hz</frequence>
</sensor>
```

Sensors are network entities with a unique name and a transport address similar to the components of the Solar system discussed in Section 4.1.1. While in service, sensors publish the sampled data via SOAP to a pre-defined aggregation node. The aggregation function is specified by directed graphs that provide operators for filtering of events with the same event type, transformation of events from one type to another, and aggregation of multiple input event to a single output event. The set of pre-defined operators includes filter functions to determine the minimum or maximum or the average of a set of samples, unit conversion functions (e.g. Fahrenheit to Celsius and vice versa), and complex aggregation functions such as Fuzzy Logic, Bayesian networks or uncertain reasoning based on the Dempster-Shafer theory.

Aggregation specifications (graphs) are serialized as XML document using the *Context Aggregation Language* (CALL) defined in [Gör05]. This language facilitates instantiation of Java classes that implement the aggregation operations at any node of the overlay sensor network. In any case, the aggregation result is stored in a *Context Server* that can be queried by applications in order to retrieve an entities current status, e.g. using the `context-lookup` primitive of the enhanced CPL.

Beyond the explicit retrieval of context via CPL, an *in-band* signaling of context information via the Session Initiation Protocol is provided as well. With this mechanism, a caller can explicitly request context information from the callee, or a callee can attach context information to a response without being queried. In [Gör05], Görtz discusses several options to enable this call flow without significant changes to the Session Initiation Protocol. the author's favored option uses the event notification mechanism defined in [RFC3265] together with a PIDF-compliant

representation of the context information. Thus, the integration was possible without proprietary extensions to the Session Initiation Protocol as postulated in Section 3.2.

Context-aware SIP user agents (dubbed *Extended User Agents*, XUA) may also use a new SIP header field `Context` the author has defined for this purpose. A XUA includes this header field containing a reference to a context description in the context server in a provisional response with status code 183 sent for an incoming `INVITE` request. The retrieved context description then is displayed to the calling user who decides if the call should proceed. The calling XUA then sends an `UPDATE` message [RFC3311] to proceed with the call setup or to tear down the session.

The extension of the SIP `INVITE` methods has two major disadvantages: First, only user agents that are aware of this extension (XUAs) can benefit from the context information. Second, the authors change the semantics of the `INVITE` method to provide a deferred call setup, making the called user agent ring only after a subsequent transaction—`UPDATE`—has been processed. Even worse, if intermediaries are involved that do not support this extension (e.g. *session border controllers*), the service may be downgraded to the standardized call setup behavior of SIP. Things may break completely if an application layer gateway such as SIP-aware firewall closes the connection opened by the `INVITE` before an `UPDATE` was encountered. This may happen if the context returned in a 183 response is presented to the calling user who does not react within three minutes (which is the required minimum for timer `C` after a provisional response has been returned in an `INVITE` transaction).

The event-based approach using the fetch-mode of [RFC3265] has the advantage that it is entirely compliant to the Session Initiation Protocol. Its only drawback is that no explicit relationship exists between the subscription operation used to fetch context information and the subsequent `INVITE`. The callee therefore cannot ensure that its context did not change since the sending of its context information in a `NOTIFY` message and the processing of the following call attempt. A caller thus might encounter *false positives*, i.e. rejected calls although the context indicated the callee's availability for communication.

In summary, we believe that the Context-aware Communication Services is a reasonable approach to the integration of context-aware applications with the global SIP infrastructure. For this purpose, extensions have been provided to the Session Initiation Protocol and to the Presence Information Data Format that facilitate the transfer of context information between specific SIP user agents (XUAs).

Context information is generated by aggregation functions processing raw sensor data as well as abstract presence information generated by virtual sensors or additional aggregation functions. Specifically useful is the sensor fusion mechanism that can select from different fuzzy operations for aggregation of data sources. Disadvantages are the limited user control of the aggregation process, as the specification language *CALL* only allows for mathematical operations to be specified. Extensions to this mechanism must be implemented as Java classes that are instantiated when the CALL specification is loaded into the aggregation system.

Finally, the use of SOAP-based Web services for event publication complements the SIP infrastructure that could have been used for this purpose. A centralized context storage is provided as root of the aggregation hierarchy similar to the distribution root of Solar. Given the SIP integration already provided in this architecture, the overhead of the Web service could have been avoided.

4.1.5 Evaluation

In this section, we have shown results from various research projects that have investigated how context information can be used to enhance interactive applications. In particular, we have focused on user-controlled interactive real-time communication, the requirements of which have been stated in Chapter 3. To show the variety of related work, we have discussed the following representative projects:

- Solar

  A technical infrastructure for description of data sources and the efficient distribution of samples in an overlay multicast network,

- Technology Enabling Awareness (TEA)

  A standalone-system that uses advanced mathematical methods for identification of relevant contexts from a large trace of sensor data.

- Mobile Interactive Space

  A service architecture that facilitates adaptive applications, i.e. agents that are configured to react automatically on specific events when in a certain context.

- Context-aware Communication Services

  A framework that integrates context-aware multimedia communication with automatic aggregation of sensor data.

We have found that any of these systems is built around a local set of self-descriptive sensors that deliver the input to the aggregation process. For Solar and Context-aware Communication Services, this data flow is made explicit through a push-based distribution service, while the Mobile Interactive Space uses abstract context descriptions that have no immediate relationship to low-level sensors. Being designed as a standalone-system, the TEA architecture does not have any event distribution mechanism at all, relying on offline processing of collected sensor data.

Data aggregation in these systems is done automatically, controlled either by a formal specification of the aggregation function through iterative learning from a recorded set of samples (as in TEA). The Solar system provides the most basic functionality as contexts must be defined through a set of cue functions and threshold values that define the allowed boundaries of context parameters. A combination with the clustering mechanism of the TEA system would be appreciated to improve the quality of identified threshold values. The Context-aware Communication Services offer an open framework to plug-in any suitable aggregation function implemented as a Java class.

The major differences between the described systems occur at the interface to the user. While the underlying technical infrastructure more or less provides appropriate mechanisms for automatic aggregation of low-level sensor data and its distribution through the network, none of these systems allows for user-provisioned rules to control the aggregation process. For example, if the current user's presence status should be published to the user's friends and family while appearing absent to any other person (e.g. colleagues and business contacts) as denoted by the pseudo-code fragment in Example 4.1.3.

Example 4.1.3: Watcher-dependent status publication

```
if ( (subscriber is family) or (subscriber is friend) )
  publish( status() )
else
  publish( "away" )
```

The statement shown in this example checks for the authorization class of a subscriber before a specific status description is being published. For the Mobile Interactive Space, this condition must be specified as a Prolog rule that fits into the existing ontology of status descriptions, while a user of the Context-aware Communication Services may use the presence-aware Call Processing Language to express this publication policy. In both cases, the aggregation process (or the rule unification of the Mobile Interactive Space, respectively) is not involved in the decision what information to publish. In fact, the only control a user has over the aggregation process of any system discussed in this section is to change the mathematical functions used for context creation.

In addition, these systems try to hide the context determination from the user. While this is a reasonable approach on the level of raw data collected from a large number of sensors, we believe that at a higher abstraction level, the user must be able to control the aggregation of presence information. In particular, if status information is published from a personal presence server within a local environment as motivated in Section 1.2.

4.2 Presence Data Models

As seen in our analysis of systems for context-aware communication in Section 4.1, various data formats exist for representing presence information. To enable the interconnection between distinct presence services and leverage the aggregation of presence information from multiple sources, we have stated in Section 3.1 the requirement of a global presence data model that defines not only a common syntax for status attributes but also a common semantics across distinct presence domains.

This section gives an overview of the relevant data models that are used by existing presence applications in the Internet. We have focused our analysis on wide-area presence services as vendor-specific enterprise systems typically use either one of the formats discussed here or do not support interchange of presence information across system boundaries at all. We begin with a brief review of the proprietary presence format used by the Microsoft Windows Messenger application in Section 4.2.1, as this format is still used by many instant messaging applications in the Internet. Moreover, the format has had significant influence on the IETF standardization of a common presence information data format that is discussed in Section 4.2.2. An alternative to this format is being used by the popular *Extensible Messaging and Presence Protocol* (XMPP) which is discussed in Section 4.2.3. The section ends with an evaluation of the existing data models in Section 4.2.4, including the extension proposals made by Görtz in his thesis on Context-aware Communication Services (see Section 4.1.4).

4.2.1 The Microsoft Windows Messenger Presence Format

One of the most popular protocol stacks for creating applications for SIP-based interpersonal communication is provided as part of the Microsoft Windows operating system. Its functionality includes call setup, modification and termination according to [RFC3261] as well as presence-support based on [RFC3856]. As this protocol stack pre-dates the IETF standardization of PIDF, the presence information is represented in a proprietary format that uses a draft version of PIDF that is feature-compatible with, but not equivalent to, the final document format defined in [RFC3863]. On the network, this proprietary format is identified by the MIME media type `application/xpidf+xml`.

An example document containing user-specific presence information from the Windows Messenger client is given in Example 4.2.1.

Example 4.2.1: Presence document generated by Microsoft Windows Messenger

```
<?xml version="1.0"?>
<!DOCTYPE presence PUBLIC "-//IETF/DTD RFCxxxx XPIDF 1.0//EN" "xpidf.dtd">
<presence>
  <presentity uri="sip:alice@example.net" />
  <atom id="a123">
    <address uri="sip:10.0.0.1:5060;user=ip" priority="0,800000">
      <status status="open" />
      <msnsubstatus substatus="online" />
    </address>
  </atom>
</presence>
```

The document declaration identifies this XML document to be an instance of the DTD defined in [XPIDF], an early draft of the Presence Information Data Format that has been standardized as RFC 3863. Although the presence documents generated by the Windows Messenger protocol stack contain various syntactical errors regarding the referenced DTD, this format is recognized by a large number of presence servers and thus has become a de-facto standard for presence information interchange. In the following, we refer to this format as "XPIDF" as erroneous denoted in the public identifier contained in the document type declaration.

The structure of XPIDF as generated by the Microsoft implementation is rather simple: Every presence document contains exactly one element `presentity` with an attribute `uri` that identifies the principal whose presence information is provided in this document. The presence status is described in an element of type `atom` which must occur only once within a `presentity` element. The `atom` then contains a device contact for the respective presentity, together with the actual status information. Extended status attributes are carried in the attribute `substatus` of the proprietary element `msnsubstatus`, while the element `status` only differentiates between `open` and `close` (following the requirements of [RFC2778]).

Besides its syntactical limitations, the Messenger implementation of the XPIDF format has introduced a widely accepted vocabulary for enhanced presence status descriptions. Table 4.1 shows the symbols used in XPIDF documents generated by the Microsoft Windows Messenger to represent different presence states. From this list, only the presence states "`offline`", "`online`", "`away`" and "`idle`" are generated automatically. The other symbols are published

only if manually selected from the application's user interface. An additional typing indication is sent to a peer when the user has opened a chat window for text communication with that peer. Table 4.1 contains two columns. The first shows the state symbol sent by the Messenger application. In the second column, we have included an interpretation of the symbol's semantics according to our experience using this application. This interpretation is for illustrative purposes only and might slightly differ from the application authors' original intention or other interpretation attempts that may be found in literature.

Property	Description
`offline`	The user's SIP device is not registered.
`online`	The user is available for interpersonal communication.
`outtolunch`	This symbol indicates that the user has gone to lunch and will return to work after a small time.
`away`	Indication that a user has gone and will not be available for some time, although the SIP device is still registered.
`berightback`	The user is having a break and will be back at work soon.
`idle`	The user may be available but has been inactive for some time. This presence state is usually triggered by the screensaver.
`onthephone`	The user is currently in a phone call and thus is not available for audio-visual communication. Other media such as interactive text messaging may be offered.
`busy`	Indicates that the particular user is busy (e.g. in a meeting). In this state, the user is not available for communication.

Table 4.1: Presence states of the Microsoft Messenger application

In summary, XPIDF provides a very basic data model that reflects the notion of a direct one-to-one relationship between users and devices. Here, the user is represented by a single presentity with exactly one communication address or device. Although the underlying format provides support for multiple presentities or atoms within a presence document, the messaging client expects a single communication endpoint to be listed here. The PIDF format, in contrast, allows for more flexible descriptions of the user's communication media.

The fixed vocabulary for the `substatus` attribute provides a simple but expressive set of symbols for enhanced presence status descriptions. Applications that map between XPIDF documents and other presence formats must be able to convert this vocabulary into the vocabulary used by the target system.

4.2.2 Presence Information Data Format (PIDF)

In February 1999, the IETF started to work on an architectural model for transport of instant messages and user-presence information over the Internet. A major objective of the new working group *Instant Messaging and Presence Protocols* (IMPP) was the definition of a common presence data model that can be used for information interchange between different presence protocols.[25]

[25]The rationale behind the IETF's generic framework for instant messaging and presence [RFC2778, RFC2779] was that the members of the IMPP working group could not agree on a single presence protocol. Instead, three

The abstract data model was defined as a set of presence tuples each of which contains a status marker that reflects the associated presentity's presence status. As the model targets on instant messaging, the presence vocabulary must at least contain the status values `open` and `closed`, describing whether or not incoming messages are accepted by the particular presentity (or its *instant inbox*, respectively). Optionally, each presence tuple can have a communication address together with a description of the transport protocol and required parameters to establish a communication channel with that presentity. Additional data may be associated with a presence tuple using format-specific extension mechanisms that are opaque to the abstract data model.

Together with the generic framework and the abstract data model [RFC2778, RFC2779], the IETF has standardized *profiles* that define the concrete syntax and semantics of this model for instant messaging [RFC3860] and presence [RFC3859]. In particular, the *Common Profile for Presence* (CPP) [RFC3859] requires any compliant presence server to support the *Presence Information Data Format* (PIDF) defined in [RFC3863]. PIDF therefore plays a central role for any Internet-scale presence application. In the remainder of this section, we give a brief introduction to the PIDF data model and discuss its apparent limitations.

4.2.2.1 XML Syntax

Having evolved from the early XPIDF draft [XPIDF] (see Section 4.2.1), the Presence Information Data Format also uses XML [XML] as serialization syntax. When included in the body of a presence message, PIDF documents are labeled with the standardized MIME media type `application/pidf+xml`. A PIDF version of the presence document from Example 4.2.1 is shown in Example 4.2.2.

Example 4.2.2: A simple PIDF presence document

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
          entity="sip:alice@example.net">
  <tuple id="a123">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.8">sip:134.102.218.61:5060;user=ip</contact>
  </tuple>
</presence>
```

As seen in this example, the structure of XPIDF and PIDF is almost the same. The PIDF document contains nearly the same information as the corresponding XPIDF version. The only differences are the missing element `msnsubstatus` and the declaration of a default namespace in the start tag of the element `presence`.

Besides the attributes mandated by RFC 2778, presence tuples in PIDF documents can use an extended status vocabulary, prioritized contact addresses, and a timestamp. Every tuple may be commented with an optional free-form text that gives additional information to be interpreted

competing proposals have evolved, and the framework was defined to ensure interoperability of these protocols at the conceptual level. [Alv02, IMPPWG]

by human readers. Example 4.2.3 depicts a document with multiple presence tuples that make use of the optional element types defined in PIDF.

Example 4.2.3: A PIDF document with multiple presence tuples

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
          entity="sip:alice@example.net">

  <tuple id="x12">
    <status>
      <basic>closed</basic>
    </status>
    <contact priority="0.8">sip:134.102.218.61:5060;user=ip</contact>
    <note xml:lang="de">Bis zum 4. Oktober ist mein Büro nicht besetzt.</note>
  </tuple>

  <tuple id="t18">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.8">sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE</contact>
    <note xml:lang="en">You can contact me via our company's SIP-to-Jabber gateway.</note>
    <timestamp>2006-10-01T17:13:12+02:00</timestamp>
  </tuple>

  <tuple id="v36">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.1">sip:alice@voicemail.example.net</contact>
    <note xml:lang="en">If everything else fails, leave me a note on my personal voicemail box</note>
    <timestamp>2006-10-01T16:56:00+02:00</timestamp>
  </tuple>

</presence>
```

Here, each presence tuple represents a distinct communication channel with its dedicated contact address. The first `tuple` element represents an office phone that is currently not available. A descriptive comment is included in an element of type `note` with an XML language tag specifying the natural language used for that comment.

The second and the third presence tuples represent a text messaging client and a voicemail account, respectively. The IM client uses a gateway to another messaging protocol and thus, only the SIP method `MESSAGE` is allowed for that particular contact. An additional `timestamp` element denotes the last change of this tuple's presence status.

As illustrated by the use of `note` elements in Example 4.2.3, the basic PIDF format lacks of a standardized vocabulary for explicit description of presence states. As the contents of the `note` element is not machine-readable, not automatic adaptation of applications is possible. The PIDF specification therefore uses the XML namespace mechanism [XML Names] to enable extension elements to be included in a PIDF document. The use of XML namespace not only ensures syntactic uniqueness of extensions from distinct authors, but also helps distinguishing semantic domains.

To foster interoperability, unknown extensions within a PIDF document must be ignored by an XML processor unless a specific attribute indicates that processing of this extension is required. Several extensions to PIDF have been defined within different IETF working groups

to carry additional information in a presence tuple. Examples for extensions to PIDF are geographic locations of presentities, information on user agent capabilities and preferences, or on technical restrictions like presence of network address translators within the communication part. The following subsection gives a brief overview of current IETF work on these topics.

4.2.2.2 Extensions

As discussed in the previous section, RFC 3863 only defines a core syntax for representation of presence documents with a limited vocabulary for expressing presence states. Concrete applications like multimedia communication systems therefore need to enhance the existing vocabulary with new XML element types in an application-specific namespace. This mechanism enables *rich presence* without having to restrict the description to a pre-defined and probably convoluted set of attributes. XML namespace names thus can be regarded as a modularization technique for presence vocabularies. Any module comes with a specification of the semantics of the elements and attributes contained therein, together with a syntactic schema of the vocabulary's intended use.

[RFC4480] defines a vocabulary to express detailed information on the presence status of persons, devices, and communication services. It shall provide a superset of existing presence systems' vocabularies to facilitate a backwards-compatible mapping between those systems with PIDF. The *Rich Presence Information Data* (RPID) format pertains the descriptive nature of PIDF hence it can be used not only for publishing presence information but also as intermediary format for filtering, aggregation, and scripting. It provides features to enumerate a person's activities, mood, status, and give hints on the logical or social context of that person.

A central aspect of RPID is the explicit distinction between *natural persons*, *devices* and *services*. An RPID document thus can explicitly identify personal activities such as "out to lunch" as well as device states, e.g. "idle". The underlying data model defined in [RFC4479] reflects these three classes of objects as illustrated in Figure 4.1 (taken from [RFC4479]).

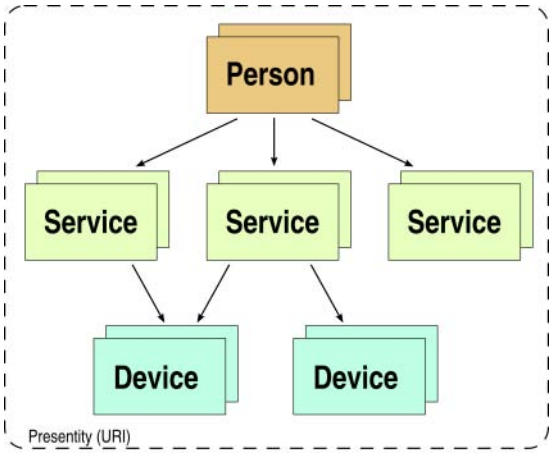

Figure 4.1: The IETF presence data model

The figure depicts a presentity, i.e. a presence-enabled entity that is identified by a unique URI, usually with the URI schema `pres` [RFC3859]. According to [RFC4479], the fundamental object types in this model are the following:

- Person

  An end user who is characterized by abstract presence states such as "busy". The state of the user in this model has an immediate impact on his willingness to communicate.

- Service

  A means for interpersonal communication, such as bidirectional audio communication, or ad-hoc text messaging.

- Device

  Physical endpoints used for communication.

To convey these objects in PIDF documents, the descriptions of persons and devices are included in extension elements using their own XML namespace. The `tuple` element type represents the communication services, i.e. the means of communication offered by a person and implemented by one or more devices. The encoding of these new object types as XML document is straightforward as illustrated by the following examples. For brevity, only parts of the document are shown here. The document as a whole is included for reference in Appendix A.

Example 4.2.4 depicts the basic RPID document structure with a number of presence tuples that represent the communication services offered by the publishing presentity. Within the contents of the `tuple` elements from the PIDF namespace, no RPID-specific elements are used. Instead, the element type `deviceID` defined by the IETF presence data model [RFC4479] is used as a reference to an extended device description. Devices are referenced by a globally unique identifier such as a MAC address or a UUID[26] .

Example 4.2.5 depicts a description of the devices Alice preferably communicates with. The first device is the hardware IP phone in her office that is currently not available because Alice is on business travel. While traveling, a jabber messaging client installed on her notebook computer is online and can be connected via her company's public SIP-to-jabber gateway.[27] The third service describes a voice mailbox that can be called via SIP to leave offline messages.

The device status is modeled using the element type `device` from the namespace of the IETF data model as illustrated in Example 4.2.5. Within a `device` element, the RPID vocabulary is used to provide detailed information on the particular device as well as the *sphere* it is associated with. A sphere can be used e.g. as authorization class to restrict access permissions to a specific group of persons (see Section 4.3.2 for an example authorization policy).

Finally, the personal presence status of Alice is described in an element of type `person`. A comprehensive vocabulary allows for detailed description of the current status and planned activities using the element type `activities`. In Example 4.2.6, the element type `travel` is being used in combination with the attributes `from` and `until` of the `activities` element to specify the duration of Alice's business trip. A `note` element in addition allows for a natural-language description of this particular activity.

The example RPID document resembles the three abstraction layers of the IETF presence data model that was visualized in Figure 4.1. As illustrated in Figure 4.2, the resulting presence description reflects the structure of the underlying data model.

[26] A UUID (*universally unique identifier*) is a 128 bit long identifier that is guaranteed to be globally unique across space and time. An introduction to UUIDs and their use in the URN namespace is given in [RFC4122].

[27] The *jabber* protocol is a predecessor of the *Extensible Messaging and Presence Protocol* (XMPP, see Section 3.3.3.1) and is still widely deployed.

Example 4.2.4: Example RPID document

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
          xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
          xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
          entity="pres:alice@example.net">

  <tuple id="x12">
    <status>
      <basic>closed</basic>
    </status>
    <dm:deviceID>mac:080046c95177</dm:deviceID>
    <contact priority="0.8">sip:134.102.218.61:5060;user=ip</contact>
  </tuple>

  <tuple id="t18">
    <status>
      <basic>open</basic>
    </status>
    <dm:deviceID>urn:uuid:abcdef00-1234-5678-beef-89234abe3f8a</dm:deviceID>
    <contact priority="0.8">sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE</contact>
    <timestamp>2006-10-01T17:13:12+02:00</timestamp>
  </tuple>

  <tuple id="v36">
    <!-- .... status description, capabilities, deviceID, etc. ... -->
    <contact priority="0.1">sip:alice@voicemail.example.net</contact>
  </tuple>

  <!-- ... further RPID elements ... -->
</presence>
```

The authors of RPID have not only developed an elaborate status vocabulary, but also put significant effort in examining typical scenarios of presence-aware communication. One observation they made complements the discussion on appropriateness of selected communication channels we presented in Section 3.1.2: When sensitive data is transmitted to a distant location, it must be secured from being intercepted by unauthorized persons. During transport through the network, this can be done by technical measures such as encryption. At the remote party's endpoint, however, the data must be decrypted to display it to the human user. Now, the information can be intercepted by other people or by technical instruments, depending on the media being used for presentation. For example, audio data could be heard from people passing by or even from a great distance with a sector microphone, textual data can be read at least from people standing right behind the communicating user.

The user's RPID document therefore may contain an abstract description of the privateness level she can assure, e.g. `public` or `private`. In a more elaborate version, this hint could be broken down to specific media level, i.e. indicating secure audio communication (e.g. using a head phone) while video and text communication are insecure and thus deprecated.

With its separation between personal presence, device capabilities and service characteristics, RPID defines three dimensions of presence information. The RPID vocabulary to some degree neglects two of these in favor of an extensive description of personal presence. To describe devices and services of SIP user agents, the language defined in [RFC3840] can be used instead of inventing it from scratch. The PIDF extension [Prescaps] adapts this language to fit the needs of a presence information system and to map the information items to XML element

types or attributes. Examples for the service description vocabulary include but are not limited to indications of supported media types such as audio, video, or text messaging, whether the service is automatic (e.g. a conference server), and an enumeration of SIP messages and URI schemes supported by that service. For presence applications, the names of supported event packages can be listed as well, denoting what type of event subscriptions are handled. In addition to service characteristics, the presence capabilities module provides basic element types to describe device capabilities, most notably to indicate whether a device is mobile or fixed. An example for such capability description is given with the RPID document in Appendix A.

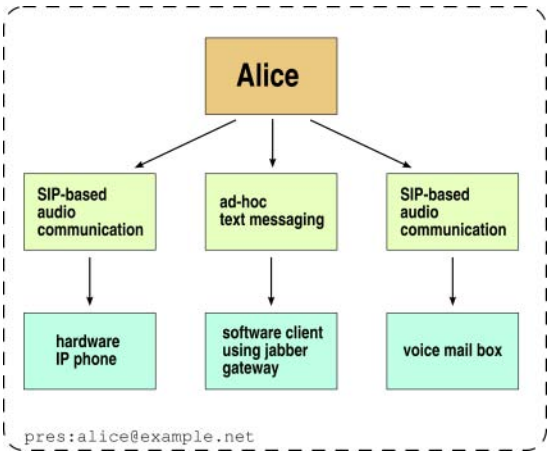

Figure 4.2: Sample instantiation of abstract presence data model

Example 4.2.5: Device description with RPID elements

```
<dm:device id="phone">
  <dm:deviceID>mac:080046c95177</dm:deviceID>
  <rpid:sphere> <rpid:work/> </rpid:sphere>
</dm:device>

<dm:device id="imclient">
  <rpid:user-input last-input="2006-10-01T17:13:07+02:00">idle</rpid:user-input>
  <dm:deviceID>urn:uuid:abcdef00-1234-5678-beef-89234abe3f8a</dm:deviceID>
  <rpid:sphere>
    <rpid:note xml:lang="en">XY company SIP-to-Jabber gateway</rpid:note>
    <rpid:home/>
  </rpid:sphere>
  <rpid:relationship> <rpid:other/> </rpid:relationship>
</dm:device>

<dm:device id="vmbox">
  <dm:deviceID>mac:0013ce6bb427</dm:deviceID>
  <rpid:sphere> <rpid:work/> </rpid:sphere>
</dm:device>
```

Example 4.2.6: Description of Alice's presence status

```
<dm:person id="me">
  <rpid:activities from="2006-09-15T17:00:00+02:00"
                   until="2006-10-04T08:00:00+02:00">
    <rpid:note xml:lang="de">Auf Dienstreise bis zum 4. Oktober.</rpid:note>
    <rpid:travel/>
  </rpid:activities>
</dm:person>
```

4.2.3 Extensible Messaging and Presence Protocol (XMPP)

In Section 3.3.3.1, we have described the *Extensible Messaging and Presence Protocol* (XMPP) as an application-level distribution mechanism for presence information. The protocol's presence model makes no distinction between persons and resources, i.e. a published status value refers to a resource if the `from`-attribute of the `presence` stanza contained a full Jabber ID, and it refers to a person, if no explicit resource has been specified. In its basic form, the XMPP presence vocabulary provides four distinct status values listed in Table 4.2 (taken from [RFC3921]).

Property	Description
`away`	The entity or resource is temporarily away.
`chat`	The entity or resource is actively interested in chatting.
`dnd`	The entity or resource is busy ("Do Not Disturb").
`xa`	The entity or resource is away for an extended period ("Extended Away").

Table 4.2: XMPP presence states

Publication of presence information in XMPP is done with the element type `show` within the contents of a `presence` stanza as shown in Example 4.2.7. Here, a client updates its presence status by sending a `presence` stanza to the server. In addition to the `show` element that contains a single token from the standardized XMPP presence vocabulary shown in Table 4.2, a natural language description of this status value may be included in the `status` element.

Example 4.2.7: Presence status publication in XMPP

```
<presence from="alice@example.net/office">
  <show>away</show>
  <status xml:lang="en">On business travel until October 4.</status>
</presence>
```

The presence data model of XMPP is similar to the basic PIDF data model. In particular, the presence status of XMPP resources can be mapped to PIDF presence tuples and vice versa, while a JID without a resource name corresponds to the entire presentity. An example for a mapping between both data formats is given in Figure 4.3.

As seen in this figure, the contents of the stanza `show` have no representation in PIDF. Instead, a new extension element must be invented to carry this information. Moreover, the

`basic` status required by [RFC2778] have no equivalent in XMPP as this information is carried implicitly by the `type` attribute of the `presence` stanza. If not set, the presentity is available for communication. If not, it must be set to "`unavailable`".

Although the XMPP data model covers a large part of the subset of PIDF that is typically used in existing applications, some important differences exist that impede a smooth interworking between XMPP-based applications and applications that use PIDF. Besides syntactic restrictions such as the lack of a timestamp for XMPP presence values and different ranges for priority values as documented in [RFC3922], the unification of presentities and resources in XMPP prohibits a concise definition of presence aggregation. While PIDF allows for atomic updates of presence descriptions from multiple sources using a collection of presence tuples in a single document, XMPP requires a single transaction for every resource that contributes to the aggregation process.

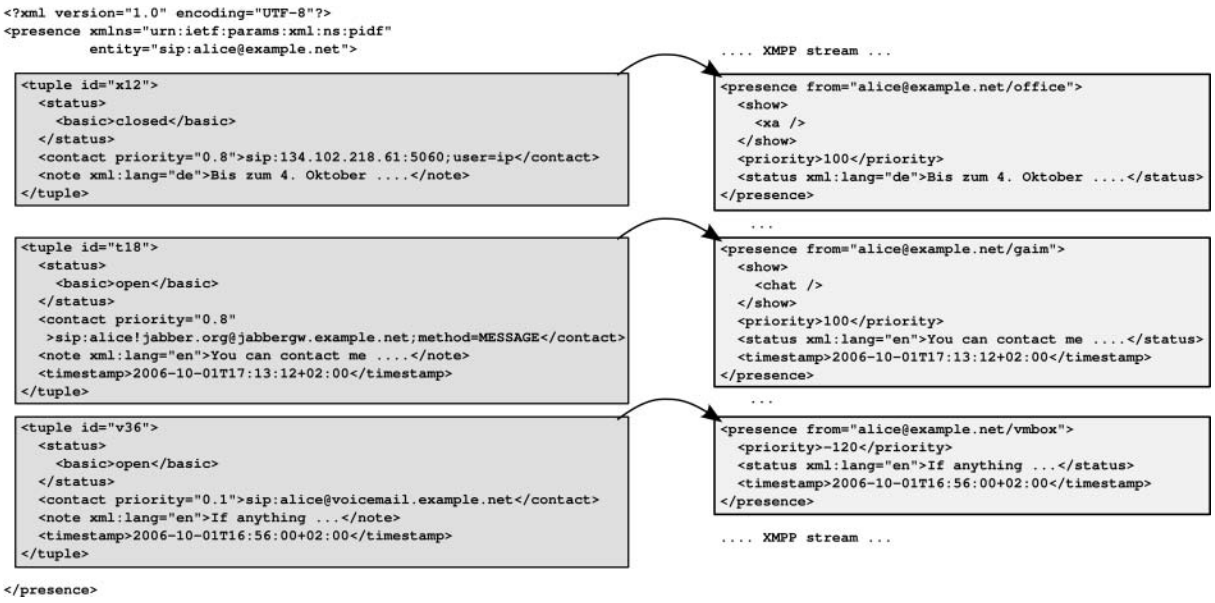

Figure 4.3: Mapping from PIDF to XMPP

Having noted these shortcomings, the Jabber community who is driving the standardization of XMPP has proposed a mechanism to convey PIDF documents within a `presence` stanza. To do so, the inherent namespace-based extension mechanism of XMPP is used as illustrated in Example 4.2.8.

Since there is little use for native XMPP clients to implement this format in addition to the lightweight presence format of the core XMPP protocol, we anticipate that tunneling will be used primarily for transit of PIDF documents through the XMPP network. One reason for this certainly is the improved publication mechanism that has been proposed for XMPP in the *Jabber Enhancement Proposal 0060: Publish-Subscribe* [JEP-0060]. This extension provides a powerful and flexible protocol for publication of XML content[28] to subscribed watchers, enabling filtering and syndication of content, and selective delivery based on pre-defined authorization classes. An extension to this mechanism has been proposed in [JEP-0163] specifically

[28]The extension targets primarily on dissemination of Atom documents [RFC4287], but can be used with any other XML data as well.

for publishing presence information, with a set of domain-specific vocabularies for specifying geographic locations, user mood and activities, music tunes currently heard, etc. [JEP-0080, JEP-0107, JEP-0108, JEP-0118].

Example 4.2.8: Encapsulation of PIDF in XMPP

```
  <show>away</show>
  <status>On business travel until October 4.</status>
  <presence xmlns="urn:ietf:params:xml:ns:pidf"
            entity="sip:alice@example.net">

    <tuple id="x12">
      <status>
        <basic>closed</basic>
      </status>
      <contact priority="0.8">sip:134.102.218.61:5060;user=ip</contact>
      <note xml:lang="de">Bis zum 4. Oktober ist mein Büro nicht besetzt.</note>
    </tuple>
  </presence>
</presence>
```

In summary, we believe that the presence data model of the core XMPP protocol is nearly equivalent to the limited data model of PIDF. XMPP's limitations regarding presence (lack of atomic status updates for multiple channels, no strict distinction between presentity and resources, implicit basic status) are outweighed by several extensions proposals to the core protocol, especially the flexible *Publish-Subscribe* mechanism for push-based content dissemination.

Currently, XMPP and SIP form two separate protocol domains that are barely interconnected. While numerous services exist to exchange instant messages and basic presence information between both domains, the interconnection of session signaling is far more complex and thus not widely available. Therefore, we focus on SIP as a robust and powerful protocol for call signaling as well as distribution of presence information. In fact, we envision an increase of SIP-based presence applications as the importance of presence information for efficient communication grows.

4.2.4 Evaluation

We have presented in this section standard formats for representation of presence information, together with their underlying data models. PIDF, the most important data format for presence at Internet-scale has been standardized by the IETF as [RFC3863], which complies to the basic framework for instant messaging and presence [RFC2778, RFC2779]. As the PIDF format targets on interoperability of presence protocols, it defines the core document structure, consisting of *presence tuples* that can be extended by other standards to carry more detailed information. Examples for PIDF extensions are the *Context-enhanced PIDF* (PIDF-CE) proposed in [Gör05] and the *Rich Presence Extensions* (RPID) [RFC4480]. Even the proprietary XPIDF format of the Microsoft Windows Messenger provides an additional element `msnsubstatus` that extends the core vocabulary of PIDF.

The large number of existing extensions—standardized or proprietary—not only indicates that PIDF is an appropriate data format for representing presence information, but also that the information carried in presence documents is always *domain specific*. Consequently, presence

vocabularies cannot be said to be feature complete as there will always be specific domains that are not covered by existing vocabularies.

This observation has implications on the data model as well as the processing model for presence documents. In Section 3.1 we have stated that a concise data model for user presence must support *communication channels* as first-class object types. The standardized data model [RFC4479] partially meets this requirement as it defines presence tuples to describe *communication services*, while additional extension elements describe a person and a set of available devices as communication endpoints. While these extensions can facilitate creation of applications that act automatically on receipt of presence information, it also introduces new ambiguities, as the presence information now must be split over three types of objects where formerly only presence tuples have been used.

The potential ambiguities introduced with this more complex data model can be avoided by proper definition of domain-specific vocabularies with a clear semantics. But as [RFC4479] also admits, the composition of presence documents from multiple source may lead to conflicts that cannot be resolved automatically. In this case, only a human user would be able to create a coherent view on the given status information.

Another issue that is not addressed by existing presence data models is the inherent inexactness of presence values. As seen in Section 4.1, various research projects have investigated mechanisms to aggregate inexact or degrading presence information. In particular, Görtz has proposed a PIDF extension to carry low-level sensor data with a description of the value's degradation over time. Applied to other presence attributes as well, this mechanism could be used to trigger the re-calculation of presence information as described in Section 3.2.

Given that only a small number of clients already supports the IETF data model for presence defined in [RFC4479], we envision that the relevant presence attributes must be provided in the contents of `tuple` elements that describe the communication channels offered by the presentity. The additional element types `person` and `device` may be used in the input of the aggregation process but should not appear in the created output document.

4.3 Specification Languages for Data Aggregation

In Section 4.1, we have argued that sensor fusion based on statistical functions and mathematical operators is a reasonable technique for aggregation of low-level sensor data, and automated learning facilitates the identification of relevant presence zones at an abstract level. While these techniques are transparent to the user, it is necessary to provide powerful control functions that enable users to actively manage their presence status. With TEA (Section 4.1.2), Context-aware Communication Services (Section 4.1.4) and Mobile Interactive Spaces (Section 4.1.3), we have already discussed applications that can be controlled by user-specific rules. In this section, we discuss specification languages to control the aggregation service, i.e. user-specific functions are not restricted to the communicating end-systems but are also available at data-processing entities within the network as illustrated in Figure 4.4.

In this figure, two communicating clients are shown, together with the path of a status publication through the Internet. The distinct trust domains of the publishing user `alice@example.org` and the subscribed watcher `john@other.com` are indicated by shaded areas, containing the network nodes therein. In our approach, users are supposed to place aggregation specifications at any network node within their trust domain. While user-specific data aggregation and

context-adaptation traditionally occur at the communication endpoints only (denoted by dashed circles), we propose in this thesis a processing model for presence aggregation that involves intermediary nodes as well. In particular, any SIP server within the trust domain of the publisher of presence information (`alice`, in this case), can host aggregation rules provisioned by the user whose personal status is being notified.

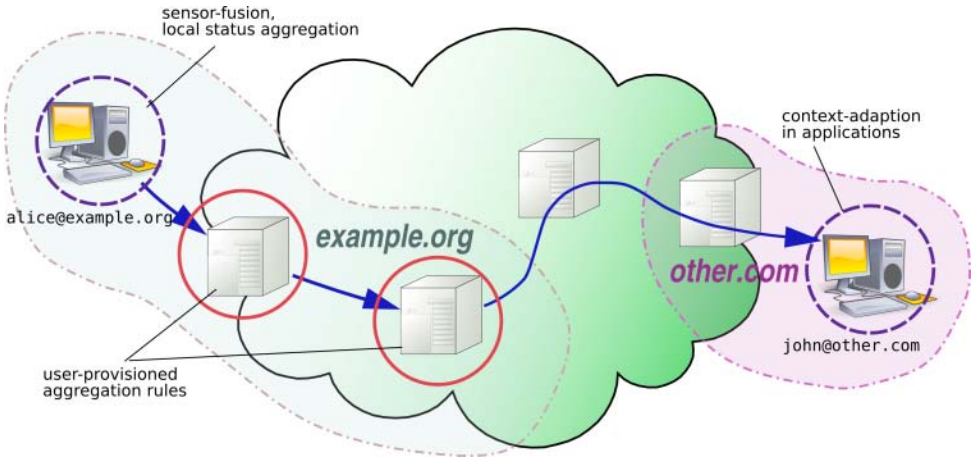

Figure 4.4: Different placements for user-provisioned aggregation rules

The following two subsections discuss existing mechanisms and languages for information processing at intermediaries within a network. As this is a new idea in the area of user presence, our analysis has focused primarily on document filtering and content syndication in the World Wide Web. We discuss our findings on this topic in Section 4.3.1. Section 4.3.2 then summarizes our analysis of ongoing work in the IETF to standardize a policy framework that provides a set of simple filtering rules to remove sensitive content from presence documents based on specific authorization classes. The section ends with an evaluation of existing languages for high-level data aggregation in Section 4.3.3.

4.3.1 Document Transformation

The manipulation of hierarchically structured documents such as SGML[29] or XML has been an important task to facilitate information retrieval, online publishing and document interchange since the early dates of automated data processing. [BOK+02] For publication of XML documents—online or in print—the rule-based *XSL Transformations* (XSLT) have become a standard technique for modifying the contents and the structure of well-formed input documents. The transformation language has been defined by the *World Wide Web Consortium* (W3C) in [XSLT] as part of the *Extensible Stylesheet Language* [XSL] used to specify the formatting of generated output from XML documents.

A transformation in XSLT is defined as a set of *template rules* that are applied to a tree-based representation of the structured input document. Each rule contains a pattern that specifies a set

[29]The *Standard Generalized Markup Language* (SGML) is a predecessor of XML, defined in ISO 8879:1986. A well-known SGML application (i.e., a document type together with its semantics) is the *Hypertext Markup Language* (HTML), see <`http://www.w3.org/TR/html4`>.

of nodes within the input tree, and a template that replaces these input nodes when instantiated. The instantiation of template rules is controlled by specific instructions within each template. Example 4.3.1 shows a simple template rule to map a proprietary XPIDF element to a corresponding element from the PIDF vocabulary.[30]

Example 4.3.1: Sample XSLT template rule

```
<xsl:template match="presence">
  <pidf:presence entity="{presentity/@uri}">
    <xsl:apply-templates select="child::*" />
  </pidf:presence>
</xsl:template>
```

In this example, the namespace prefix `xsl` is bound to the namespace of XSLT, `http://www.w3.org/1999/XSL/Transform`, and `pidf` is bound to the PIDF namespace, `urn:ietf:params:xml:ns:pidf`. XPIDF does not use any specific namespace, hence do XPIDF elements appear without a namespace prefix.

The attribute `match` of the element `template` contains the pattern that specifies the input nodes to which this rule applies. In this case, all nodes of type `presence` should be transformed by this rule. To do so, these nodes must be specifically selected by another template rule as input for the next transformation step, and the rule evaluation must be triggered. Usually, this is done automatically by a default rule that traverses the input tree starting at the root node. For each node, the rule instantiation process is triggered recursively, with the current node's child nodes as input set. At some point, this process has a `presence` element as the current node, and hence the XSLT processor instantiates the template contained in this particular rule as visualized in Figure 4.5.

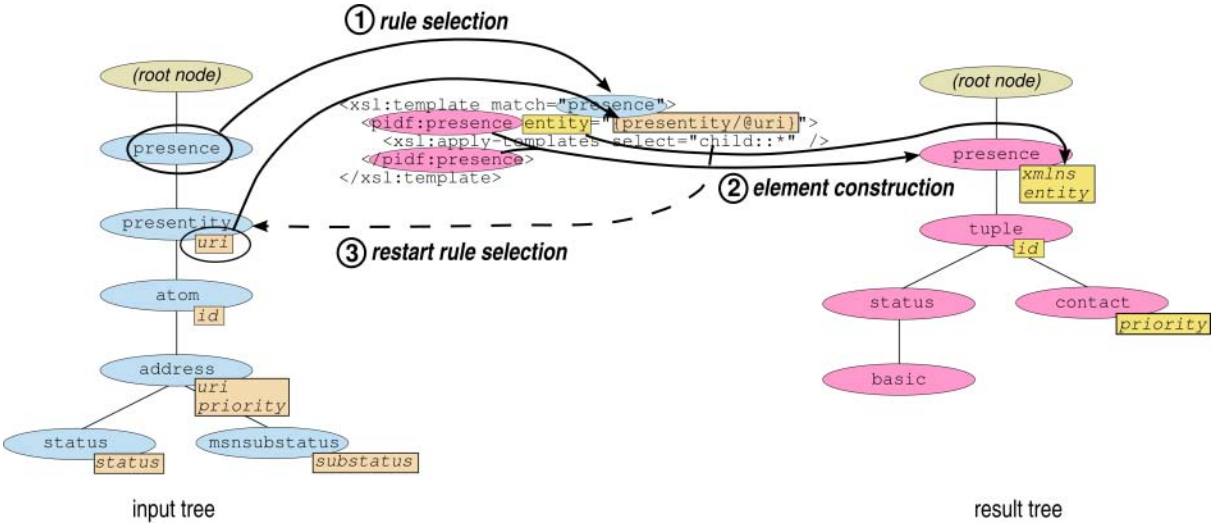

Figure 4.5: Instantiation of a template rule

[30]This example is taken from a transformation specification for the input format converter we describe in Section 7.2.2.2. The entire specification is included in Appendix E.

The example template is constituted by elements from the XSLT namespace as well as elements from a foreign namespace, i.e. PIDF. In Figure 4.5, the distinct namespaces are visualized by different colors for element nodes (shown as ellipses) and attributes (shown as rectangles). The PIDF elements constitute an XML fragment that is inserted at the current position into the constructed result tree. In other words, the currently processed XPIDF element of type `presence` will be replaced by a PIDF element of type `presence` (including an appropriate namespace declaration) that carries an attribute `entity`. The attribute's value is made up of the result of the expression "`{presentity/@uri}`" which evaluates to the contents of the attribute `uri` of the child node `presentity` of the currently processed tree node. In an XPIDF document, this attribute contains the URI of the presentity whose status is published.

The element `apply-templates` then triggers the recursive template evaluation process. Its attribute `select` contains an XPath (*XML Path Language*) [XPath] expression that selects the tree nodes to be used as input for template evaluation. In this case, all child nodes of the current node are selected.

Among the primitives provided by the XSLT language are conditional branches and recursive template calls, hence making the language Turing-complete. Authors of transformations rules thus must be careful not to write functions that loop infinitely when using enhanced features of XSLT.

In summary, XSLT is a flexible and powerful tool for specifying modifications of structured documents. The rule-based language uses XPath to address nodes within a tree structure that represents the syntactical components of an XML input document. The selection of nodes to be transformed is controlled by the currently evaluated rule, together with the modifications of the current node. Thus, the entire transformation process is controlled by the rules contained in a transformation specification.

A disadvantage of XSLT regarding presence aggregation is the missing support for multiple input documents. Even though a reference to the contents other documents is possible within a transformation rule, it is not possible to include these documents as input to the current transformation process. A new paradigm that is currently used primarily for interactive Web pages comprises a combination of document transformation using XSLT and imperative scripting languages. Typical transformation tasks such as the conversion of documents from one format to another (e.g. from XPIDF to PIDF) are performed using XSLT, while the dynamical part is covered by user-provided scripts. Most of these *Rich Internet Applications* today use the *JavaScript* programming language that is embedded in most modern Web browsers. When loading a Web page, the browser's runtime environment creates a tree representation of that document, with a set of JavaScript functions to navigate through the tree and modify its contents. The server thus provides static documents with mobile code to make Web pages more usable, anticipating the requirements for presence aggregation stated in Section 3.1.

4.3.2 Policy-based Filters

When combining information from multiple presence sources into a single, coherent view on the user's presence status, most of the presence tuples provided in the input document will be removed from the output set as it provides information that is redundant or intended for private use only. In its current standardization work, the IETF is therefore developing a rule-based language to filter presence documents according to user-provisioned authorization rules. Although

the language is not intended for presence aggregation, it is at least suited as a specification language for simple transformations on presence documents.

The rule-based filtering language is defined in [Auth] which is based on a generic policy framework for expressing privacy preferences [Policy]. Here, policies are sets of rules, each of which consists of the following parts:

- *Conditions*

  Boolean expressions that determine a rule's applicability. The contents of a rule are processed only if all conditions given for that rule evaluate to true for a specific watcher.

  Examples for a condition include an identity check to filter authenticated watchers that match a particular URI or a user from a specific domain.

- *Actions*

  Actions can be used to block watchers from presence status updates. If access to the presence information is granted, the transformation specify what data is actually sent to that particular watcher.

- *Transformations*

  The core part of the filtering language. The set of applicable rules for specific subscription constitute a filter that is applied to a status update notification before transmission.

The transformation primitives defined in [Auth] allow for selection and of items from an incoming PIDF document that uses the data model of [RFC4479]. In particular, the proposal defines primitives to provide specific object types in the generated output set, e.g. for inclusion of `person` information, `service` or `device` information or specific extension elements such as the `user-input`. Example 4.3.2 depicts an example for a presence filtering policy.

The example policy contains two filtering rules that specify which information is published to specific watcher groups. The first rule is evaluated for any authenticated watcher, and provides only presence information of Alice's office phone and her voice mailbox. The details of her private instant messaging account are revealed only to her friends and her family.

The granularity of filtering rules is limited to the standardized vocabulary of the authorization framework. [Auth] defines a set of XML element types with a prefix "`provide-`" that identifies the syntactic components of a presence document that shall be included in the generated output set, if contained in a PIDF input document. As a consequence, the rule author must know that the service description contained in the element `tuple` will contain a reference to a `device` element that must be included in the output set as well. The rule therefore not only contains an element of type `provide-services` but also an element of type `provide-devices` with a hard-coded identifier for the office phone, or the element type `all-devices` that copies all device descriptions contained in the input document into the output set.

An additional filter for information on devices, services and persons can be specified using the element type `class` that refers to the RPID element with the same name. The token contained in `class` element denotes the watcher authorization class that is allowed to receive the information it is associated with. A watcher's authorization class is denoted by the attribute `value` of the element `sphere` of the rule's guarding condition. This sphere-based authorization model resembles the access model of the proposed Publish-Subscribe extension of XMPP,

with the only difference that XMPP defines a fixed set of five access models, each of which has certain permissions.

The brief example given here already demonstrates the limitations of the policy framework regarding presence composition. As the processing model targets only on filtering of information based on the syntax of input documents, no semantic aggregation is possible. In addition, the pre-defined set of transformations is rather limited to avoid complex policy decisions. As a result, the policy framework is a useful basis for filtering sensitive data from presence documents, but not sufficient for user-specific aggregation of their personal presence information.

Example 4.3.2: Example policy with custom filtering rules

```
<?xml version="1.0" encoding="UTF-8"?>
<cr:ruleset xmlns="urn:ietf:params:xml:ns:pres-rules"
            xmlns:pr="urn:ietf:params:xml:ns:pres-rules"
            xmlns:cr="urn:ietf:params:xml:ns:common-policy">

  <cr:rule id="public">
   <cr:conditions>
    <cr:identity> <cr:many /> </cr:identity>
   </cr:conditions>
   <cr:actions> <pr:sub-handling>allow</pr:sub-handling> </cr:actions>
   <cr:transformations>
    <pr:provide-services>
      <pr:occurence-id>x12<pr:occurence-id>
    </pr:provide-services>
  </cr:rule>

  <cr:rule id="detail">
   <cr:conditions>
    <cr:sphere value="friends family"/>
    <cr:identity> <cr:many domain="example.net"> </cr:identity>
   </cr:conditions>
   <cr:actions> <pr:sub-handling>allow</pr:sub-handling> </cr:actions>
   <cr:transformations>
    <pr:provide-devices>
      <pr:class>family</pr:class>
      <pr:class>friends</pr:class>
    </pr:provide-devices>
    <pr:provide-services>
      <pr:class>family</pr:class>
      <pr:class>friends</pr:class>
    </pr:provide-services>
    <pr:provide-persons>
      <pr:all-persons/>
    </pr:provide-persons>
   </cr:transformations>
  </cr:rule>

</cr:ruleset>
```

4.3.3 Evaluation

In the previous sections, different paradigms for the definition of a presence aggregation language have been discussed. Existing approaches favor rule-based transformations based on the syntax of structured documents, with only little or no support for operations that refer to the semantics of the document contents. The IETF has standardized a domain-specific language for filtering of presence attributes based on the semantics of standardized components.

Although the generic approach of the underlying policy framework would allow for more complex transformations, no proposals yet exist for a presence aggregation language that uses this mechanism.

Having analyzed a number of languages for manipulating XML documents, we found that a tree-based notation of the element structure has significant advantages over other existing approaches, especially those that use regular expressions to address the document's syntactical components. Modern applications often use document transformation in conjunction with scripting, and thus leverage interactive content in otherwise static documents. The language must provide an application programming interface that enables users to author aggregation specifications based on the semantic model of the presence service, i.e. specifications should not require knowledge of the concrete syntactic representation of the language's abstract data model.

As a consequence, XSLT should be used primarily to facilitate the conversion of document formats. Thus, a presence aggregation server can support various input formats and possibly generate output formats that are not covered by the aggregation engine's serialization function. The advantage of syntax-based transformations is the simple adaptation of transformation specifications to changed or newly-defined document formats, as the language definition is not affected. In contrast, domain-specific languages provide a semantic model of the underlying document and thus are more robust against erroneous rules that break the output document's structure. Although this could be enforced for syntax-based document manipulation as well—e.g. by validating the generated output against a given XML schema—it would be difficult to handle errors that have been detected by this validation step as no direct interaction with the author of the transformation specification occurs.

Unlike syntax-based transformations, domain-specific languages for aggregation of documents define a fixed vocabulary that represents the relevant semantical knowledge about the respective area of application. New features of the underlying document format such as PIDF extensions then must be added to the language's vocabulary before they can be used in an aggregation specification. The filtering language discussed in Section 4.3.2 gives an example, as only those elements from the source document are added to the result that can be named explicitly by an element in the filter rule.

4.4 Summary

This chapter has discussed research projects and standardization efforts that are related to the topic of this thesis. First, we have analyzed the results of existing projects in the area of context-aware applications, with a specific focus on the techniques used for aggregation of sensor data and dissemination of the resulting context information. We found that sensor-fusion is an accepted mechanism for the synthesis of context information from low-level sensor data within a local environment. Data distribution in local networks is typically based on content-addressable networks where sets of attribute/value pairs are used to subscribe for a specific data stream, while wide-area transport of abstract presence information (usually at a lower rate) is often based on the generic event notification mechanism of the Session Initiation Protocol.

Related to the aggregation and wide-area distribution of presence information is the question of the data model and language paradigm to be discussed. The existing projects we have studied use various techniques to control the aggregation process, but none of them has its primary focus

on the general usability of this language. Instead, specifications are authored by specialists with technical skills that are beyond the qualification of average users. As the main goal of context-aware applications is the automatic adaptation of an application's behavior according to inferred contexts, no specific requirement exists for user-provisioned aggregation rules.

As a result, no dedicated languages for presence aggregation at an intermediary in the network yet exist. Our survey therefore was expanded to investigate information systems that are similar to presence. In the World Wide Web, XML documents are typically transformed using XSLT at the origin server of the particular document. XSLT enables syntax-based processing of presence documents, while semantics-based operations are typically performed by mobile code that is distributed to the client as part of the requested page. The scripting language used for Web page manipulation provides a set of functions for dealing with semantic components from HTML documents. A similar language for presence aggregation hence must build upon a presence-specific data model and define relevant functions to access or modify objects from the generic data model. Our design of this data model and language is discussed in the following chapters.

Chapter 5

An Architecture for a Distributed Presence Aggregation Service

The evaluation of existing systems for aggregation of presence data in the previous chapter has revealed that there has been little progress in providing users with the facility to control the *entire* aggregation process since the early days of CSCW systems. We have found that sensor-fusion and context-determination are useful mechanisms especially in local presence environments. The information inferred from low-level sensors can be distributed over the Internet by flexible presence protocols such as the Session Initiation Protocol (SIP).

Although the designers of SIP-based presence have been aware of the fact that presence data published by multiple sources for a single presentity must be combined into a consistent view on the presentity's presence status, the resolution of ambiguities has been left to the watcher. According to the results of our requirements analysis in Chapter 3, we believe that only the publishing user should specify how conflicts between status notifications from distinct presence zones should be resolved, and which details of this information should be presented to a specific watcher.

As a consequence we have developed a framework for aggregation of presence information that is based on user-specific aggregation specifications. In the architecture we present in this chapter, any node along the publication path of presence information for a specific user can evaluate a user-provisioned set of aggregation rules. This aggregation specification describes how the input documents should be processed to generate an output document describing the user's presence status. Our major contribution is the creation of a presence aggregation service that applies to all steps of the presence distribution process based on the existing SIP architecture we have described in Section 3.3.2.

The SIP protocol was selected as wide-area distribution infrastructure because it is accompanied by a large number of extensions for status subscription and event notification as well as a growing set of application-specific presence vocabularies, dubbed "event packages". Being a widely deployed protocol for signaling of IP telephony calls and management of multimedia conferences, SIP provides a powerful basis for presence-aware communication services including the concept of composing presence documents from multiple sources. Thus, the use of SIP not only facilitates integration with existing communication systems but also ensures backwards-compatibility with deployed clients. The latter especially leverages acceptance of our system by end-users as these are not required to give up communication clients they are accustomed to.

This chapter begins with a high-level overview of the overall system architecture covering the dissemination of presence documents using the Session Initiation Protocol (Section 5.1). In Section 5.2 we discuss the concrete processing model for presence aggregation that we have developed during our research work. The specific requirements stated in Section 3.2 also advocate an extended data format to associate presence values with meta-data describing the exactness of these values over time. Our standards-compliant extensions to the presence information data format (PIDF) are detailed in Section 5.3, followed by an overview of the design we have chosen for the implementation of the local presence environment where most of the presence information originates (Section 5.4). A possible mechanism for authorization of watchers to access specific parts of aggregated presence documents is discussed in Section 5.5. The final section then summarizes the major results of this chapter in Section 5.6.

5.1 Overview

An important aspect of our design is the integration with the SIP architecture. Any component of our system that is part of a global presence infrastructure must comply to official IETF standards, especially the SIP family of standards as described in Section 3.3.2 and the presence information data format (PIDF) we have discussed in Section 4.2.2. Both, the use of a common syntax and a core vocabulary for presence documents as well as a standardized protocol to convey this data ensure compatibility with existing clients and facilitate interoperation with future developments.

The interconnection interface of one or more local presence environments in our architecture with the global SIP event notification service is depicted in Figure 5.1. In this picture, every presence server marked with a dashed circle can host a *presence aggregation module* (PAM) for semantic aggregation of incoming presence documents. Users can upload aggregation specifications on these servers to make them process incoming presence documents in a specific way, e.g. to facilitate call completion as discussed in Section 2.6.

The figure recapitulates the use case where a principal is in a phone call while another user tries to contact the principal. The call completion is handled by the participating application using the information that has been aggregated from the local presence zones as motivated earlier. To do so, the principal's central state agent creates a global view on the several presence zones of the principal. Figure 5.2 shows an example with three distinct presence zones, *home*, *travel*, and *office*. A dedicated presence server for every zone collects low-level data from computers, phones, network nodes and possibly several other sensors and generates an abstract description reflecting the principal's status as seen in this local environment.

In the given example, the user Alice is talking with her mobile phone to some other user, Bob, while on travel. The presence server governing that presence zone therefore publishes a status description indicating that Alice is currently in a phone call, while the presence servers for the zones *home* and *office* publish status descriptions indicating that she is not available for communication. The receiving presence server representing Alice's presence status in the public Internet must generate a consistent view on these status descriptions from the user's three presence zones according to her rules for resolving inconsistent presence states. In this case, Alice might have installed a rule that generates busy-notifications whenever the mobile phone is in a call while she is neither at home nor in her office.

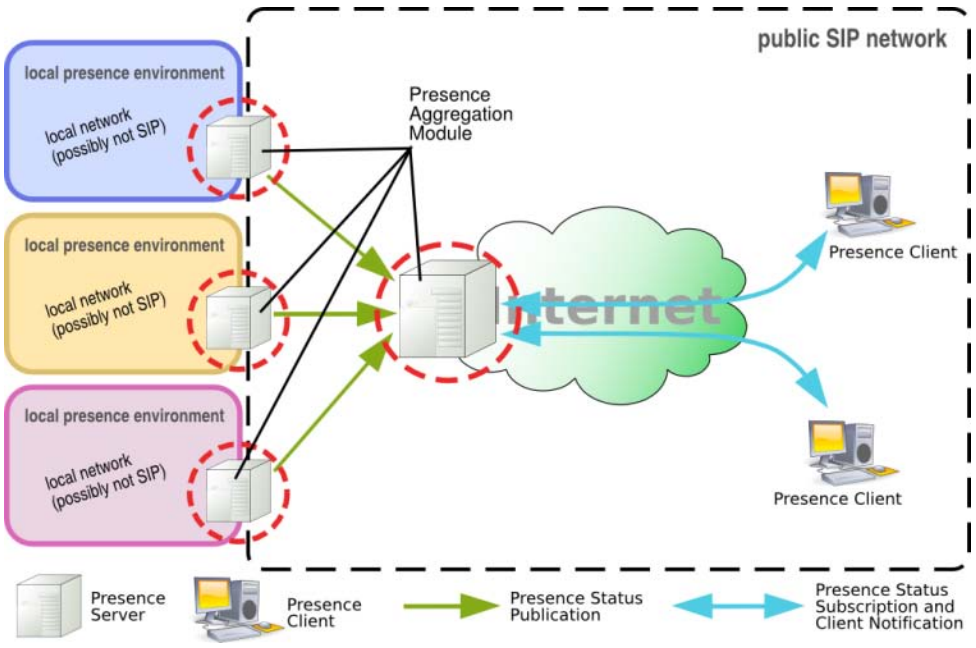

Figure 5.1: Presence Aggregation in a global SIP network

When the third user—Jill—tries to call Alice, the call attempt fails with a SIP error code `486` indicating that the called user is busy and thus cannot answer the call at this time. Now, Jill can subscribe to Alice's presence status to be notified when the ongoing call has been terminated and Alice is available for another phone call. According to [RFC3265], the new subscription will be acknowledged by a notification message indicating the subscribed user's current presence status—`busy`—as shown in the picture.

After Bob and Alice have finished talking, the call is terminated and the presence service indicates that Alice again is available for communication (possibly after a short detention interval that Alice has configured to avoid being called immediately after the previous call has ended) as illustrated in Figure 5.3. Jill's phone then automatically tries to establish a communication channel until Alice either accepts or declines the call attempt.

As most phones today only have limited support for presence—or none at all—the call completion function may be implemented by a telephony service provider as well. A possible setup scenario using the Jill's trusted outbound proxy is shown in Figure 5.4. Here, the arrows denote complete SIP transactions, i.e. pairs of requests and corresponding final responses and the subsequent `ACK` if the initiating request was an `INVITE`.

In the first step, Jill sends an `INVITE` message to her outbound proxy that has been configured to perform automatic call completion if necessary.[31] According to the rules of [RFC3261], the proxy passes the best response for the outgoing `INVITE` request back to the initiating user, Jill. Doing so, the user is aware of the fact that the call attempt has failed at this time and the

[31] There might be several ways to configure the proxy, e.g. manually via a Web interface, or through a to-be-standardized header field in the request indicating the preferences for the current call attempt. Currently, there is no standardized way to request automatic call completion within a request or in the set of registered caller preferences.

proxy will try to establish the call automatically. The proxy therefore subscribes to the called user's presence status as shown in the previous example. When the proxy receives a `NOTIFY` request indicating that the called user is available for communication, it tries to setup a new call between Jill and the called user. This is done according to [RFC3725] that describes best practices for controlling calls when acting as intermediate proxy.

We have selected the call completion scenario here to illustrate the benefit of a smooth integration with the common SIP architecture described in Section 3.3.2. As stated before, the use of standardized communication protocol facilitates creation of new services based on the existing service infrastructure. On the protocol level, the integration with SIP is straightforward as no specific adaptations of the protocol are required. The actual implementation of our SIP-based presence aggregation servers denoted by the dashed circles in Figure 5.1 is discussed in Chapter 7.

The strict separation of the distribution mechanism from the aggregation logic proposed in this chapter also leverages the modular implementation of architectural components as shown in Section 7.2. Specifically, the SIP server we have implemented acts as proxy if no extension modules are present. Adding a module for subscription handling yields a state agent as defined in [RFC3856]. This state agent then can be associated with an aggregation module that controls the composition of incoming presence documents, and a web-interface for managing user-specific aggregation scripts. Due to a role-based addressing scheme provided by the local communication platform, no explicit control messages between the presence server and its associated processing logic is needed.

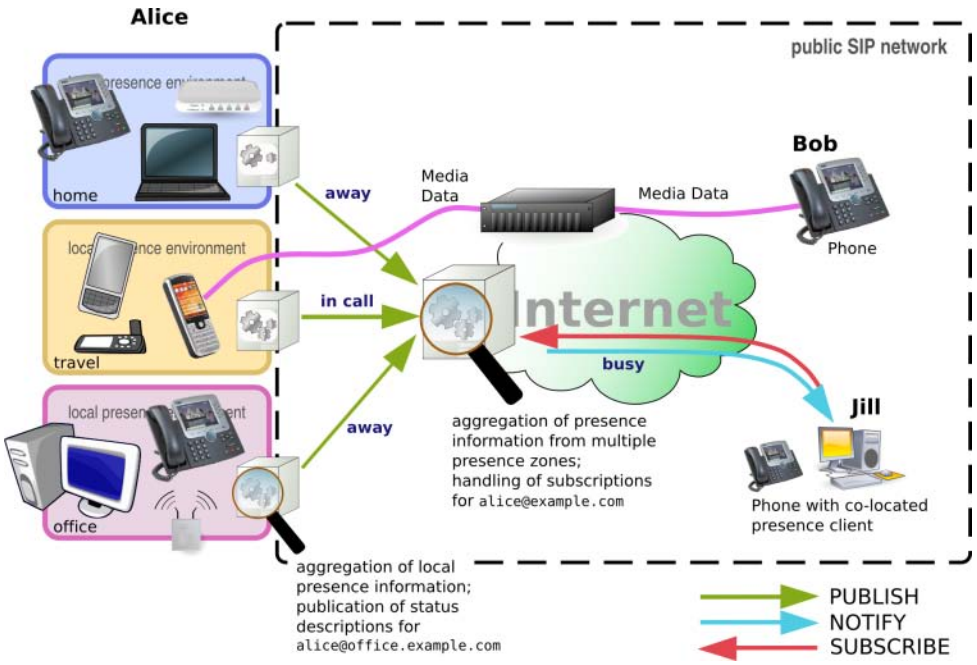

Figure 5.2: Subscribing the presence status of a busy user for automatic call completion

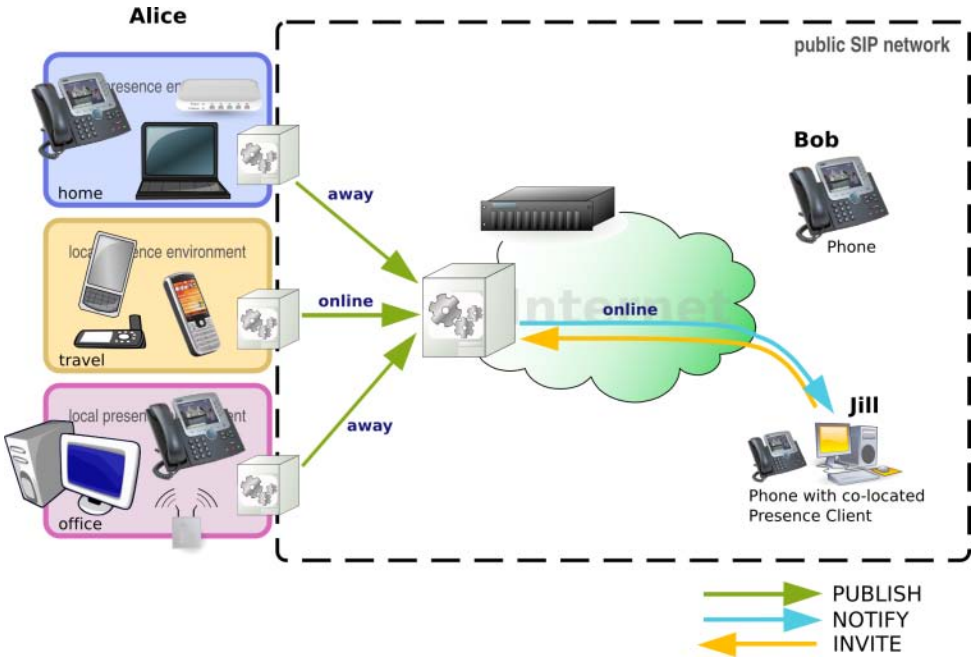

Figure 5.3: Complete call when the subscribed user becomes available

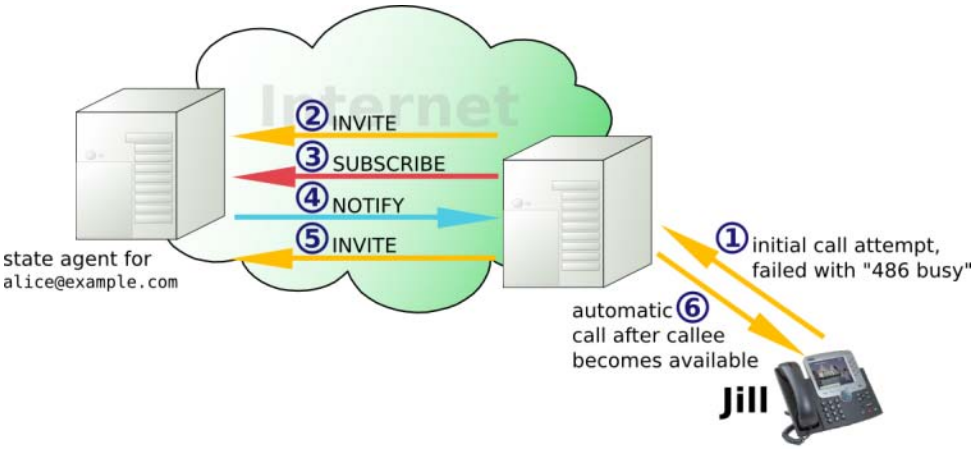

Figure 5.4: Server-controlled call completion to busy subscriber

5.2 Processing Model for Script-based Aggregation

An essential feature of the presence aggregation framework defined in this document is the semantic composition of presence information from multiple sources. Semantic composition (cf. Section 3.3.2.4) is especially helpful for low-level presence data where a meaningful composition based on the document syntax is not possible, and for composition of high-level presence information generated by a large number of equivalent sources. In our architecture, we address both scenarios, thus covering the complete information flow from the data sources to

the receivers of abstract presence status descriptions. The rules for composition of presence documents are given in user-specific scripts that specify how to transform the input data into new presence documents to be sent to subscribed watchers. This way, not only the aggregation process is controlled but also the creation of watcher-specific views on the presence data.

In this section, we discuss the actions that must be taken by a presence aggregation server when processing incoming presence documents. We suppose that the aggregation server is co-located with a SIP server for distribution of aggregation results. As shown in Section 7.2.3, the aggregation result may be published to a dedicated presence server using the SIP `PUBLISH` method or transformed in a watcher-specific output format for notification of entities via `NOTIFY` that have subscribed for the particular presence record.

In any case, the processing is governed by an aggregation specification that is used to combine incoming presence documents with the stored status information that has been generated from previous presence documents. The evaluation of aggregation specifications—called *PAL scripts*—is controlled by the *presence aggregation module* (PAM) which is responsible for providing correct input parameters and handling the scripts' output values. Several events may trigger script execution, e.g. the receipt of a status update notification for the corresponding presentity or a system-generated event (e.g. expiry of a user-defined timer).

When executed, the script is expected to fill a global variable, the output set, with presentity objects to be used as template for outgoing status update notifications. For each subscribed watcher the presence aggregation module creates an extended presence information document according to Section 5.3 including the presence tuples the watcher is allowed to see. If a watcher is not allowed to see any information given in the output object, no status update is sent to that particular entity.

To determine whether or not a watcher is permitted to receive status information about a specific communication channel of a principal, the PAM determines an *authorization class* the watcher belongs to. Authorization classes are sets of URIs as defined in Section 5.5. A watcher belongs to a particular class, if the watcher's URI as specified in the subscription request is listed in the definition of that class or if the watcher belongs to an included class. If contained in a class, the watcher is permitted to receive status information for communication channels that are bound to the authorization class as defined in Section 5.3.

Based on this general processing model, the remainder of this section discusses available design options for the specification language. Our main focus is on the treatment of inexact presence attributes that describe the current presence status as well as the creation of watcher-specific event notifications. Based on the results of this discussion, we have developed the presence aggregation module described in Section 7.2.2.

5.2.1 Basic Data Model

During the long history of distributed applications dealing with presence information in some way, numerous data formats have evolved to represent the information set of a presence information document. In general, all representations exist of a structured set of attributes describing the status of some abstract resource, the *principal*. A principal may be associated with several of these attribute sets over time causing status update notifications whenever a change happens. Here we propose a mathematical representation for presence attributes that can be mapped easily to most of the existing presence formats in the Internet today, especially the standardized presence information data format we have described in Section 4.2.

In its most generic form, the attribute set contains a number of pairs (k, v) with $k \in K$ being a unique key and $v \in V$ a corresponding value. Typed attributes may be defined as members of $V \subseteq V_t \times$ Type where Type is a set of type names and V_t is a vocabulary containing only values of type $t \in$ Type. Given this definition, a set $p \in K \times V$ that is an injective function K' $\longrightarrow V$ with K' $\subseteq K$ denoting the *presence status* of a *presence entity*.

Note that the presence status does not contain multiple values for a given key k. This is important because this facilitates mapping of different data models to this model. For example, *presence tuples* as defined in [RFC2778] use vectors v' $\in V_1 \times V_2 \times ... \times V_n$ to describe a presence entity's status. A bijective mapping can be created between both models by defining an ordering function that maps keys to natural numbers used as vector index. The same trick can be used to map the presence status to structured data types in imperative languages like Java or C++, or to payloads of binary presence protocols. (Note that nested structures can be mapped as well, e.g. with an indexing function that describes the path from the root of the nesting hierarchy to every node within the tree structure.)

5.2.2 Update Notifications

As the status of a given presentity may change over time, we define $\sigma_i(t)$ with $i \in I, t \in$ Time to be the status of the presentity with the unique identifier i at a system-specific time t. The function signature for σ_i then is defined as: σ_i: Time $\longrightarrow K \times V$, where $K \times V$ denotes the presence attributes as described before. It is important to note that this definition restricts the granularity with which status changes may occur. In mathematics, the set Time may be any infinite set with a total order, yielding a graph that shows the presence status for this particular entity at any point in time. Figure 5.5 gives an example of a graph illustrating an office worker's accumulated idle time. While no keyboard input or mouse movement is detected, the idle time increases continuously. As soon as the user enters any data, the value of the idle counter is reset to zero, and the accumulation process starts over.

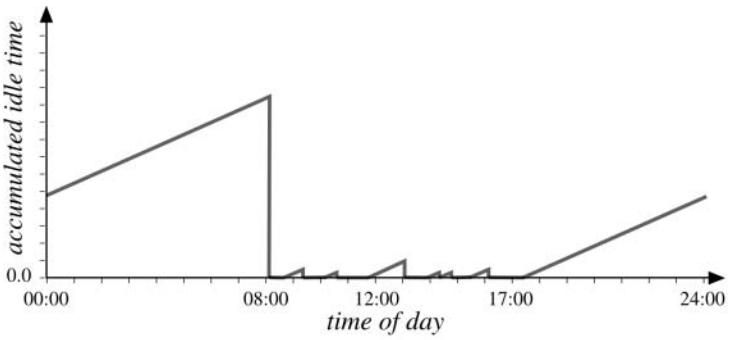

Figure 5.5: Example for the accumulated idle time of a single user

In this example, day work extends from 8 am to 5 pm, with a one-hour lunch break at noon. At night, no activity was registered, so the idle time has summarized up to about 15 hours immediately before the user starts working at morning. Until lunch, only small idle periods occur as the user is on the phone or gone for a coffee. The same holds for the afternoon until the user leaves right after 5 pm.

Because discrete event systems do not allow for continuous status updates, some maximum sampling rate delimits the frequency of observable presence changes. Observed status changes hence are modeled as *notifications* that are members of the set $N_i \subseteq \text{Time} \times P$ with $i \in I$ being the unique identifier of the changed presence entity and $P = K \times V$ being the set of presence attributes. In the most simple case, there is a set of sampling times $T \subseteq \text{Time}$ generating a notification set $\{(t,s) \in T \times P \mid s = \sigma_i(t)\}$. As we will see below, a *transformation function* may be applied to the presence status before a notification is created. Therefore, the sequence of notifications for a given principal is defined as $\{(t,\tau(\sigma_i)) \in T \times P \mid \tau\colon P \longrightarrow P\}$.

In a distributed system there will be additional data attached to notification messages to implement message routing, identity management, accounting, or reliable transfer if necessary. This meta data included with each message must be accessible from within the aggregation specification as it may contain important information that is not part of the status representation itself. For example, some presence protocols are known to have no strict separation between message transport and notification contents, hence using e.g. transport addresses to identify presence entities. The presence information document in this case does not contain any address information and thus impedes identity management based on the presence data. As discussed in Section 5.5, aggregation specifications are able to modify the set of authorized watchers for specific presence attributes and thus must know about the identities they deal with.

5.2.3 Handling Inexact Status Descriptions

So far, a principal's presence status has been modeled as graph over a continuous, fine-grained time axis. If watchers who are subscribed to that entity were notified instantly of any status change—without delay and at the same time that any other watcher gets this information—this model would be fine for presence applications. Unfortunately, in a globally distributed system based on a best-effort service a watcher cannot rely on timeliness and atomicity of the update notifications. Moreover, Internet protocols provide congestion control mechanisms to protect the network and specific endpoints from being overloaded with messages being sent too frequently. For presence protocols like SIP, there is a mechanism to limit notification messages being generated at the sender to a given rate.

These factors cause information loss at the subscriber as status updates may come late or do not come at all. This is even worse if the notifications are not self-contained, i.e. they represent a delta-encoding that refers to a previous presence status. To overcome these problems, notification protocols typically provide means to adapt the sender rate and to request notification of full status descriptions. For scenarios where neither is an option (e.g. because the network latency is too high or the bandwidth does not allow large notification bodies as is the case for typical wide-area mobile networks), applications should at least be able to determine what significance a specific presence attribute has for the current status.

The following example illustrates the problems that may arise: Consider a presence entity whose status description is derived from the principal's idle time. The idle time is throttled down to the order of minutes in order to avoid unnecessary traffic on the network. Now, if the principal has just finished writing a document and left his office there exists a small time span with an inconsistent presence status: The last notification sent a few seconds ago indicated the principal to be available while the next update notification will be sent in about one or two minutes. A watcher trying to call him will experience some sort of *false positive*: the call cannot be established because the callee has left although the presence system indicated availability.

Though it is oversimplified to some degree, this example already reveals many of the problems that arise when dealing with constant presence attributes being distributed more or less frequently: If the called user hears his phone ringing, he might return to his office and answer the call. In this case, the presence system correctly indicated his status as being available for calls. Unfortunately, if the user was too far away, he would not have heard the ringing and consequently did not return to his office. As the calling party does not know about this, it might observe an indeterministic behavior in the presence system, the *false positives* mentioned before.

This problem is inherent to the indeterministic nature of the event notification system being used: information may be delayed, lost or suppressed. In addition, the decision to get back into the office and to accept or to decline a call cannot be anticipated by any presence system as it is completely up to the user what the indicated presence status means in this particular situation. Thus, a computer system cannot decide whether or not a user is available for communication using a specific channel—it can only give hints about what various sensors have measured. This situation is best modeled using a probabilistic approach: Presence status attributes are annotated with a *probability value* that indicates the particular attribute's exactness at a specific point in time.

The exactness of information can degrade over time i.e. with the age of that data. In the example above, the probability of successful call completion decreases after the principal has stopped typing and nothing is known about his current location. The notification of the writing activity therefore underlies the same aging process and will degrade over time as other presence attributes. To implement this behavior, presence attribute values are modeled as pairs of the last known status and a function that takes the current time as input and returns a number within the interval $[0,1]$ indicating the exactness of the status value. Figure 5.6 gives an example for the decreasing exactness of an event description over time.

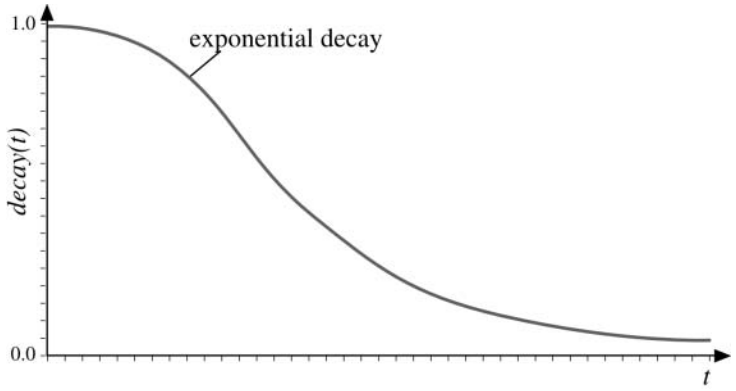

Figure 5.6: Example for a decay function for status descriptions

The quality of approximation provided by this decay function depends on a proper exactness function: The better the degradation of sampled presence data is simulated by the mathematical function associated with that item the better a watcher can predict the probability of successfully contacting the user without knowing the exact presence status. Looking at common system configurations, three different classes of degradation functions have been identified:

- *Constant*

  Making a presence attribute constant yields the behavior of common presence systems that have no aging mechanism. The attribute value picked from a finite vocabulary is published as is, without any statement about lifetime or quality of the value being indicated. Usually, the interpretation of constant attributes is left to the user.

  A common application for constant attributes is the advertisement of asynchronous communication channels that are always available, e.g. electronic mail or voicemail.

- *Linear*

  To model the degradation of information quality over time a linear function can be used. This is useful especially for attributes that have only small impact on the overall presence status. For example, a typing indicator could be represented by a linear function with a small decrease factor. While the user is typing, updates are sent in large intervals—e.g. five minutes—to avoid network congestion. When typing has stopped for more than, say, ten seconds, the presence system will send an update notification indicating that the user is not typing any more. The watching application does not know in advance when the next status update notification will arrive. Therefore, it displays a nearly stable state (pretending that the subscribed entity is continuously writing), adjusted only by a small decrease of the exactness value.

  The linear decay function is defined in the interval $[\,\text{timestamp,timestamp} + k\,]$ as:

  $$\text{linear}_k(t) = 1 - \frac{(t - \text{timestamp})}{k}$$

  and $\text{linear}_k(t) = 0$ for any other t.

- *Exponential*

  The most natural way of expressing degradation is an exponential function starting with an arbitrary initial value at time t_0 and falling steadily with an exponential factor to a non-negative limit. The function approximates the behavior of spurious events or events that occur in irregular intervals like phone-calls or ad-hoc meetings. After such an event has occurred, the presence status does not necessarily change immediately, but the information degrades faster than in the linear case. Consider the above example of a user who stops typing and goes home just at the moment a watcher decides to call him. While the typing information is represented by a linear function as stated above, the finish event is modeled with an exponential function as it gives only a snapshot of the user's current activity (if treating the stopping of a task as activity at all). The fact that the user may be reached for some seconds after the finish event has occurred is reflected by an initially low degradation which grows the longer the initial event has passed.

  The exponential decay function is defined in the interval $[\,\text{timestamp},\infty)$ as:

  $$\log_k(t) = e^{-k(t-\text{timestamp})}$$

  and $\log_k(t) = 0$ for any other t.

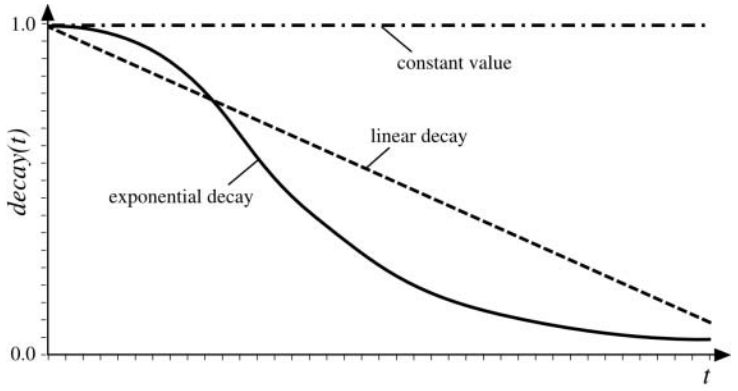

Figure 5.7: Different types of decay functions

An example for the three types of decay functions is shown in Figure 5.7. Note that the value $d(0)$ for a decay function $d \in \{\text{const,linear,log}\}$ may be any real value in the interval $[\,0,1\,]$.

The drawback of using decay functions is the increased effort the aggregation server has in re-calculating the current presence status from time to time even if no update message has arrived. As a decision support the sender adds a fixed *threshold value* with the attribute's degradation function that marks the particular point where the information loss for that attribute is too high and the presence status may change significantly. When multiple attributes are treated together in order to determine a principal's presence status, the threshold values may be taken as hints when a re-calculation should be performed. More on this topic is given in the next section.

5.2.4 Event History

The ordered sequence of notifications observed at some point in the information system is called *presence history*, or *history* for short. A vector $((t_1,s_1), (t_2,s_2), ..., (t_n,s_n))$ with $t_i \in$ Time and $s_i \in P$ for all i=1, 2, ..., n is called *history vector* if $t_i < t_{i+1}$ for all i=1, 2, ..., $n-1$. We define the operator $\circ$: $V^n \times V \longrightarrow V^{n+1}$ where V=(Time $\times$ P) to append a given tuple (t_{n+1},s_{n+1}) to a history vector $((t_1,s_1), (t_2,s_2), ..., (t_n,s_n))$ if $t_n < t_{n+1}$. For $t_n \geq t_{n+1}$, the result is equal to the initial history vector used as input argument. Appending the tuple (t,s) to the empty set yields (t,s) as result.

Applications can use the history of observed status changes to infer additional status information. For example, a graphical user interface showing the idle indication of a remote entity may decide to display the status as being `non-idle` instead of frequently toggling the description if it has observed long `non-idle` periods interrupted by short `idle` times. The exactness value may be adjusted accordingly to smoothen the presence graph.

In this thesis, history vectors are used for aggregation of multiple presence sources. As there is no synchronization between contributing sources, an aggregation server may decide to store incoming notifications for some time before a new presence status is calculated and published.

5.2.5 Transformations

The most important aspect of presence aggregation is the transformation of unrelated status descriptions into a global view on a principal's presence status. The transformation process is used to convert different formats of presence information documents into the generic representation used internally by the aggregation engine.

Speaking formally, a *transformation* τ of a presence status is defined as function $P \longrightarrow P$, i.e. a given status record is mapped to another status record. A transformation can change the amount of information a status record contains. This definition does not restrict the amount of context information used by the transformation function itself. Hence τ might use any other status record, a history vector, sampling information etc. to calculate the transformation result.

The above definition reflects the intended operation of an aggregation engine: Whenever a status update notification containing a presence information document arrives, it is either dropped or used as input for the transformation process. When transformed, the result is fed to the output processor that generates exactly one presence information document for every authorized watcher.

Several options exist to transform presence documents, ranging from multi-dimensional functions applied to the vector of presence attributes to rule-based transformations on its XML representation. In our implementation, we have decided to use *XSL transformations* [XSLT] to convert between different presence document formats as described in Section 7.2.2.2.

Having converted incoming status notifications into a suitable internal representation, the principal's global presence status can be reconsidered with respect to this new information. In Chapter 3, we have argued that aggregation of presence information from multiple unrelated sources can hardly be done automatically. Even if the first transformation step could adjust the semantic level of presence attributes coming from different sources, the aggregation engine has not enough application knowledge to handle this information. For example, a voice mailbox at the user's home indicates availability by sending SIP `REGISTER` requests to the user's telephony service provider while the presence application on his PDA describes the user to be busy. From the perspective of the aggregating entity there are two contradictory status descriptions, and it cannot tell which one to trust more. Instead, the publishing user has to provide a set of rules to be used to solve this conflict.

One possible approach to do this is by providing a complete description of presence sources, including their capabilities and limitations intended by the IETF presence data model (see Section 4.2.2.2). A set of processing rules must be specified by the user to handle conflicts between devices. The problem with this approach is that several finite vocabularies must be defined to be able to describe presence sources and author processing rules. The more automatic reasoning is desired, the more expressive such vocabularies have to be as seen for the pattern language of the Mobile Interactive Spaces project we have described in Section 4.1.3.

The inflexibility caused by finite vocabularies as well as the complexity induced by automatic reasoning render these approaches unusable for effective presence aggregation. In a global information system, data formats and protocols have to be as simple as possible to facilitate wide deployment but must be flexible enough to meet the users' needs as presented in Section 3.1. In addition, backwards-compatibility must be assured to avoid breaking existing client implementations users are accustomed to.

To leverage the adoption of presence services by implementors and users, we have designed the presence aggregation language based on an existing programming language that is already

known to developers of Internet-based services. With *ECMAScript* [ECMA-262], we have found a standardized object-oriented language with well-defined semantics. Being a subset of *JavaScript*, the language is widely used for implementing dynamic Web applications, hence a significant number of developers already knows the syntax to be used for writing aggregation specifications. The definition of the specific object model and library functions needed for authoring aggregation specification is contained in Chapter 6.

5.2.6 Output Generation

The final step of the presence aggregation process is the generation of presence documents for dissemination to subscribed watchers using the Session Initiation Protocol. The distribution of the newly created presence information document can happen in two ways, depending on the configuration of the aggregation server:

- If the server is a node within a hierarchy of aggregation servers, the generated document is forwarded to the next aggregation server using the SIP `PUBLISH` method.
- If the server handles presence subscriptions to the entity whose status has just been updated, a watcher-specific view of the presence document is sent to every subscribed watcher. When sending these status descriptions, the server must be careful not to disclose any sensitive data contained in the source document. To guarantee this, every presence attribute has an authorization class associated with it as described in Section 5.5. Before sending the presence status description to a watcher, it is transformed to contain only those presence attributes the client is permitted to see. Other attributes as well as the definition of the authorization classes are removed from the document prior to sending it.

The last step is an optional transformation of the PIDF document into a specific format requested by the subscribed client. For example, some instant messaging programs pre-dating [RFC3863] (like e.g. *Microsoft Windows Messenger* and *kphone*) use a document format similar but not equivalent to PIDF. These formats are based on XML, with a PIDF-like structure, so a simple XSLT transformation can be used to map between these document formats as described in Section 7.2.2.2.

5.3 Enhanced Presence Information Data Format

The aggregation process is defined independent of a particular representation of presence attributes on the network. The minimum information that is required to exist in a presence document is the presentity's identity and a (possibly empty) set of channel definitions with their specific presence attributes. To ensure interoperability with existing presence services, at least the following information items required by [RFC2778] must be contained in a presence document:

- The principal's unique identifier

  A globally unique name that represents the user whose presence status is published. Typically, the principal is identified by a URI with the protocol-independent scheme `pres` defined in [RFC3859] or a transport-specific scheme like `sip`. For compatibility with

proprietary protocols, other formats may be used as well, as long as they can be proved to be globally unique at least for the time the presence publication process is active.

- Presence status description

  For every communication channel offered to contact the principal, a formal description of its status is included. According to [RFC2778], a finite vocabulary is used to denote the basic status (`OPEN` and `CLOSED`). A presence application may use an extended set of status descriptors to give hints about the user's current activities. For example, *Windows Messenger* from Microsoft Corporation defines additional literals `busy`, `onthephone`, `idle`, `online` etc. as additional qualification of the basic status. A complete documentation of this application's vocabulary can be found in Section 4.2.1.

- Contact addresses

  Communication channels in this model are supposed to have a transport address associated with them where messages can be sent. For conformance with [RFC3863], channel descriptions without a contact address are allowed, but should be avoided where possible.

- Timestamp

  The optional timestamp denotes the time when the status sample has been created. The timestamp value allows for global ordering given that the distributed systems' clocks are nearly synchronized. If a status update notification does not contain a timestamp, the aggregation engine instead has to insert the time of arrival of the notification message.

  As for [RFC3863], the timestamp has to be specified in the standardized format for date and time specifications on the Internet [RFC3339].

By design, the core attribute set presented here is a subset of the *presence information data format* (PIDF) defined in [RFC3863] (see Section 4.2). The extension mechanism of PIDF allows for additional attributes to be included in status descriptions without breaking existing clients. As many presence clients already support this XML-based document format, it is used by the aggregation system as native syntax, too. To convey additional information items between aggregation servers, the specific XML namespace name `urn:ietf:params:xml:ns:pidfxy` is used to mark the extensions.

A basic example of an enhanced PIDF document using extension elements from the namespace `urn:ietf:params:xml:ns:pidfxy` (called "PIDF-XY" for short) is given in Example 5.3.1.

The basic structure of this presence document is defined by [RFC3863] to ensure interoperability across the SIP-based presence service. The extension elements use a prefix `pidfxy` that is bound to the namespace of PIDF-XY. A standards-compliant presence server then is allowed to drop the extension elements it does not recognize when processing the document.

The example document shows the essential language features of PIDF-XY that are used to annotate the status description contained in the presence tuple that is specified here. In the semantic model of PIDF-XY, the element type `tuple` describes a communication channel of the presentity that is identified by the attribute `entity` of the surrounding `presence` element. To augment the status description for a communication channel, PIDF-XY defines the following element types:

Example 5.3.1: A simple PIDF-XY document

```
<presence entity="pres:alice@eaxmple.net"
               xmlns="urn:ietf:params:xml:ns:pidf"
               xmlns:pidfxy="urn:ietf:params:xml:ns:pidfxy">

  <!-- media channel: shared office phone  -->

  <tuple id="t18">
    <status>
      <basic>OPEN</basic>
      <pidfxy:name>Jabber</pidfxy:name>
      <pidfxy:type type="tokenlist">COMMUNICATION, INTERACTIVE, TEXT</pidfxy:type>
      <pidfxy:authclass type="string">protected</pidfxy:authclass>
      <pidfxy:ownership type="number">1.0</pidfxy:ownership>
      <pidfxy:interval type="number">35</pidfxy:interval>
      <pidfxy:typing type="boolean" timestamp="2006-10-01T17:13:06+02:00"
          decay="linear(500)" threshold="0.4">true</pidfxy:typing>
    </status>
    <contact priority="0.8">
       sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE
    </contact>
    <note xml:lang="en">
       You can contact me via our company's SIP-to-Jabber gateway.
    </note>
    <timestamp>2006-10-01T17:13:12+02:00</timestamp>
  </tuple>

<presence>
```

- `name`

  Specifies a unique name for this particular channel that can be referenced in aggregation specifications. This element type is useful for presence sources that generate tuple identifiers that change for every update of the status description.

- `type`

  The element `type` describes the media type of the communication channel using an extensible vocabulary. The attribute `type` of this element can be used by validating parsers to check the format of the element contents. In the given example the element contains a list of tokens that are separated by commas.

  The tokens contained in the element `type` are combined to describe the role of the communication channel. Pre-defined values for the element `type` include `COMMUNICATION` to denote a channel that can be used for interpersonal communication. The token `INTERACTIVE` denotes bi-directional communication. If not specified, the channel describes typically an automaton such as a voice mailbox. The third item in this list defines the specific communication media. In this case, the channel is used for text messaging. Other values are `AUDIO` for speech conversation and `VIDEO` for visual links.

- `authclass`

  The authorization class defines which user groups are authorized to access the presence information generated by this channel. A detailed description of this element's semantics is given together with the discussion of the authorization policies in Section 5.5.

- `ownership`

  Each channel provides presence information about a particular user. As some channels represent devices such as public or shared phones that cannot be associated to a single user the element type `ownership` allows to specify to which degree the respective presence information can be associated to the user whose presence status is described by this document. The value of `ownership` must be a numeric value between 0 and 1.0.

- `interval`

  The element type `interval` can be used to specify the sampling frequency for extended presence attributes. In the example document the interval specification is related to the element `typing` that describes the principal's current activity. `interval` contains a numeric value specifying the number of seconds between two samples. In this case the status is updated at minimum every 35 seconds.

- `typing`

  The presence attribute `typing` indicates that the principal is currently writing text on her keyboard, reflected by the boolean value contained in this element. Depending on the channel's media type, different activity elements can be used. For example, a telephone provides a boolean attribute `incall` that is set to `true` when the media channel is currently in use.

The start tag of `typing` contains additional attributes to describe the degradation of this value. The decay function denoted by the value of the attribute `decay` is defined in Section 5.2.3 as:

$$\text{linear}_{500}(t) = 1 - \frac{(t - \text{timestamp})}{500}$$

The attributes `timestamp` and `threshold` determine when the given value is outdated and a recalculation of the presence status is required. In this case, the start time is given as `2006-10-01T17:13:06+02:00`. The threshold value will be passed after five minutes, because

$$0.4 = \text{linear}_{500}(\text{2006-10-01T17:18:06+02:00}) = 1 - \frac{300}{500}$$

It is important to notice that the calculation of decay values uses the value of the attribute `timestamp` if specified in the start tag of the respective element as the `timestamp` element in the contents of `tuple` might be added later (e.g. at receipt of the PIDF document at the aggregation server).

With PIDF-XY extension elements, presence documents can be used to convey data that was generated by low-level sensors to a presence aggregation server without extensive preprocessing. The only requirement is that the generated traffic must overload neither the network nor the receiving server. Hence, sensors that generate frequent status updates should use alternative functions such as sensor-fusion before a PIDF-XY document is generated from the results.

The use of PIDF-XY in presence documents thus combines the advantages of *sensor-fusion* as described in Section 4.1 with the improved control of aggregation provided by user-specific scripts. Examples for efficient and powerful aggregation functions are given by Hartmann in

[Har04], where the annotation of presence attribute with an explicit decay function was proposed for the first time. Later, these concepts have been elaborated in a joint research effort with the TU Darmstadt that led to the definition of the extensions to PIDF proposed in [Gör05] as well as the refined document format discussed in this thesis.

5.4 The Local Presence Environment

In the previous section we have discussed the PIDF-XY extensions to annotate channel descriptions with specific meta-data typically generated by the local presence environment. This additional information can be used by an aggregation server to determine a consistent presence status and to re-calculate the status when the given information is outdated. In this section, we describe our architecture for a local presence environment that contains various sensors sending data samples to a *personal presence server* (PPS) that is used for aggregation of this data and distribution to the next-hop aggregation server. The transmission of data samples may happen continuously—even when no status change occurred—, after comparison of subsequent measurements, or may be triggered by external events such as a motion being detected by a passive infrared sensor.

As sensors can be added or removed at any time, a message-oriented middleware must be used for communication within the local presence environment. We have selected the *Message Bus* (Mbus) defined by the IETF in [RFC3259] for this purpose as it provides a flexible addressing scheme and various coordination patterns ranging from loose coupling to tight integration of entities using RPC-style commands. The multicast-based message transport allows for dynamic adding and removal of components, with automatic detection of existing or new entities.

In addition to these features, the Mbus has the advantage that it is a communication protocol without any platform-specific dependencies. Numerous implementations exist in different programming languages on UNIX-based platforms as well as on the Windows-family of operating systems. A detailed description of the Mbus together with a comparison against alternative messaging frameworks can be found in [Kut03]. The document also contains a survey on the applicability of the Mbus as a coordination framework for components of conferencing systems, especially for controlling call signaling servers and media processors. The results show that the Mbus is a reasonable choice especially for component-based applications such as modular protocol gateways.

Being a generic communication platform, the Mbus framework only provides a small number of pre-defined commands, used primarily for outage detection and clean shutdown. In addition, a number of templates is defined to facilitate specific communication patterns such as point-to-point RPCs and anycast messages. Additional primitives must be defined by the application designer in an application-specific *Mbus profile* that specifies a vocabulary for Mbus commands together with their semantics. The Mbus framework supports this task with a set of guidelines to be used for a clean design and well-defined semantics of custom Mbus commands.

The remainder of this section describes the *Mbus application profile for presence servers* we have created to define the architecture of the local presence environment. But before going into the details of the command set for controlling the decomposed presence system, we provide some background information on the distinct roles that Mbus entities have in a local presence environment (Section 5.4.1) and discuss the Mbus-specific addressing scheme in Section 5.4.2, followed by the relevant interaction models the messaging framework supports (Section 5.4.3).

After having clarified the terminology and representation conventions generally used in Mbus application profiles, Section 5.4.4 gives a detailed overview of the command set that can be used to communicate with the components of a local presence environment.

5.4.1 Mbus Entities

The local presence environment is characterized by its support for multiple sensors with ad-hoc-style communication. The status updates published by these sensors typically have a very low semantic level, i.e. additional knowledge is necessary to interpret the data. In addition, the sensors' update frequencies may be very high as continuous measurements are published on the local messaging channel. An aggregation engine hence not only collects data from multiple sensors to calculate a consistent presence state for the local presence environment but also alleviates the number of update messages sent into the global SIP network. Figure 5.8 illustrates this throttling of update notifications with respect to the overall system architecture.

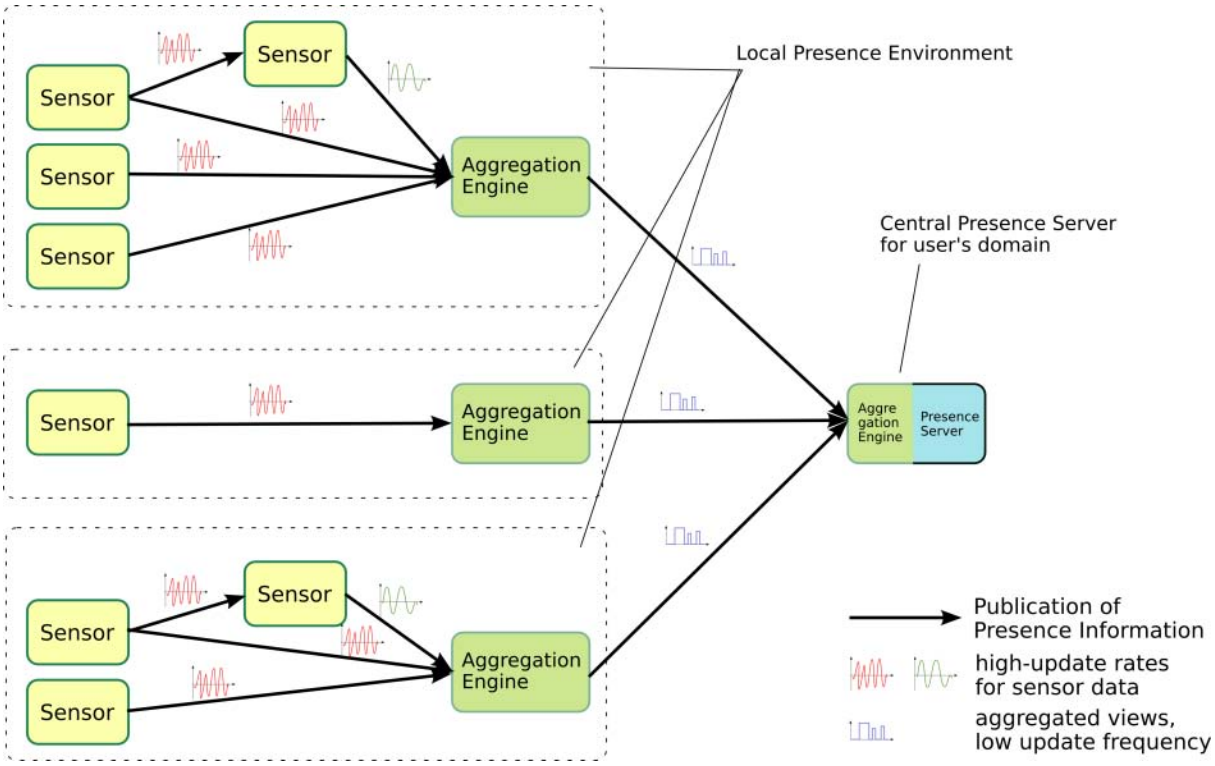

Figure 5.8: Aggregation engines throttling status update notifications

The picture shows three local environments with several sensors, e.g. desktop phones, door sensors, screensaver demons. A specific type of sensors—called *virtual sensors*— can use the data published by other sensors to calculate new sensor values and publish these as well. For example, a virtual sensor could be used to implement status changes based on a trigger function that fires when a certain sequence of events has been detected, similar to the matching of pattern sequences in the SIENA event notification system. [Car98].

A thorough investigation of devices that can be used as presence sensors is documented in [Har04]. The author shows examples for calculation of presence states from idle times or

keyboard activity on computer systems as well as the call status of communication devices such as mobile phones and ISDN adaptors. Even the presence of bluetooth devices is used as possible indicator for the user's availability. In addition, a decay function can be provided to instruct aggregation engines to indicate the actual exactness of published presence attributes. In Section 5.2.3, we will show how this information can be used by aggregation engines to calculate a new presence status.

The aggregation engines within a local presence environment act as *personal presence servers* (PPS) to the outside world, i.e. they publish the aggregated status of the user's local environment either to subscribed watchers or to another presence server as shown in Figure 5.8. As we use the Session Initiation Protocol (SIP) for distribution of presence data, the PPS also acts as outbound proxy for the local environment and handles `REGISTER` requests for the corresponding subdomain. SIP requests for other domains (including the domain of which the local environment is a subdomain) then are forwarded in accordance with [RFC3261]. Acting as a local registrar, the PPS is able to aggregate registration updates with other types of sensor information that occurs within the local presence cloud.

A major improvement regarding existing systems like SIENA or impress [Har04] is the evaluation of user-specific scripts with incoming status updates to give users more control over their data. The evaluation process is triggered by status update notifications that have been received via SIP or on the local message bus. With optional pre-processing of received sensor data, a higher abstraction level can be achieved.

Before discussing the details of the script-based aggregation process, we give a detailed description of the local communication infrastructure and the enhanced format for presence documents generated by the personal presence servers. First, the command set for event notification and control of the aggregation engine is described. Though compatible with the standardized format PIDF presented in Section 4.2.2, the content format used for event notification on the local message bus does not use the common XML notation to be more compact for local delivery. While the mapping into XML syntax is straightforward for most of the presence attributes, there are some extension attributes that have no direct counterpart in PIDF. The integration of these components with PIDF concludes this section.

5.4.2 Addressing Scheme

Our presence-specific application profile utilizes the tuple-based addressing scheme of the Mbus framework to denote the roles that components have in the local environment. Sensors, for example, can be identified by their address tag `module` having the value `sensor`, while the personal presence server has the value `presence`. The unique address of a specific sensor then is a combination of several address tags that give additional hints about the type of that particular sensor. The mandatory tags `app` and `id` show the implementation-specific application name and a unique identifier for the actual instance of that application. The latter is needed specifically to enhance the robustness of the Mbus platform and can be safely ignored by the application profile.

Figure 5.9 shows an example of an Mbus environment with five entities. These entities reflect the four sensors and the aggregation engine that is shown in the upper left of Figure 5.8. The distinct address tags not only reveal the types of sensors in this system but also help to cluster Mbus entities into groups.

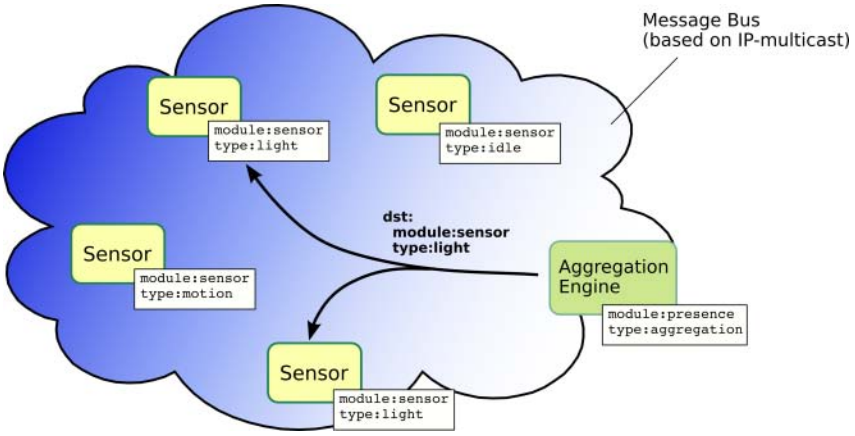

Figure 5.9: Tuple-based Addressing in the Mbus Environment

A data packet sent on the multicast-based messaging bus will be received by any entity that has subscribed to that particular multicast group on the network layer (given that is in the sender's scope indicated by the `TTL` field of the Internet Protocol header). Receiving entities then filter out messages that do not match the particular entity's Mbus address or a subset thereof. Thus, if an Mbus command is to be processed by the aggregation engine but not by one of the sensors in our example, it can be sent to the following group address:

```
(module:presence type:aggregation)
```

An Mbus module can use additional address tags to express additional roles it takes. For example, a presence module that provides an interface to a SIP network for dissemination of presence documents over the Internet could include the tag `protocol:sip` in its address to indicate the external presence protocol it provides an interface to. A presence document that is to be published to an external presence server then could be sent to the group address

```
(module:presence protocol:sip)
```

The aggregation engine containing the specified `protocol` tag then would receive the publication request and act accordingly.

As sensors should not get requests like this, another value for the tag `module` is used. A command that is meant to be handled by one or more sensors is therefore tagged with the following destination address:

```
(module:sensor)
```

Additional `type`-attributes might be specified to further narrow down the command's scope. If, for example, the light sensors within the presence zone representing a user's home should be queried for their status to determine if there is a light somewhere in the house switched on, an anycast command was sent to the following Mbus address:

```
(module:sensor type:light)
```

This flexible scheme for addressing groups of components can be used for any command with explicit semantics for groups of receivers. Note, however, that only one address tag `type` can be specified in a single address, and that the value of an address tag is always a single token. Thus, no complex unions of address groups can be defined for sending a command to different types of sensors (such as "all motion and sound sensors"). If this was desired, a new sensor type must be defined. In this case, care must be taken not to break existing implementations that rely on a specific type's semantics. This is a problem especially for super-types, i.e. a union of existing types such as a door sensor that reports the time span the door has been open.

Table 5.1 lists the available values for the address-tag `type` for `module:sensor`.

Type	Description
`light`	Used for sensors that report the status of a single light source. Values are `on` or `off`. Various other types have been defined for sensors that report environmental data such as `temperature`, `humidity`, `acceleration`, `velocity`, `location`.
`door`	Used by sensors that indicate the position of an office door, usually interpreted as indication of availability. Door sensors may use abstract status tokens such as `half-open` or `closed`, or may publish a numeric value that indicates the openness in percent.
`chair`	A chair sensor detects the pressure on office chairs to determine if someone is sitting there.
`idle`	Usually a virtual sensor that calculates the time since detection of the latest input activity. On most computer systems, the screensaver demon also can be used to determine if the user has been idle for some time.
`motion`	Motion detectors can report the presence of persons at a specific location such as an office or a hallway. In general, the detection is based on signals of passive infra-red sensors or—in the case of surveillance cameras—on calculation of differences between subsequent video images. Therefore, the sensor will only report the fact that there has been a motion but no identification of the moving person is done.
`sound`	The presence of persons can be derived from continuous recordings of a microphone, similar to the camera-based motion detection shown before. To detect human voices, the background noise must be filtered from the recording. Having done that, remaining sound samples with specific characteristics indicate the presence of talking people which is published on the Mbus.

Table 5.1: Address tags for distinct sensor types

Based on this scheme for component addressing we have defined a set of commands for controlling decomposed presence applications. As some of these commands make use of the non-trivial interaction models defined in [Kut01], the next section gives a brief summary of the relevant interaction patterns offered by the Mbus framework.

5.4.3 Relevant Mbus Interaction Models

As the Mbus is based on IP-multicast, messages are not guaranteed to be sent reliably by the underlying transport layer. Moreover, as there is no formal registration of Mbus entities, a short delay exists between the actual appearance or disappearance of an entity and its detection by other entities (using pre-configured timeout periods). This delay may cause a race condition when messages are sent to the network although the destination entity is already gone. Depending on the actual application area, this may or may not be a problem. For example, presence detection may be based on environmental data published by a temperature sensor at a fixed update frequency (say, once every ten seconds). As the temperature only increases very slowly at the presence of humans, losing a few update messages is perfectly acceptable.

Some commands, however, must be transmitted reliably (e.g. to give feedback on user interactions) and hence must be acknowledged by the receiver. The Mbus—unlike many other messaging platforms—supports both interaction models as well as some variations that address additional aspects such as atomicity, consistency, serializability, and durability of transactions. Application designers then can select the best interaction model regarding their requirements without having too much overhead for applications where reliability is not an issue.

The presence-specific application profile defined here makes use of the most common models of interaction, i.e. *remote commands*, *event notifications*, and *remote procedure calls*. Remote commands and event notifications are defined as one-way messages that do not require an acknowledgement on the application layer. Transport layer acknowledgements are required by [RFC3259] only for point-to-point messages because of the additional complexity in terms of signaling overhead that reliable group-messages would introduce. As one-way commands do not expect responses anyway, there is only little benefit in detecting whether or not all addresses have received the message. As described later, the presence profile uses these two interaction models especially for event notifications and publication of status descriptions.

Remote procedure calls (RPCs)[32] are a well-known concept in distributed systems such as the *Network File System* (NFS) [RFC3530] for invocation of operations on a remote system. Mbus RPCs build on the basic messaging system that is also used for remote commands, i.e. an RPC is just a pair of a request message with its corresponding response from the remote peer. A unique RPC identifier is used by the sender to match incoming RPC responses to their corresponding request. This way, multiple RPCs can be invoked asynchronously as long as they do not require a specific execution order.

RPC responses carry two distinct result codes reflecting the transport-specific result of the RPC and the result of the operation on the application layer, respectively. If the RPC could not be handled at the server side (e.g. because there is no handler function for the specified operation) an RPC error would be indicated in addition to an empty application result. Otherwise, the application-specific result of the called function would be denoted together with a symbolic status code from a fixed vocabulary. The following example shows the text-based syntax of an RPC invocation and its corresponding response message (lines are wrapped for readability):

```
math.div ((("ID" "37") ("RPC-TYPE" "UNICAST")) (17 0))
math.div.return ((("ID" "37") ("RPC-STATUS" "OK"))
                 ((ERROR DIV_ZERO "E:division by zero") ()))
```

Here, the operation `math.div` is invoked with two application-specific parameters, the numbers `17` and `0`. The first parameter of this Mbus command is a list of two key/value pairs

[32] One of the first RPC protocols, *ONC RPC* from Sun Microsystems, is described in [RFC1831].

containing parameters for RPC processing, the first of which is a unique identifier for this particular RPC, "37". The second parameter indicates that the message's destination address identifies a unique entity that must handle the RPC. Alternatively, the keywords "ANYCAST" or "MULTICAST" could be used to request one or more answers from a group of recipients. In this Mbus application profile, we do not make use of these features because of their increased complexity compared to the unicast RPC mechanism.

A possible RPC return message is shown in the second line of the previous example. Again, the first argument is a list of key/value pairs containing RPC-specific parameters. The ID parameter correlates the return message with the original request. In addition, a mandatory status token is given as value of the RPC-STATUS field. The value "OK" denotes that the RPC invocation was successful, but nothing was said about its application-specific result. The result of the function evaluation is contained in the second argument of the message math.div.return. Its first part always consists a list of a status token (ERROR or OK), an application-specific symbolic result code (such as DIV_ZERO to indicate an attempt to divide by zero), and a textual description of the error for presentation to a human user. The second part of the application-specific result contains the actual result of the successful operation or is empty in case of an error.

The example given here also shows the representation of messages on the Mbus that is also used as command syntax in this document. A command always begins with a structured token, the command name. RPC returns are identified by the RPC command name with an appended suffix ".return". The arguments of an Mbus command are organized in list structures that are delineated by parentheses. From the various simple data types defined in [RFC3259], the previous example shows integer values (17, 0), strings ("ID", "OK"), and tokens (ERROR). Additional data types are floating point numbers and base64-encoded data blocks (enclosed in angle brackets).

Based on the general interaction models shown in this section, we have defined a set of presence-specific Mbus commands for controlling a presence server and sensor devices. The following subsections give a detailed description of their syntax and semantics. We describe specific commands for publication of events and explicit retrieval of status descriptions. After that, we introduce additional commands for controlling the aggregation engine, upload aggregation specifics and modify system parameters.

5.4.4 A Presence-specific Mbus Command Set

After having clarified the terminology and representation conventions of Mbus application profiles in the previous sections, we will discuss the *Mbus application profile for presence servers* we have defined during our research work. Our profile was created according to the conventions of [Kut01], and therefore we have defined address tags that reflect the distinct classes of Mbus entities in our profile, and have re-used existing Mbus interaction models where possible. The command sets discussed in this section address the publication of presence events (Section 5.4.4.1), the management of subscriptions (Section 5.4.4.2) and the modification of parameters related to the operation of the presence aggregation module (Section 5.4.4.3).

5.4.4.1 Publication of Presence Events

The central function of a presence service is the transmission of status change notifications to interested users. In this section, we define an Mbus event message that is used for unsolicited sending of entire status descriptions to subscribed users, and an RPC for explicit retrieval of presence status descriptions from Mbus entities. In addition, we define an RPC command to enforce the publication of presence documents by an external protocol gateway. The latter command allows decoupling of aggregation engine and protocol gateway. This way, the Mbus environment can be used to implement multi-protocol gateways that interconnect several presence protocol domains.

The Mbus commands for event publication and subscription management use the prefix `presence` to indicate their semantics. Actual commands hence are named `presence.notify` or `presence.subscribe`. The control commands use the standard prefix `param` for modifying system parameters of Mbus entities, and an entity-specific prefix hierarchy for other operations. The *presence aggregation module* (PAM) therefore provides handlers for commands beginning with the string `pam`. A detailed description of this category is given in Section 5.4.4.2.

Event Notifications

The command `presence.notify` is generated either by a protocol module that has received an external event notification or by sensors implementing the presence-specific Mbus command set. Recipients of the command are all subscribers for the event type specified with the event description. A notification command looks as follows:

```
presence.notify (PRESENTITY TIME WATCHERS EVENT DESC)
```

The meaning of the command arguments is shown in Table 5.2. The types denote the Mbus types as defined in [RFC3259]. A list of a specific type is indicated by parentheses, base64-encoded data blocks are called `Data`.

The notification command provides a generic mechanism to indicate the occurrence of status changes to entities on the local message bus. Explicit subscriptions to potential senders of notification commands are required to restrict notification messages to those events which are of interest to other entities. A specific optimization of the Mbus transport layer then can be used to save bandwidth for messages with no or a small number of interested peers.

The notification message refers to the status record identified by the first argument, `PRESENTITY`. Usually, this identifier will have the form of a URI as defined in [RFC3986] but there is no formal restriction of this string's format. Alternative formats could be used as well, as long as the identifiers are globally unique. Entities that subscribe for the `presence.notify` message must be prepared to receive notifications with identifiers not in URI-syntax. This may happen especially when acting as a gateway for older presence protocols.

The watcher list can be used to name the external watchers that should be notified of this event as specified in a previous subscription (see Section 5.4.4.2). Thus, the aggregation engine may create different views of a single presence document, each of which is published to a specific set of subscribers. The receiver of this notification must not publish the event to any external watcher that is not contained in the given watcher list. The only exception to this rule is the special watcher list `"*"`. If this string is given as the only entry of the watcher list, the particular event may be published to any external watcher.

Name	Type	Description
PRESENTITY	String	Unique identifier of the presentity the event notification is sent for.
TIME	String	The local time specifying when the event has occurred. The value of TIME must be specified in the syntax of [RFC3339] to avoid confusion with other time formats.
WATCHERS	(String)	A list of string objects denoting the watchers this event is destined for.
EVENT	String	Specifies the type of the event. By definition, EVENT must be one of the event types the watchers have subscribed for using the command presence.subscribe.
DESC	(TYPE BODY)	The description is a list of exactly two strings denoting a presence document and its type. The TYPE parameter specifies the MIME media type of the presence document given in BODY, if applicable. If the MIME media type is not known or not defined, TYPE must be an empty string. The actual event description is contained in the data block BODY, usually a document of type application/pidf+xml.

Table 5.2: Command arguments for presence.notify

A simple filtering mechanism for notification messages is achieved by the argument EVENT. Event types are protocol-specific tokens identifying a specific class of events such as registrations of contact addresses, or changes of the presence status. Subscriptions always refer to a list of events the subscriber is interested in, requesting the event source to send only notifications for the respective event class. Hence, if a client has subscribed for registration events only, no status change notifications would be sent to this client.

The presence.notify command can be used by entities for publishing sampled sensor data or by protocol engines passing received presence status descriptions to the local Mbus. Both scenarios are shown in Figure 5.10.

The picture shows a modular presence server that performs protocol translation and aggregation of presence values. The aggregating entity has subscribed to events generated by sensors in the local Mbus environment. When receiving external presence status notifications, the aggregation engine evaluates any scripts associated with the respective presence status record. If this step yields a status change, an appropriate notification is sent to subscribed watchers—usually entities that convert the given notification to another protocol.

Since many foreign presence protocols do not yet build on the standardized PIDF format to describe the presence status, the command presence.notify can be used to carry any document format that has an official MIME media type registration.[33] As we use the Session Ini-

[33]MIME is short for *Multipurpose Internet Mail Extensions* [RFC2045], a mechanism for identification of standardized content formats. MIME media types are registered with the *Internet Assigned Numbers Authority* (IANA). Existing registrations are listed online at <http://www.iana.org/assignments/media-types/>.

tiation Protocol (SIP) for wide-area event notification, the MIME media type `application/pidf+xml`. will be used together with PIDF documents to describe presence information. For publication of sensor data, implementors may decide to invent local conventions for encoding to minimize protocol overhead. For example, a temperature sensor could send the following Mbus command:

```
presence.notify ("sip:window@office.example.net"
                 "2006-03-09T23:25:16+01:00"
                 ("sip:bob@otherdomain.org" "sip:clark@example.net")
                 "temperature"
                 ("text/plain" <MjAuNzMgKzAuMDI=>))
```

In this example, the message contains the (base64-encoded) string "`20.73 +0.02`" to indicate an environment temperature of 20.73 degrees centigrade which is an increment of 0.02 since the last sample. The event type was chosen to be "`temperature`", and this particular thermometer has the unique identifier `sip:window@office.example.com`. The URI scheme SIP is used only to guarantee a specific URI format. As the sensor has no SIP protocol interface, direct subscriptions to this SIP URI are not possible for external users.

The notification example assumes that previous subscriptions were made for the watchers Bob and Clark, identified by their SIP URIs `sip:bob@otherdomain.org` and `sip:clark@example.net`. Mbus entities conforming to this specification would not forward this event notification to any external watcher other than those specified in this command. An aggregation engine thus could create custom views on this event (e.g. using different MIME media types) for specific watchers.

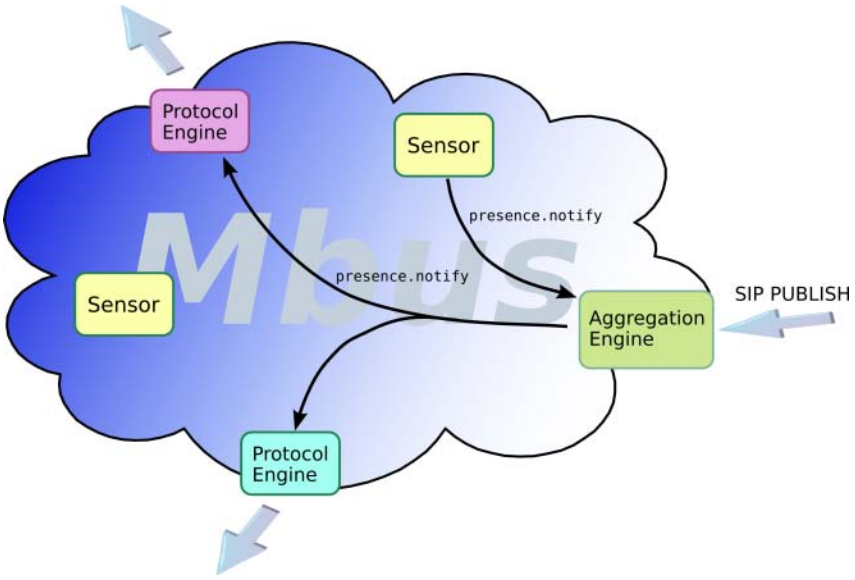

Figure 5.10: Event notifications on the Mbus

A Reliable Publication Command

After collecting the sensor data and aggregation of presence attributes, the resulting status description in general shall be published outside the local Mbus environment. For this purpose,

the presence profile offers a unicast RPC `presence.publish` that takes the same arguments as the event notification. The only difference between both messages is the reliable transmission of the RPC and its corresponding acknowledgement message, `presence.publish.return`.

The result of this command indicates the correct processing of the publication request by the receiving entity. A gateway module that passes the published data over to an external presence server using any wide-area event notification protocol (e.g. SIP or XMPP) will acknowledge the corresponding Mbus request with an `OK` status token immediately after sending out the presence document. Thus, the protocol engine will not wait for a response from external servers where the data has been sent to, as the delay usually would exceed the maximum timeout value of Mbus RPCs.

In case of an error, e.g. if the presence document could not be converted to a document format required by the external protocol, the Mbus entity that received the `presence.publish` request will return an application-specific error token together with a textual description of the actual error condition. Table 5.3 shows the symbolic return values that have been defined in the Mbus presence profile.

Value	Description
`OK`	The request has been processed without error.
`ECONV`	Given presence document could not be converted to the target protocol's native document format.
`EAUTH`	The sending entity is not authorized to publish data for the specified principal.

Table 5.3: Return values for `presence.publish`

The following example for a complete publish request shows the use of application-specific parameters within an Mbus RPC argument list. As explained in the previous section on Mbus interaction models, the first argument of an RPC command contains a unique identifier to correlate a response with its originating request, and a specification of the actual RPC type. In this case, the value "`UNICAST`" is used because `presence.publish` must be handled by exactly one Mbus entity. The second argument of an Mbus RPC contains the list of application-specific parameters. `presence.publish` is called with the parameters listed in Table 5.4.

The application-specific parameters listed here have the same semantics as the arguments of `presence.notify`. The only difference between both commands' arguments lists is the missing watcher list which is not needed by `presence.publish`. An invocation of this command hence may look like the following:

```
presence.publish ((("ID" "18") ("RPC-TYPE" "UNICAST"))
                  ("pres:john@example.com"
                  "2006-03-10T20:36:08-04:00"
                  "presence"
                  ("application/pidf+xml" <....>)))
```

Possible return values for this RPC are shown below. The first example denotes a successful `presence.publish` request, while the second indicates an error caused by an invalid document format. Note that the value of `RPC-STATUS` is "`OK`" since the RPC itself was handled correctly on the Mbus layer.

```
presence.publish.return ((("ID" "18") ("RPC-STATUS" "OK"))
                         ((OK OK "document published") ()))

presence.publish.return ((("ID" "18") ("RPC-STATUS" "OK"))
                         ((ERROR ECONV
                           "E:document format not allowed") ()))
```

`presence.notify` and `presence.publish` offer a simple yet powerful mechanism for push-based publication of presence information. While `presence.notify` is used to send event notifications to watchers that have explicitly subscribed to a particular status record and event type they are interested in, the RPC `presence.publish` is used primarily in pre-configured communication relationships with external presence servers. A gateway implementation could, e.g., use the Mbus address tag `protocol` to detect the presence of Mbus modules that implement a specific presence protocol.

As explicit registration is not always feasible, the presence profile also provides an RPC for explicit retrieval of complete status descriptions as a one-time operation. The following section defines this last command beginning with the prefix "`presence`".

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the principal the event notification is sent for.
`TIME`	`String`	The local time specifying when the event has occurred. The value of `TIME` must be specified in the syntax of [RFC3339] to avoid confusion with other time formats.
`EVENT`	`String`	Specifies the type of the event. By definition, `EVENT` must be one of the event types the watchers have subscribed for using the command `presence.subscribe`.
`DESC`	`(TYPE BODY)`	The description is a list of exactly two strings denoting a presence document and its type. The `TYPE` parameter specifies the MIME media type of the presence document given in `BODY`, if applicable. If the MIME media type is not known or not defined, `TYPE` must be an empty string. The actual event description is contained in the data block `BODY`, usually a document of type `application/pidf+xml`.

Table 5.4: Command arguments for `presence.publish`

Querying Presence Documents

As some applications cannot use the push-based notification of status changes due to local policy or technical constraints, the Mbus presence profile defines an RPC for explicit one-time retrieval of presence information for a particular resource. This command is targeted especially on applications that do not require periodical updates but can also be used for regular polling as Web-based groupware systems typically do.

The Mbus RPC `presence.fetch` must be called with two arguments specifying the status record's identifier and the event type the caller is interested in. Table 5.5 again shows these parameters already known from push-based notification via `presence.notify`.

Name	Type	Description
PRESENTITY	String	Unique identifier of the principal whose status record is requested.
EVENT	String	Specifies an event class the caller is interested in. This argument can be used when the RPC handler is a protocol gateway that translates the Mbus request into an external protocol such as SIP.

Table 5.5: Command arguments for `presence.fetch`

On success, the RPC handler returns a list containing the requested presence description and the unique identifier of its MIME media type. In general, the document's type will be `application/pidf+xml` for standards-compliant PIDF document. If, however, the entity that handles `presence.fetch` requests happens to be a gateway for an external presence protocol, another document type may be used in the result message. The following example illustrates the fetch request together with possible results for this request:

```
presence.fetch ((("ID" "62") ("RPC-TYPE" "UNICAST"))
                ("pres:john@example.com" "presence"))

presence.publish.return ((("ID" "62") ("RPC-STATUS" "OK"))
                         ((OK OK "Success")
                          ("application/pidf+xml" <...>)))

presence.publish.return ((("ID" "62") ("RPC-STATUS" "OK"))
                         ((ERROR ENOTFOUND
                           "E:unknown presentity") ()))
```

The first line shows a fetch-request for the status record identified by `pres:john@example.com`. The event type argument is included for compatibility with the generic SIP notification mechanism defined in [RFC3265]. This way, not only presence information can be retrieved from external (SIP) servers but also registration information (using the event type "`reg`" as defined for SIP in [RFC3680]) or message summaries of a voice mailbox (according to the SIP event type "`message-summary`" defined in [RFC3842]). If the request cannot be processed, the result indicates the error condition as usual. Table 5.6 lists the result codes that have been defined for `presence.fetch` together with a description of their semantics.

As stated before, the fetch operation is intended for one-time retrieval of local resources. When used with a protocol engine as illustrated in Figure 5.11, a timeout may occur during the Mbus RPC caused by high latency times for traffic in the global Internet. Though the expiration timers for RPCs could be increased to overcome this situation, this is not a good idea as it affects any other RPC as well and thus can deteriorate the entire system's performance. Instead, the push-based notification described in Section 5.4.4.1 should be used where possible. To do so, the watcher must have installed a subscription to the particular status record as described in the next section.

Value	Description
OK	The request has been processed without error.
ENOTFOUND	There is no status record associated with the given identifier.
EAUTH	The sending entity is not authorized to access data for the specified principal.

Table 5.6: Return values for `presence.fetch`

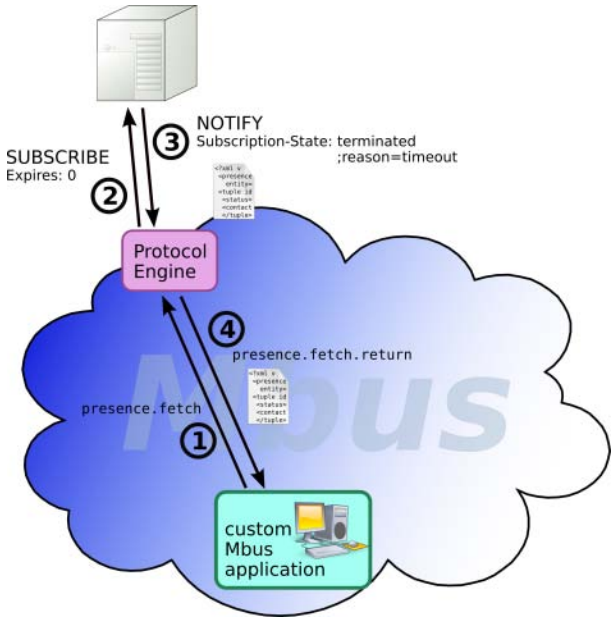

Figure 5.11: Querying information from external resources

5.4.4.2 Managing Subscriptions

Entities that are interested in status changes of a particular principal either must poll the status in regular intervals or subscribe for automatic event notifications. Each subscription is tied to the principal's unique identifier and the types of events that shall be notified. To establish a new subscription or refresh an existing one, the command `presence.subscribe` defined in Section 5.4.4.2 can be used.

The established subscriptions are soft-state and thus expire after some time if not refreshed. If necessary, subscriptions can be removed explicitly using the command `presence.unsubscribe` which is described in Section 5.4.4.2.

Subscribing to an Address

To add or refresh a subscription the Mbus presence profile provides the RPC `presence.subscribe`. The command is invoked with three arguments which are listed in Table 5.7. The first argument is a unique identifier of the principal whose status is to be subscribed, and the second is a list of event types the subscriber is interested in.

An additional list of key/value pairs may be used to specify further options for this subscription. Currently, only two options are defined as shown in Table 5.8. In this list, implementation-specific options could be specified as well, e.g. to install a filter expression for throttling of event notifications depending on their content.

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the status record to subscribe.
`EVENTS`	`(String)`	List of event types for which notifications to the subscribed watchers shall be sent. The event type "`presence`" is used to be notified whenever the principal's presence status has changed.
`OPTIONS`	`((String String))`	A list of key/value pairs specifying optional parameters for the subscription request. Available options are described in Table 5.8.

Table 5.7: Command arguments for `presence.subscribe`

Name	Description
`expiry`	Expiration of subscription in seconds. If this option is not specified, an implementation-specific default will be used, typically 3600 seconds.
`watcher`	A list of unique identifiers of the subscribing watcher. The identifier can be used by an external protocol gateway to forward subscriptions to an aggregation module as described in Section 5.5. If this parameter is not specified, it matches the least specific set of watchers (i.e. the *public* zone).

Table 5.8: Options for `presence.subscribe`

If a new subscription has been established or an existing subscription has been renewed the RPC result will contain the result code `OK`. Otherwise, it will denote the error condition using one of the values specified in Table 5.9.

Note that the event types specified as second argument of the subscribe request are used as filter when sending notification messages. Hence, when subscribing for an event that does not exist, the watcher will never receive a status update from the presence engine. A special value "`*`" can be used as wildcard to subscribe to any event generated by the presence agent.

As for the fetch operation, the subscribe command is applicable only to existing resources. If the first argument of `presence.subscribe` refers to an unknown principal the result code `ENOTFOUND` is returned. This might happen especially when the subscribe command is passed on to an external presence protocol or when referencing a local resource that does not exist. Moreover, if the requesting watcher is not authorized to subscribe to the status record of the specified principal the server may indicate this fact using the error code `EAUTH`.

The action taken for requests for resources that do not exist or for which the subscriber has not sufficient access privileges also depends on the local policy of the presence system. Instead of sending explicit error responses, the server might as well accept the subscription request but never send any status notification (*polite blocking*, see Section 5.5).

Value	Description
OK	The subscription has been established or refreshed.
ENOTFOUND	There is no status record associated with the given identifier.
EAUTH	The sending entity is not authorized to subscribe to the specified principal.
EEXPIRY	An invalid expiration time has been specified.

Table 5.9: Return values for presence.subscribe

Finally, the server may return the error code EEXPIRY to indicate that the given expiration value was invalid. The textual reason string may give additional hints on the actual error condition, e.g. to distinguish between invalid syntax or an expiration interval that is not allowed.

As a result of sending an error result, the subscription is not established or updated, respectively. Moreover, the expiration timer is updated only if the subscription is new or if the remaining lifetime for an existing subscription is less than the value given with the request, if any. presence.subscribe thus cannot be used to shorten the lifetime of an existing subscription. This can be done only with the unsubscribe command defined in the following subsection.

Removing a Subscription

To remove an existing subscription for a specific event type the RPC presence.unsubscribe must be used. The arguments for this command are the same as were described for presence.subscribe in the previous section. An overview of the arguments and their meaning is given in Table 5.7.

Similar to presence.subscribe, only existing subscriptions for the status record with the given identifier can be removed. If an unsubscribe request is sent for an unknown status record, the server returns the result code ENOTFOUND. If more than one event type was specified as second argument of presence.unsubscribe or the argument's value was "*" the return code will be OK if at least one existing subscription was removed successfully. This way, applications do not need to track if a particular subscription already has expired before sending an unsubscribe request for multiple events.

presence.unsubscribe might be called with an optional parameter watcher included in the command's third argument. The option's value indicates the identity of the watcher whose subscription is to be removed. If the watcher is not authorized to remove this subscription, the result code EAUTH will be returned. On successful removal of a subscription, the command's result will be OK.

5.4.4.3 Controlling the Aggregation Engine

In addition to presence-specific commands, we define a number of commands for controlling the presence aggregation engine. The Mbus commands in this section are sent to the address (module:presence type:aggregation) that identifies the presence aggregation server. An application-specific command prefix "pam" also indicates the intention of manipulating the internal status of the aggregation engine rather than the presence-information that is managed by that module.

Modification of Configuration Parameters

The commands described in this section provide means for modifying the configuration of the presence aggregation engine In contrast to the generic RPCs defined in [Kut01] for accessing global properties of Mbus modules the `pam` family of commands refers to configuration parameters associated with a specific status record. The commands therefore take the corresponding principal's unique identifier additional argument.

The value of a configuration parameter can be set with `pam.parameter.set`. Table 5.10 lists the arguments of this RPC. The first argument is the identifier of the principal whose configuration is to be changed as stated above. The second argument is a list of key/value pairs specifying the parameters and their new values to be set in an atomic operation. If an error occurs for any of these parameters the module's configuration will not be changed at all to preserve a consistent state.

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the local resource the parameter to be modified is associated with.
`PARAMS`	`((String String))`	A list of key/value-pairs specifying the parameters to be changed together with their new values.

Table 5.10: Command arguments for `pam.parameter.set`

If, for some reason, the operation cannot be performed, the return of the RPC is `ERROR` and the reason string describes the first error that was encountered. The following example shows how to use the command `pam.parameter.set` and a possible result message.

```
pam.parameter.set ((("ID" "711") ("RPC-TYPE" "UNICAST"))
                   ("pres:carl@example.net"
                    (("SYSTEM.MAX_HISTORY" "4")
                     ("SYSTEM.MAX_CPU" "abc"))))

pam.parameter.set.return ((("ID" "711") ("RPC-STATUS" "OK"))
  ((ERROR ERROR "E:invalid data type (expected numeric value)")
   ()))
```

The previous example shows a request for setting two system parameters of the presence aggregation engine to the given values, using the application-specific namespace of the presence aggregation language documented in Chapter 6. Here, the properties `MAX_HISTORY` and `MAX_CPU` of the object `SYSTEM` should be changed to the value "`4`" and "`abc`", respectively. As both properties accept numeric values only, the receiver will return an error result for the second parameter change request.

To retrieve the current value of a parameter, only the status record's identifier and the parameter name must be specified with the RPC `pam.parameter.get` as shown in Table 5.11. If the parameter was found and the sender has read access, its current value will be returned as Mbus string. Possible return codes are listed in Table 5.12.

The following example shows a get-request with its corresponding answer. The numeric value of `SYSTEM.MAX_CPU` has been converted to a string for inclusion in the result message.

```
pam.parameter.get ((("ID" "712") ("RPC-TYPE" "UNICAST"))
                   ("pres:carl@example.net" "SYSTEM.MAX_CPU"))

pam.parameter.get.return ((("ID" "712") ("RPC-STATUS" "OK"))
                          ((OK OK "Success") ("30")))
```

To obtain an overview of the available configuration parameters associated with a single status record, we have defined the RPC `pam.parameter.list`. The command takes the principal's identifier as its only argument and returns a list of key/value pairs containing the known parameters with their current values. An error code is returned only if an unknown status record has been requested or if the sender is not authorized to access that information. The applicable result codes have been described for `pam.parameter.get` in Table 5.12.

Name	Type	Description
PRESENTITY	String	Unique identifier of the local resource the parameter to be retrieved is associated with.
KEY	String	Name of the configuration parameter to be read.

Table 5.11: Command arguments for `pam.parameter.get`

The commands for retrieval and modification of parameters that have been described so far do not take into account the actual data types of the PAL objects that are passed around. To overcome this problem, the Mbus command `pam.parameter.describe` returns detailed information about a given parameter name. The RPC is invoked with the status record's unique identifier and the parameter name and returns a list of key/value pairs on success. The command's error codes resemble the possible results of `pam.parameter.get` as shown in Table 5.12.

Value	Description
OK	The request has been processed without error. The value of the requested variable is returned as Mbus string in the result part of the return command.
ENOTFOUND	The requested parameter was not found.
EAUTH	The sender is not authorized to access the requested parameter.

Table 5.12: Return values for `pam.parameter.get`

The following example illustrates the invocation of `pam.parameter.describe` and a possible result containing a description of the given configuration parameter. This description is comprised of its access permissions, data type, and a textual description of the parameter's semantics. An overview of the keywords contained in the result message together with their meaning is given in Table 5.13.

```
pam.parameter.describe ((("ID" "4812") ("RPC-TYPE" "UNICAST"))
  ("sip:john@example.com" "SYSTEM.MAX_HISTORY"))
```

```
pam.parameter.describe.return
  ((("ID" "4812") ("RPC-STATUS" "OK"))
   (OK OK "Success")
   (("access" "readwrite") ("type" "number")
    ("description"
     "Maximum number of entries in execution history.")))
```

With the commands described so far, basic configuration parameters of a presence aggregation server can be modified such as the system parameters that are shown in Section 6.3.2.2. Another important aspect of server configuration is the management of user-specific aggregation scripts. The following section gives a detailed overview of the commands that have been defined specifically for this purpose

Value	Description
`access`	Specifies the access permissions for the given key `KEY`. The value "`read`" denotes read-only access, "`write`" denotes write-only access, and "`readwrite`" gives no restrictions at all.
`type`	The parameter's data type as defined in Section 6.3.2.2. Currently, this must be one of `string`, `number`, `boolean`. Note that the parameter values are represented as strings in message bus commands, independent of the semantic type listed here.
`description`	Contains a short description of the parameter `KEY` to be used for display, e.g. in an interactive WWW interface.

Table 5.13: Return values for `pam.parameter.describe`

Managing Aggregation Scripts

Scripts can be associated with a presentity in multiple ways depending on the actual implementation of the presence server. A common solution is to provide a web interface using the `PUT` method of HTTP to upload a document into the script storage. Other protocols such as the *XML Configuration Access Protocol* [XCAP] could be used as well.

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the principal the script is associated with.
`SCRIPT`	`String`	The actual PAL script, encoded as UTF-8.
`OPTIONS`	`((String String))`	A list of key/value pairs specifying optional parameters for the script. Available options are described in Table 5.15.

Table 5.14: Command arguments for `pam.script.add`

The only requirement on the mechanism for script upload imposed by this specification is the authentication of clients and the preservation of the script order. In particular, the client user who is requesting the association of a presentity with a new script must be identified and

the permissions to store scripts locally must be checked using the policy framework defined in Section 5.5.

In this section, we define a set of Mbus RPCs to add user-specific scripts to a presentity, retrieve information about existing scripts, and to modify or delete them. The commands are prefixed with the string `pam.script` indicating their tight relationship with the aggregation engine.

Most important, clients that publish PAL scripts must be authorized to install PAL scripts on behalf of the corresponding principal. The PAL engine therefore must verify the client's identity, e.g. using the authentication mechanisms provided by the underlying transport protocol. In the case of HTTP upload, the uploading client should be authenticated according to [RFC2617] as most HTTP clients at least support HTTP basic authentication. In addition, it is recommended to transfer the script's source-code encrypted over a secure channel to avoid disclosure of blocking rules. For TCP-based connections, the use of TLS [RFC2246] is recommended.

Name	Description
`priority`	A numeric value greater or equal zero that specifies the relative priority of the given script. Once a script has been uploaded, it is only replaced with another script if the new script's priority value is greater or equal to the value of the old script.
`filter`	A filter expression, e.g. to cause the PAM to collect several subsequent events of a specific type before the corresponding PAL script is evaluated. The use of filter expressions can help to reduce CPU time to handle high-volume event sources.

Table 5.15: Options for `pam.script.add`

Value	Description
`OK`	The request has been processed without error. In this case, the RPC returns a pair `(SCRIPT_ID String)` that contains a unique identifier for the uploaded script. The identifier can be used to modify or delete this script.
`ECONV`	The given script uses a character encoding that the script interpreter was not able to parse.
`EAUTH`	The sending entity is not authorized to upload a script for the specified principal.
`EPRIO`	The script's priority was too low to replace an existing script.

Table 5.16: Return values for `pam.script.add`

Before the script is installed, the PAL engine must parse it for syntactic errors. The PAM may perform additional checks for semantic constraints like avoidance of infinite recursive function calls. If these optional checks are performed and any of these semantics checks fails, the script must be rejected, and a descriptive error output according to the publish mechanism being used for transferring the script to the server should be sent to the initiating client. Otherwise, a positive response is to be sent.

If the initial checks succeeded, the new PAL script can be installed for the corresponding presentity. As there might already be a script installed for the particular presentity, the PAL engine has to decide whether to replace the existing script or to chain with the new one. For the PAL version described here, the existing script is always replaced. An existing script must not be replaced while it is executed. Instead, the PAL engine has to wait until the script run is finished and outgoing notifications have been sent.

When replacing a script, any timers or static variables that exist for that script are removed, history lists are cleared from all events that have already been processed and from old events that have been collected due to implementation-specific event-throttling functions. The interpreter must always guarantee a clean environment when a script is executed for the first time.

A PAL script is always associated with a specific presentity and therefore is associated with incoming update notifications for that particular presentity. A client may change the script that is associated with a presentity using the Mbus commands `pam.script.add`, `pam.script.modify` and `pam.script.delete`—provided it is authorized to do so.

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the presentity whose script set is to be changed. This parameter has the same semantics as in `pam.script.add`.
`SCRIPT_ID`	`String`	Identifier of the script to be modified as returned by `pam.script.add`.
`OPTIONS`	`((String String))`	A list of key/value pairs specifying optional parameters for the script. Options that are not specified remain unchanged. This command provides the same options that are available for `pam.script.add` as specified in Table 5.15.
`SCRIPT?`	`String`	An optional parameter containing a new script to be used as replacement for the script that is currently identified `SCRIPT_ID`. The script's identifier will not change with this operation.

Table 5.17: Command arguments for `pam.script.modify`

Value	Description
`OK`	The request has been processed without error.
`ECONV`	The given script uses a character encoding that the script interpreter was not able to parse.
`EAUTH`	The sending entity is not authorized to modify the script of the specified principal.
`EPRIO`	The script's priority was too low to replace an existing script.

Table 5.18: Return values for `pam.script.modify`

The commands described in this section must be sent by a client only if the action that triggered this command was caused by a user who is authorized to modify the presentity's PAL

scripts. For example, a WWW interface must not pass uploaded scripts to the PAM unless the requesting user has successfully been authenticated and is permitted to add a PAL script. In typical scenarios, this is only the case for the principal being represented by the presentity.

The commands specified here are Mbus RPC commands, i.e. there is always an Mbus command sent in response to the request. The most important command is `pam.script.add` that is used to add a new script or replace an existing one. The command's parameters are described in Table 5.14. With the option `priority`, the replacement of existing scripts can be controlled. If this value is less than the corresponding value of an existing script, the old one will not be replaced. In this case, the command returns the error `EPRIO` to indicate that the script has not been uploaded.

An existing script may be modified with the command `pam.script.modify`, with the arguments described in Table 5.17. The most common use is to replace a script with a new script. In addition, priority values may be changed with this script. The command will return one of the values specified in Table 5.18 as seen for `pam.script.add`.

To remove an existing script without replacing it with a new one can be done with the command `pam.script.delete`. The only parameter besides the principal's unique identifier is the identifier of the script that has to be deleted as specified in Table 5.19. On success, this operation will return `OK`, otherwise a descriptive error code will be returned. Currently, only the authentication error `EAUTH` is defined, as shown in Table 5.20.

Name	Type	Description
`PRESENTITY`	`String`	Unique identifier of the presentity whose script should be deleted.
`SCRIPT_ID`	`String`	The identifier of the script to be removed. It is not an error if no script is associated with this identifier.

Table 5.19: Command arguments for `pam.script.delete`

Value	Description
`OK`	The request has been processed without error.
`EAUTH`	The sending entity is not authorized to delete the script of the specified principal.

Table 5.20: Return values for `pam.script.delete`

As interactive Web interfaces must be able to provide a list of available scripts, the command `pam.script.list` returns the identifier of the script associated with the given principal (see Table 5.21). If no script is available, an empty `OK` response will be returned. Otherwise, the response contains a pair with the key `SCRIPT_ID` and the script's identifier as value as specified for the command `pam.script.delete` in Table 5.20.

The final command allows for retrieval of a script with a certain identifier, see Table 5.22. If the specified string is not a valid identifier for an existing script of the given principal, the response will be empty. Otherwise, it will contain a pair `(SCRIPT String)` with the associated script encoded in UTF-8.

Name	Type	Description
PRESENTITY	String	Unique identifier of the presentity the script identifiers should be retrieved from.

Table 5.21: Command arguments for pam.script.list

Name	Type	Description
PRESENTITY	String	Unique identifier of the presentity whose script is to be retrieved.
SCRIPT_ID	String	The identifier of the script that is to be retrieved.

Table 5.22: Command arguments for pam.script.fetch

With the commands specified here, simple management of the aggregation engine is possible. In addition to script upload, deletion and modification, two commands are provided to facilitate creation of Web interfaces for script management. For any command to be successful, the sending client must have been authenticated and must be authorized to perform the requested operation. The permissions are checked by the policy engine as described in the following section.

5.5 A Policy Framework for Presence Aggregation

An essential aspect of every presence service is the handling of sensitive data. As discussed in Section 3.4, the generation of watcher-specific presence documents requires a policy framework that defines the definition of authorization classes to restrict access to certain parts of presence documents. The authorization policies defined by a presence source must be honored by any entity along the path that a published presence document takes through the network.

In this section, we discuss a minimal authorization framework we have defined to protect sensitive presence information from being disclosed to unauthorized watchers. Aggregation servers are advised to filter out any presence attribute that the receiver of status update notification is not permitted to access before the notification message is sent. Policy documents that define watcher authorization classes can be passed to the presence server either through its Web-based configuration interface or may be included in a presence document.

Example 5.5.1 again shows the example policy document contained in Section 3.4 to illustrate the definition of watcher classes. The namespace declaration reveals that the language for definition of authorization classes is part of the PIDF-XY extensions we have discussed in Section 5.3.

The example document defines several sets of watchers that are identified by the URI used when authenticating for a presence subscription. Each class definition is enclosed in an element of type authclass and has a unique name that can be used to reference the particular class from within the document. A watcher is specified in the contents of the element type authelem as a regular expression that is matched against a subscriber's URI. To avoid partial matching of URIs, the given expression is implicitly anchored to the beginning and the end of the string to match. Otherwise, an expression like alice@example\.net would match a substring of

a much longer URI as, e.g., malice@example.net.intruder.com which is typically not intended.

As the definition of watcher classes using regular expressions is a tedious task for more complex scenarios, authorization class definitions may be included by reference as shown for the definition of class family. Here, the authorization class private is referenced using the element type authref. The inclusion is evaluated as if the definitions of class private were literally included in the content of the element family.

Example 5.5.1: Specification of authorization classes

```
<authdef xmlns="urn:ietf:params:xml:ns:pidfxy">
  <authclass name="private">
    <authelem>alice@example\.net</authelem>
  </authclass>

  <authclass name="friends">
    <authelem>Bob@otherdomain\.com</authelem>
    <authelem>[^@]*@mysportsclub\.com</authelem>
    <authelem>jill@example\.net</authelem>
  </authclass>

  <authclass name="family">
    <authref name="private" />
    <authelem>philip@example\.net</authelem>
    <authelem>edward@example\.net</authelem>
    <authelem>anne@example\.net</authelem>
  </authclass>

  <authclass name="protected">
    <authref name="friends" />
    <authref name="family" />
  </authclass>

  <authclass name="public">
    <set-difference>
      <authref name="protected" />
      <authelem>.*</authelem>
    </set-difference>
  </authclass>
</authdef>
```

To exclude elements from the set of character strings defined by a regular expression (or the union of those sets specified by multiple authelem elements), we have defined the element type set-difference to specify the symmetric difference of two authorization classes. The class public hence contains any URI that is not contained in the class protected.

A presence aggregation server that supports this authorization mechanism must process the PIDF-XY element type authclass defined in Section 5.3 together with the authorization class definition. If the contents of authclass refer to an existing class definition, the channel description containing this element must be published only to watchers that are contained in this class. For example, the channel description in Example 5.3.1 references the authorization class protected. When Bob@otherdomain.com subscribes to this resource, his URI will be matched against the definition of the authorization class protected. As this includes the definition of the class friends, the status description will be published to Bob. If, in contrast the user mallory@example.com subscribes to this presence record, the aggregation engine finds the URI to be contained in the authorization class public and thus, the channel description

will be deleted from the output. As presence documents must not be empty, no notification will be sent to Mallory at all.

5.6 Summary

This chapter has described the basic architecture of our presence aggregation service and discussed alternative design options where available. As the aggregation process is independent of the used protocol for dissemination of presence information, the event notification mechanism defined in [RFC3265] can be used for distribution of presence documents without any changes. SIP servers that support the aggregation of presence documents can be integrated easily with the public SIP infrastructure. The only differences between non-aggregating servers are the processing of incoming presence documents and the creation of watcher-specific output.

The processing of received presence documents may require the transformation of custom presence formats into an internal representation that is compatible to the standardized *presence information data format* (PIDF) defined in [RFC3863]. For a better support of presence aggregation, the enhanced data format PIDF-XY could be used. Besides augmented status descriptions, PIDF-XY enables the re-calculation of the presence status under specific conditions. A degradation function was defined to simulate the aging of presence attributes, i.e. a decreasing exactness of the given value over time. Further PIDF-XY attributes can be used to qualify the presence values to foster the notion of fuzziness that is inherent to presence information.

In addition to the enhanced data format for presence information, an Mbus application profile has been defined to control the components within a local presence environment. Due to its flexible addressing scheme and the dynamic detection of new entities, the lightweight Message bus protocol defined in [RFC3259] provides an appropriate middleware for building decoupled applications such as the personal presence server.

Finally, we have defined a minimal authorization framework that can be used to specify distinct classes of watchers to be referenced in PIDF-XY channel descriptions. A channel description containing a class reference will be published only to subscribed members of this particular class.

Chapter 6

Using ECMAScript for Presence Aggregation

The service architecture presented in the previous chapter provides a flexible platform for processing information from multiple presence sources. Aggregation specifications that control the aggregation process can be placed at any server that generates or forwards event notifications. In this chapter, we describe the *presence aggregation language* (PAL) for authoring these documents based on the object-based *ECMAScript* [ECMA-262] programming language. First, we give a short introduction of ECMAScript's main language features. We then describe the integration of PAL-specific data types and operations into the language, followed by a detailed explanation of the language constructs' semantics. The chapter ends with a number of example scripts that show how to use this language. An overview of the presence-specific function library we have developed is shown in Appendix D.

6.1 Language Characteristics

The ECMAScript language has evolved as a standardized version of proprietary programming languages that have been invented in the 1990s by Microsoft and Netscape to control the dynamic behavior of Web browsers. The 3rd edition from 1999 defined in [ECMA-262] is the standardized subset of JavaScript 1.5 which is typically used in modern Web browsers that support client-side scripting. According to a recent survey[34] more than a third of the web developers being queried for their preferred scripting technologies use JavaScript libraries in their projects. This result indicates the growing importance of JavaScript as a flexible language for development of mobile code that is transferred through the network.

We have selected ECMAScript as the core language for presence aggregation as it combines features of modern programming languages with a clean language design. Being an object-based language, ECMAScript has the notion of *objects* with specific properties. Unlike object-oriented programming languages such as Java or C++, objects are not instances of classes. Instead, object structures have a specific property `prototype` that may contain a reference to a parent object. When an object property is accessed, the interpreter first tries to find this property

[34] SitePoint Pty Ltd. and Ektron, Inc.: The State of Web Development 2006/2007. Available at <`http://www.sitepoint.com/launch/survey06/`>.

in the object's property list. If not found, the search continues at the prototype object, achieving a simple form of inheritance.

The language features defined in [ECMA-262] already meet most of the requirements stated in Section 3.2.4, i.e. ECMAScript provides a boolean algebra, conditional execution of statements, mathematical functions, and containers for storing objects at runtime. In addition, the core language provides an object `Date` that represents time data, `String` objects for storing character sequences, and an object `RegExp` for string matching using regular expressions as well as several objects used for parsing and evaluation of scripts and handling of errors.

To fulfil the remaining requirements of Section 3.2.4, we have added the native objects described in Section 6.2 to provide functions and properties that are specific to presence aggregation and thus not part of the ECMAScript language. During our work, we have also realized that the container objects offered by ECMAScript lack support for collections that eliminate multiple entries which are treated as equal (for simplicity, we use the mathematical term *set* when referring to this type of container). As the presence aggregation language makes extensive use of sets and associative bindings of elements in container structures, we have decided to define two native objects implementing these containers. First, the object `Set` is defined, providing common operators known from mathematic set algebra. The second object defined here, `ASet`, is a specialization of a set that provides a mapping from unique keys to arbitrary values, with additional properties specific to associative sets.

6.1.1 Set

The object type `Set` in PAL is a container for arbitrary objects that can be pairwise compared for equality. The order of objects contained in a `Set` object is undefined. To manipulate `Set` objects, the following functions are defined:

`isElement` Takes an arbitrary object as argument and returns true if the given object is contained in the set. To check whether an object is contained in the set, it is compared to the elements within the set using their specific equality operator. Hence, type coercion may be used e.g. to identify objects by their unique id (as in the case of the object types `Channel` or `Presentity`).

`addElement` Takes one or more objects and adds them to the set, traversing the argument list from left to right. Before an object is added to the set, a containment check is performed on that particular object using the function `isElement`. If there exists an object in the set that is equal to the one being added, the former is being replaced. The function returns `true` if the object has been added to the existing set. Otherwise, the value `false` is returned.

`removeElement` Removes any object from the set that is equal to any object in the argument list. The function returns `true` if and only if an object was removed from the set.

`map` The argument is a `Function` object that takes exactly one parameter. The function is applied in sequence to every member of this set, with the results being added to a new `Set` object. This `Set` object then is returned as result of the function `map`.

`filter` The argument is a `Function` object that takes exactly one parameter and returns a `Set` containing the filtered objects. To determine the result, the given function is applied in sequence to every member of the set. The members for which the function yields true are added to a new `Set`, forming the result of the function filter.

`foldl` The function takes two arguments: A binary function and an arbitrary object. The binary function is applied to every member of the set, with the current element being passed to this functions as its first argument. For the first evaluation step, the object given as second argument to the function `foldl` is used as second parameter. For any subsequent step, the result of the preceding function evaluation of the given operator is used as second parameter.

Sets can be created in several ways: The object constructor creates an empty `Set` object and adds any object given in the constructor's argument list. For existing `Set` objects, new sets can be created with the following global functions:

`union` Takes zero or more `Set` objects as parameters and returns a new `Set` object containing every object that is an element of one of the sets in the argument list. The argument list is processed from left to right, using the function `addElement` of the new set for any member of that particular argument.

`intersection` Creates a new `Set` object filled with those objects, that are contained in every `Set` in the argument list.

`difference` Creates a new `Set` object containing those elements of the first argument that are not contained in any of the sets denoted by the remaining arguments.

6.1.2 Associative Set

`ASet` objects (associative sets) in PAL contain only `Arrays` that have exactly two elements (hereafter being called "pair"). A pair denotes a binding from its first component (called "key") to its second component (the "value"). Two pairs in this context are equal when they have equal keys, independent of the pairs' second component.

In addition to the set operators defined above, the following functions exist:

`assoc` Takes a key and a value as argument and adds a new pair (key, value) to the `ASet` object. As any existing pair with the same key is replaced by this operation, a new binding is created.

get Takes a single argument treated as key to search among the existing bindings. If the `ASet` object contains a pair with this argument being the key, its value is returned as result of the function `get`. If no such pair exists, the function returns the undefined value.

defined Takes a single argument treated as key to search among the existing bindings. The function returns true if and only if the `ASet` object contains a pair with the given key,

delete Takes a single argument treated as key to search among the existing bindings. If found, the pair identified by this key is removed from the set.

6.2 Native Objects

The internal representation of presence information is based on a set of *native objects*—objects supplied by the runtime environment—that implement the data model defined in Section 5.3. The objects are used by the PAL processor to construct the input set for script execution and to generate status notifications from the output set returned by a PAL script. Thus, the entire aggregation process is centered around the internal object representation of presence documents. In addition, the PAL runtime environment provides native objects to control script execution using external triggers and container objects with specific methods to access the items therein.

6.2.1 Abstract Representation of Presence Information Documents

The abstract representation of presence information documents in PAL uses three types of complex objects: `Presentity`, `Channel`, and `Attribute`. With these objects being used as prototype, any presence information document that is semantically equivalent to PIDF [RFC3863] may be expressed in a PAL script. The mapping function between PIDF and the PAL language also includes PIDF extensions such as RPID [RFC4480] on a syntactical basis. However, to preserve simple access methods, the extension elements recognized by a PAL processor are restricted to elements that have only textual contents. In addition, the application-specific data types defined in Section 5.3 are supported.

For a smooth integration with ECMAScript character strings and to ensure compatibility with XML-based presence document formats, the internal representation of presence information uses the *Universal Character Set* (UCS) [ISO93]. Following BCP 70 [RFC3470] on the use of XML within IETF protocols, a PAL implementation should output documents in UTF-8 encoding by default. The original input encoding may be stored in the meta-data set of the corresponding `Presentity` object.

6.2.1.1 Presentity

A `Presentity` object represents the root element of a PIDF document, i.e. an element of type `presence` in the PIDF namespace `urn:ietf:params:xml:ns:pidf`. The contents of the

required attribute `entity` are used as object identifier, as it is unique among the set of existing presentities by definition.

The constructor of `Presentity` is invoked with a string denoting a unique identifier for that object. This should be the URI of the corresponding principal, e.g. `"pres:john@example.com"`. Table 6.1 gives an overview of the PAL-specific properties of the `Presentity` object.

Property	Description
`String entity`	Contains the principal's URI.
`Channel[] channel`	An array of `Channel` objects representing the presence tuples of a PIDF document.
`ASet auth`	An associative set with authorization information for this presence document.
`Channel getChannel(`*`ID`*`)`	Searches channel list of current `Presentity` object for a channel with identifier *`ID`*.
`Null setChannel(` *`ID`*`,` *`ch`*`)`	Sets the channel with identifier *`ID`* to the value *`ch`*. If the presentity already contained a channel *`ID`* it is replaced by *`ch`*, otherwise *`ch`* is added to the object's channel list. In any case, *`ch`* must be a valid `Channel` object.

Table 6.1: Properties of `Presentity`

The `channel` property of `Presentity` reflects the channel information given in a corresponding presence document hence the array contents are in the same order as the presence tuples in PIDF. This is especially important for specific extension elements or certain presence clients that are only partially PIDF-compliant in that only the first channel description is processed. When the PAL engine generates a PIDF-document from the internal representation, the relative order defined in PIDF [RFC3863] must be honored.

Authorization class definitions as defined in Section 5.3 are parsed into an associative set with the identifiers of the defined authorization classes being used as keys and the class definitions as values. The class definitions use set operations that are mapped to PAL set operations. Authorization class definitions are part of the meta-data of the `Presentity` object and can be accessed via the property `auth`.

6.2.1.2 Channel

Abstract specification of a communication channel provided by a specific presentity. `Channel` objects have a unique identifier within the principal's set of channels. If the PAL engine consumes or creates PIDF-compliant output documents `Channel` objects represent the `tuple` elements of a PIDF document. The channels' components then correspond to the presence attributes contained in a presence tuple. In particular, the attribute `id` is used as channel identifier. Its value can be accessed via the property `id` of a `Channel` object. All PAL-specific properties of `Channel` are listed in Table 6.2.

In addition to these pre-defined properties the `Channel` object contains the `Attribute` objects generated from the extension elements of a presence document. Currently, these must have disjunct names which are used as property names in the `Channel` object. Thus, an element

mood within the presence tuple would be parsed into an Attribute object accessible from the Channel object via the property mood.

Property	Description
String id	Contains the channel identifier. Its value is equivalent to the value of the attribute id in a PIDF presence tuple.
String contact	Contact information of the channel. Usually, this is a device-specific URI such as sip:john@vbox.example.com.
String status	Status of this communication channel according to RFC 2778. This property's value is either open or close.
String substatus	Extended status value from an application-specific vocabulary.
Number priority	Contains a numeric priority value that can be used by clients to order the communication channels listed in a presence document by their relative priority.
String authclass	The authorization class of the presentity as specified in the PIDF-XY element authclass.
String getType(*ID*)	Returns the contents of the property type for the attribute with the identifier *ID*. If no such property exists the function returns the Null value.

Table 6.2: Properties of Channel

6.2.1.3 Attribute

Attribute objects represent presence traits such as availability status, contact address, modification time, etc. An attribute always has a name, a type, and a value. While the type and value are contained explicitly in the property list of an Attribute object, its name is implicitly given by the property of a Channel object containing the Attribute (see Example 6.2.1). Optionally, the Attribute object can also have a name of an extension namespace to be used when generating a PIDF document. The PAL-specific properties of Attribute objects are shown in Table 6.3.

Decay functions are of major importance in presence aggregation systems as they add a fuzzy component to the static rule-set. Given an Attribute object that represents some sensor sample taken at a time t_0 (specified as value of property timestamp), the exactness of that attribute at that point in time is 1.0. With time passing, this sample gets less reliable, but may be good enough to be used to determine the aggregation result. After a certain time, however, the degradation of this value is significant, and thus the presence status should be calculated anew.

To achieve this behavior, PAL provides an adaptive mechanism to re-calculate the exactness of attribute values, and to trigger the evaluation of the whole script whenever the value gets below an attribute-specific threshold. In alignment with the enhanced presence document information format described in Section 5.3, the decay function is specified as symbolic value of the property decay in an Attribute object. Depending on the function type, additional parameters may be specified to fine-tune the function characteristics.

Property	Description
`String type`	The attribute type as defined in Section 5.3.
`Object value`	The attribute value according to its type. A PAL engine may automatically convert between arbitrary PAL objects and types defined in Section 5.3 (e.g. map numeric values to strings and vice versa).
`String decay`	For compatibility with the format defined in Section 5.3 the decay property is a string that contains a symbolic representation of the decay-function for this attribute value. The function symbol is one of `const`, `linear`, or `log`, optionally followed by a parameter list enclosed in parentheses specifying the degradation factor according to the definitions of Section 5.2.3.
`Number threshold`	A numeric value between `0` and `1.0` that specifies when the recalculation of the presence status should be triggered, depending on the specified `decay` function.
`Date timestamp`	A `Date` object representing the exact time when the given value was observed by the sender of this presence information document.

Table 6.3: Properties of `Attribute`

Example 6.2.1 shows the generation of a presence document containing appropriate decay values for a typing user. The value for the property `type` of an `Attribute` object is usually inferred from the type of the value given in the object's constructor. As seen for the attribute `type`, the type (i.e. the property `type.type`) must be set explicitly as token lists are not recognized at the language level of ECMAScript.

The document created from this PAL specification may look as shown below. Note that the timestamp for the attribute `typing` was automatically set to the point in time when the attribute has been created. As this value is specifically important for attributes where the `decay` property was specified, the output generator suppresses the date specification for constant attributes. A timestamp element specifying the creation time of this channel is added automatically.

```
<presence entity="pres:alice@example.net"
              xmlns="urn:ietf:params:xml:ns:pidf"
              xmlns:pidfxy="urn:ietf:params:xml:ns:pidfxy">

  <tuple id="t18">
    <status>
      <basic>OPEN</basic>
      <pidfxy:name>Jabber</pidfxy:name>
      <pidfxy:type type="tokenlist">COMMUNICATION, INTERACTIVE, TEXT</pidfxy:type>
      <pidfxy:authclass type="string">protected</pidfxy:authclass>
      <pidfxy:ownership type="number">1.0</pidfxy:ownership>
      <pidfxy:typing type="boolean" timestamp="2006-10-01T17:13:06+02:00"
          decay="linear(500)" threshold="0.4">true</pidfxy:typing>
    </status>
    <contact priority="0.8">
       sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE
    </contact>
    <timestamp>2006-10-01T17:13:06+02:00</timestamp>
  </tuple>

<presence>
```

Example 6.2.1: A simple PIDF-XY document

```
var result = new Presentity("pres:alice@example.net")
    jabber = new Channel("jabber");

jabber.contact          =
  new Attribute("sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE");
jabber.priority         = new Attribute(0.8);

jabber.name             = new Attribute("Jabber");
jabber.status           = new Attribute("open");
jabber.type             = new Attribute("COMMUNICATION, INTERACTIVE, TEXT");
jabber.type.type        = "tokenlist";
jabber.authclass        = new Attribute("protected");
jabber.ownership        = new Attribute(1.0);

jabber.typing           = new Attribute(true);
jabber.typing.decay     = "linear(500)";
jabber.typing.threshold = "0.4";

result.setChannel("t18");
```

6.2.2 System-generated Events

In Section 5.2, we have motivated the need for meta-data associated with presence status values, most notably a decay function denoting the degradation of the given information. Besides a set of parameters to describe the characteristics of that function, the meta-data may include a threshold value to control aggregation of attributes in transformation functions and to smoothen a displayed presence graph.

For any of these tasks, the principal's presence status has to be re-calculated once in a while. The presence aggregation engine supports this with an object type `Trigger` that can be used like triggers in active database systems (see [CarDay89] for a detailed description of this concept). A trigger is instantiated with a callback function and a predicate that causes the function to be executed when it becomes true for the first time. Two types of triggers are defined in this version of PAL: A timer fires after a given time span has passed. The other type, `Trigger`, is parameterized with a callback function, the degradation function and its threshold value. It will fire as soon as the exactness value has reached the threshold, executing the given callback function.

Using these objects, PAL scripts can register function closures as *callbacks* that are executed as soon as the specified condition yields true. Thus, two types of events can be used to invoke user-specified functions, i.e. timer expiration and passing of a given threshold value. Both types of registration interfaces are described in the following sections.

6.2.2.1 Timer

Timers can be used to re-start script execution after a given period of time has passed. The timer definition of PAL is aligned to the common practice of JavaScript runtime environments of most Web browsers, i.e. the interpreter provides a function `setTimeout` that allows to register a callback function with a specified timeout value. As a result, only one timer can be active.

The object returned by `setTimeout` can be used to remove an existing timer before is has

fired. The functions available for timer manipulation are listed in Appendix D, together with examples for their usage in aggregation specifications.

6.2.2.2 Trigger

To implement callback-functions triggered by passing a certain threshold of decay functions, PAL provides the object type `Trigger`. A `Trigger` object has the properties listed in Table 6.4.

Property	Description
`Function callback`	Function object that is invoked when a trigger is expired. The respective `Trigger` object is passed as argument to that function, allowing direct manipulations of the object.
`String status`	A string specifying the trigger's status. Possible values are listed in Table 6.5.
`Attribute attribute`	The `Attribute` object to test. This property is set by the constructor for the governing `Trigger` object at instantiation time.
`Number test`	A numeric threshold value between `0` and `1` that causes the callback function to be evaluated as soon as the specified attribute's exactness value gets below.

Table 6.4: Properties of `Trigger`

Value	Description
`run`	Used to re-start a timer that has been manually stopped but not removed from the system's timer
`stop`	Stops a timer object without removing it from the system's timer list. This flag is typically used to suspend a timer until another event has occurred on which the timer depends.
`active`	Indicates that the evaluation of the callback function currently is in progress. This value is set by the system when the callback function is invoked. Manipulation by the PAL script has no meaning.

Table 6.5: Possible values for `Trigger.status`

To create a `Trigger` object, the callback function, the `Attribute` object to be tested, and the corresponding numeric threshold must be given. The following example shows how to create a trigger for an attribute `attr` and the callback function `recalculateStatus`. The trigger-event is raised as soon as the exactness value is below 80%:

```
var trigger = new Trigger(recalculateStatus, attr, 0.8);
```

Similar to the timer registration, triggers are registered with the PAL engine by calling the function `setTrigger` (cf. Appendix D). When called with a `Trigger` object as its only

argument, the function installs the trigger and starts it immediately after the script has been executed.

When the threshold value specified for a trigger has been passed, the callback-function registered with the `Trigger` object will be called, with the trigger being its only argument. To avoid concurrent script invocation, the PAL interpreter must not invoke a callback function if the corresponding script is currently being executed. Instead, the callback function should be invoked as soon as the current script run is finished.

A `Trigger` object can be removed from the system with the function `clearTrigger` (cf. Appendix D). The function's only argument is the trigger's identifier. The identifier can be specified literally or determined from a `Trigger` object using its function `id`.

6.3 Language Semantics

The major purpose of PAL is to specify the effect of incoming status update notifications on the principal's current presence status and to generate watcher-specific notifications to indicate changes of this status. Input and output information is represented as complex objects as described in the previous section. For calculating the current presence status, the PAL runtime environment provides a set of system variables and a rich core function library. Static variables and history lists enable inference of status values based on past events. Before describing the runtime environment and its specific configuration parameters to be used for changing the PAL engine's default behavior, we give a brief overview of the aggregation process in terms of script evaluation.

6.3.1 Script Processing

After a script has been installed following one of the options described in Section 5.4.4.3, it is run with an empty input set to initialize the script. After this initialization step, script execution is triggered only by asynchronous events like reception of status update notifications or system-generated events such as timer expiration (see Section 6.2.2.1 for more information on this).

Incoming status update notifications may use any presence document format supported by the PAL implementation, specifically the enhanced PIDF as defined in Section 5.3. The PAL engine tracks the current state of presentities it has PAL scripts associated with. Incoming notifications containing a partial status update are first transformed into a full status document with respect to the principal's current presence status as known by the aggregation engine. The corresponding status information objects are tagged with timestamps reflecting the local time of the notification being received.

Obviously, there is a bootstrapping period after upload of the first script where no status aggregation can be performed for partial notifications. The PAL engine then might either forward those messages to their final destination as it did before the script was installed or it might reject the partial notification with an explicit request for a full status description.

After calculating the full status the PAL script is executed with the abstract representation of the new status being passed as input parameter in a special input set, `IN`. The presentity's current status (as seen from former update notifications) can be accessed from the PAL script at the global environment variable `SYSTEM.CURRENT_STATUS`.

The input set may contain more than one presence status object if local policy allows for temporary suppression of incoming events that match a specific pattern. Suppressed events are stored in the input set and delivered with other events at a later time. Distinction between old (i.e. suppressed) events and recent notifications is done via the reception timestamps.

When executed, the PAL script transforms input status information received from external presence sources into status information to be sent to subscribed watchers. The output consists of abstract presence information objects as well (i.e. the input parameters could simply be copied to the output set). The output set's name is `OUT`.

Immediately after script execution, the PAL engine updates the local presence status information with the presence information documents (or their abstract internal representation) originally received from the remote presence source, ordered by the arrival timestamp. In addition, the history list is updated according to the rules defined in Section 6.3.2.1 making the generated notifications available for later script runs.

After the internal state has been updated, the PAL engine generates watcher-specific notification messages based on authorization information given in the script output. As the PAL engine cannot know the trust relationships between arbitrary subscribers and the principal, every presentity object comes with a formal definition of authorization classes that have a priori been identified by the particular principal (or generated by the principal's presence user agent). An authorization class in PAL is represented as a binding of a name with a set containing literal strings or identifiers of other authorization classes. The authorization classes defined by a presentity are stored in an associative set, with the keys being the classes' names.

A PAL engine may have pre-defined authorization classes to support presence information document formats that do not transport authorization class specifications. It is recommended to define at least the classes `private`, `protected` and `public` with their semantics specified in Table 6.6.

Class	Description
`private`	The presentity's presence URI.
`protected`	The presence URIs of the administrative domain the corresponding presentity is part of. Typically, this is a regular expression matching the presentity's domain name.
`public`	All subscribers that are not contained in the authorization classes `private` or `protected`.

Table 6.6: Pre-defined authorization classes in PAL

Additional presence classes can be defined by the PAL script by manipulating the output set. In this case, care has to be taken not to disclose any private data by changing the authorization classes defined in the input set. A PAL implementation should therefore check the script's output against the input parameters to detect disclosure of private information. In this case, the following actions of a PAL script should be logged as an error in the PAL engine's user log file:

- Changing the definition of authorization classes from script input.
- Modifying the authorization class of a channel object from script input without any other change.

Although these actions are logged for user reference, the watcher notification is performed as if no errors were detected. The error log is for user information only and can help debugging of errors in the PAL script.

With the given authorization class definition, the PAL engine can generate watcher-specific status update notifications containing only those channels the particular watcher is permitted to see. In this model, a channel object is included in the status update notification for a particular watcher if and only if the watcher's presence URI is part of any authorization class given in the channel definition.

A watcher-specific notification is sent only if it differs from the previous notification sent to that watcher. Local policy of the PAL engine may allow for notifications sent in a regular interval although no status changes are signaled.

Notifications according to this process are sent to all watchers with their subscription state being active at the time the corresponding notification is about to be created by the PAL engine.

6.3.2 Runtime Environment

In addition to presence status descriptions and system-specific objects such as event triggers, the PAL runtime environment stores a history list associated with each script and provides a number of configuration parameters for the aggregation engine. This section describes the history concept of PAL including a brief discussion of the lifetime of objects in PAL scripts. After that, we give an overview of the system's configuration parameters offered by the PAL runtime environment.

6.3.2.1 History

The global PAL environment provides two collections, `IN` and `OUT`. At the beginning of a script execution, the collection `IN` contains a set of presentities reflecting the status information of the principal whose presence status is to be calculated. During script execution, the set `OUT` is filled with new presentity objects. After all rules have been evaluated, the contents of the output set is transformed into a specific presence information document and sent to the principal's subscribers as explained before.

`IN` and `OUT` are array objects that give access to the PAL interpreter's script execution history. To determine a presentity's status, a PAL script may need to know status information that has been calculated in previous execution steps of that particular script. For example, if a principal had a workstation and a shared ISDN phone located in her office, change notifications of the phone could be treated more important if there was also recent user activity at the workstation.

Given an integer value `N` between `0` and an implementation-defined maximum `MAX_HISTORY`, `IN[N]` contains the presentities that have been fed into the presence aggregation module `N` steps ago. Similarly, `OUT[N]` is constituted from the presentities that have been created from their corresponding input sets when the PAL script has been evaluated. For $N < 0$ or $N \geq$ `MAX_HISTORY`, `IN[N]` and `OUT[N]` alway returns `null`.

In a PAL script, all objects have an infinite lifetime. However, if an object cannot be accessed by any PAL expression anymore during the current evaluation process or in any future PAL script to be evaluated with the same configuration parameters, this object may be destroyed at any time by the internal garbage collection mechanism. An object can be treated as inaccessible if no active binding exists at evaluation time and in future evaluations of the same script. In

future evaluations, bindings may exist in the execution history collections `IN` and `OUT`. Given these collections and a constant `MAX_HISTORY`, following invariant holds:

For all `N` $\geq$ `0`, `N` < `MAX_HISTORY`, objects that are in `IN[N]` or `OUT[N]` immediately after the PAL script has been executed, will remain existent for the next `N` invocations of this script. Figure 6.1 illustrates the aging of history information in subsequent script executions. The letters `n` to `s` in this example denote distinct status records that are stored in an input queue `IN`.

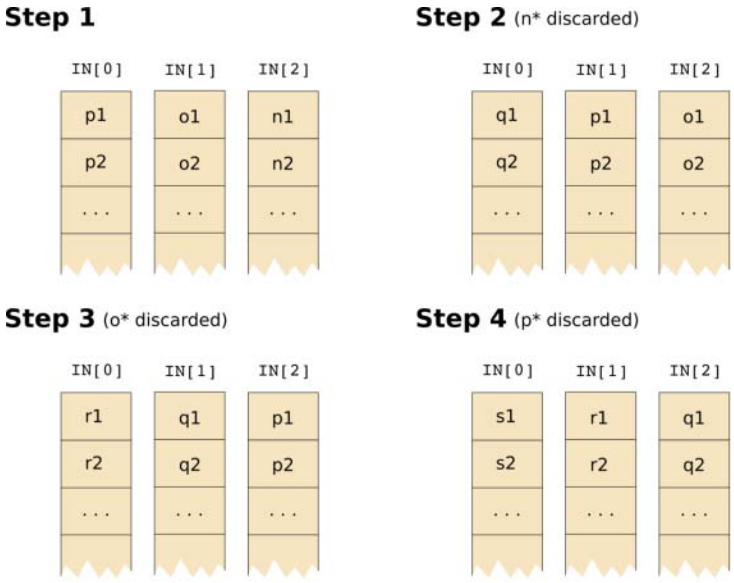

Figure 6.1: Aging of objects with `MAX_HISTORY == 3`

Note that the length of the history list can be changed with the configuration parameter `SYSTEM.MAX_HISTORY`. Decrementing this value causes objects at the end of the history list to be discarded. When incremented, the list is filled with `Null` values, i.e. these entries are not accessible until a sufficient number of aggregation steps has been executed.

6.3.2.2 System Configuration Parameters

The interpreter's configuration parameters can be accessed via the object `SYSTEM`. This object cannot be modified directly from the script. The PAL interpreter updates the object's contents immediately whenever a script has called a function that caused a change of the system configuration (e.g. when an extension module has been loaded by the script).

The object `SYSTEM` contains at least the bindings shown at the end of this chapter in Table 6.7. For each property of `SYSTEM`, the table defines the recognized object type and its default value. A short description in addition explains the semantics of each property.

6.4 Example Specifications

After having discussed the design of the presence aggregation language in the previous sections, this section provides several short examples to demonstrate the usage of PAL. The examples

given here are primarily for illustration of language features and may not be useful for real-world aggregation scenarios. A more realistic example taken from an existing aggregation service will be discussed in combination with the deployed test environment in Section 7.3.1.

Example 6.4.1: PAL script that changes a status variable

```
OUT[0]=IN[0];

var ch = getChannel(/light/i);


if (ch) {
  print("found light sensor");
  ch.light.value="off";
  OUT[0].setChannel(ch);
}
else
  print("no light sensor?");
```

Example 6.4.1 shows a typical sequence of PAL statements to change a value from a presence attribute. First, the current input set is copied to the output queue. After that, the function `getChannel` is used to retrieve a `Channel` object with the name "`light`" from the current status record. For simplicity, the argument is given as a regular expression for case-insensitive matching of channel identifiers.

If the specified channel was found, a descriptive message is printed to the log of the PAL interpreter and the value of the channel attribute `light` is set to the string `off`. The result is then copied to the first presentity contained in the output queue, replacing any existing channel with the same identifier.

Example 6.4.2: PAL script that creates a new presentity

```
/* get status descriptions for known sensors */
var light = getChannel("light-sensor"),
    idle  = getChannel("xscreensaver"),
    phone = getChannel(RegExp("phone"));

/* Set presence status depending on sensor states;
 * The state described here has one channel with id 'ch1'.
 */

var c = new Channel("ch1");

if (light.light == "on")        // light is on, use phone's status
  c.status = phone.state;
else                            // light is off, user might be away
  c.status =  (phone.state == 'busy') ? phone.state : "closed";

p = new Presentity("pres:alice@example.net");
p.channel[0] = c;
```

A more complex example is given in Example 6.4.2. Here, a new presentity is created with status attributes derived from several channels of the input set. The status of the new channel

`ch1` is set to the current status of the existing channel `phone` if the light indicates that the user is in his office. If it is dark, the user is treated as absent unless the phone's status is set to `busy` indicating an ongoing phone call.

This example also shows the use of application-specific presence vocabularies. Although the concrete vocabulary is out of scope for PAL, we encourage the use of the status vocabulary defined by the *Windows Messenger* application we have described in Section 4.2.1 as this is widely used not only by presence applications from Microsoft but also from other vendors. As many application also support the standardized PIDF [RFC3863], we suppose that the rich presence extensions defined in [RFC4480] will soon be supported by most presence clients as well.

6.5 Summary

In this chapter, we have defined an imperative language based on the standardized ECMAScript Language to control the process of presence aggregation, called *Presence Aggregation Language* (PAL). The presence information of incoming status notifications as well as outgoing presence documents is accessible from within a PAL script using an object-based notation. Presence-specific objects are first-class data types of the PAL language and thus can be used in expressions and passed as function arguments just like any other data type in that language.

Scripts, when executed, are expected to create a presence status description as output object stored in the container `OUT`. Received status descriptions are passed in a special input container, `IN`. These containers also provide access to input data and aggregation results from past script execution cycles, up to a pre-defined maximum stored in a global configuration object in the runtime environment.

Besides configuration parameters, the runtime environment stores so-called trigger objects used to invoke an associated callback function after a specific event has occurred. System-generated events are caused by timer expiration or by a time-dependent decay function yielding a result below a given threshold value.

To enforce local authorization policies a PAL engine honors authorization classes that are defined in presence documents or explicitly by the principal. As PAL scripts are allowed to modify the authorization information for incoming presence documents, care must be taken to avoid disclosure of sensitive data.

After we have described the architecture of our presence aggregation system in Chapter 5 and specified a language for presence aggregation scripts in this chapter, we will give a brief overview of our implementation of this system in the following chapter. The focus is laid on the message-oriented communication in the local presence environment and the implementation of the PAL processor. In addition, a SIP-based presence server has been created that handles subscriptions as well as publication of presence information to subscribed watchers.

Key	Type	Default	Description
VERSION	String	"1.0"	Specifies the language version. The string must not contain whitespace characters. It always starts with an integer number specifying the major version number, followed by a dot and an integer number that specifies the minor version number. Optional segments separated by a dot may follow. Each segment can consist of digits, alphabetical characters, "_" and "-".
MODULES	Array		This array contains a unique identifier for every extension module that is available. Only strings are allowed as module identifiers. It is recommended to use absolute URIs with at least one globally registered name component to guarantee uniqueness.
MAX_HISTORY	Number	0	Maximum number of entries in execution history. The value is a non-negative integer number and must always be specified by the PAL interpreter and must not change during script execution.
MAX_CPU	Number	30	Maximum CPU time that is assigned for a script evaluation cycle. The value is given in seconds and cannot be changed by the script.
CURRENT_ STATUS	Presentity		The current presence status as seen from former presence notifications, except those in the input set IN. After script execution, new presence status is calculated from the information that has been used as script input.
TRIGGERS	Array		List of active Trigger objects for this script.
WATCHERS	Array		List of current subscribers for this presentity. Watchers are represented as a String containing the watcher's unique URI. The contents of this field may change during script execution if the watcher status of subscribed clients changes or new subscribers occur.

Table 6.7: Properties of the object SYSTEM

Chapter 7

Implementation and Evaluation

After we have presented the conceptual architecture for our presence aggregation service and have motivated the fundamental design decisions in the previous chapters, this chapter discusses a concrete implementation we have developed to prove the applicability of our approach. In Section 7.1, we give an overview of the generic considerations made before the implementation work started, followed by a detailed description of the implemented modules in Section 7.2. Section 7.3 then describes the test phase of our system in several projects and gives ideas for further evaluation. The chapter ends with a brief summary of the presented topics.

7.1 Implementation Considerations

As presence aggregation is triggered from external events such as the receipt of a SIP message or a timer that has fired, the server implementation resembles a *reactive system*. Focusing on UNIX-based systems, we have decided to follow the single-threaded, asynchronous programming model which integrates well with the BSD-style socket operations. The major motivation we had for this decision is the clear program flow that results from a single thread, and the avoidance of race conditions that could occur if status updates for a specific presentity are received while the aggregation process for the previous event notification is still in progress.

In the past, many authors have discussed the advantages and drawbacks of either the event-driven approach or the multi-threaded model (see e.g. [BCB03, DZK+02]). As our intention was not to create a high-performance server, but an experimental research platform, we agree with Ousterhout's statement given in [Ous96] that the event-driven approach introduces less complexity on the application and is easier to debug.

As a consequence of our decision to use the single-threaded, event-driven approach, any input or output operation of our system has to be implemented in a non-blocking mode to avoid the entire application to wait for a system call to return. At the application level, this means that any relevant system event such as timer expiry or receipt of new data must be passed to the processing logic that coordinates the entire application. For example, when sending a large HTTP response (e.g. containing an HTML document that lists the current configuration of the aggregation engine), multiple `write` operations may be necessary before the transmission is complete. The application therefore must check the return value of the non-blocking system call used to send the data and decide whether or not the response is complete. If not, preparations must be made to continue the operation at a later point in time. The function then will return to the main dispatcher loop that controls the application's event multiplexing.

7.1.1 Layered Architecture

Following the functional requirements listed in Chapter 3, we have chosen to implement three communication interfaces for our presence aggregation server. First, an Mbus interface for decoupling the server from other components such as the parser for aggregation rules or the set of low-level sensors, a SIP interface for wide-area distribution of presence information, and an HTTP interface to facilitate upload of aggregation specifications and modification of the aggregation engine's global settings. Figure 7.1 shows the three external interfaces of the server component to be implemented.

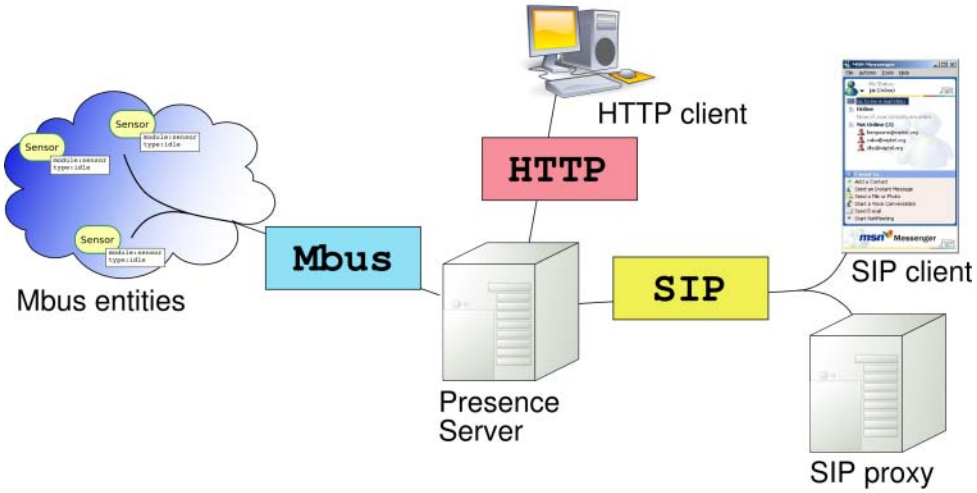

Figure 7.1: External interfaces of the presence aggregation server

In a single-threaded application, these communication interfaces share the same resources and thus must be coordinated from a single main loop. To facilitate later replacement of protocol implementations or adding new protocols to our system, we have decided to create a *server abstraction layer* as abstraction of underlying input/output operations. In addition to several utility functions for non-blocking file I/O, the layer provides application programming interfaces (APIs) that allow for easy creation of server applications for distinct transport protocols such as UDP and TCP. This design results in a layer architecture for our implementation as depicted in Figure 7.2.

The base of this architecture are the system interfaces that provide the external communication, usually the Mbus or an IP-based transport protocol stack. The server abstraction layer then encapsulates the actual transport protocol by providing an object-oriented interface for sending and receiving data. This layer is designed to interwork with the dispatcher loop provided by the C++ reference implementation for the Message Bus, which is used as the central event multiplexing mechanism.[35] Any external event as well as expired timers are signaled to the application logic by this dispatcher via an object-oriented interface that allows the association of handler functions with timers or socket descriptors. The application then is responsible for instantiating the appropriate message parser class depending on the interface that received the

[35]The Mbus reference implementation is available at `<http://www.mbus.org>`. See [Kut03] for a detailed description of its user-space event notification library.

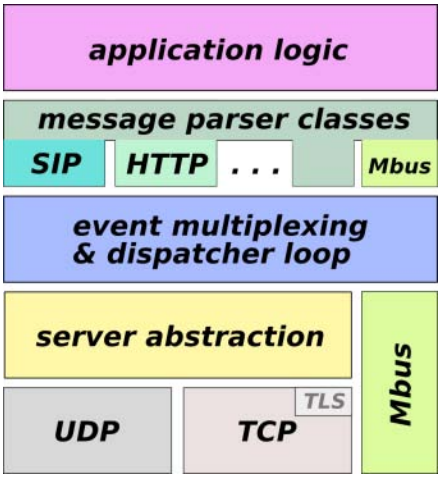

Figure 7.2: Abstract schema of implementation layers

incoming data, i.e., SIP, HTTP or Mbus. Once the incoming message has been parsed, the processing depends on the actual application.

The semantics of presence events and Mbus messages have been carefully defined to avoid inconsistencies of the server caused by overloading the processing entity with requests. In particular, presence events and Mbus RPCs have been designed to be robust against timeouts and packet loss. Events that have not been processed by the server and that are still present in the input queue, can be replaced by refreshed events. Outstanding Mbus RPCs will fail after some time, raising an error condition at the initiator's side.

Still an open issue is the lack of rate control for Mbus applications. As some of the sensor devices such as acceleration sensors may generate very frequent status update events sent to the Mbus, the aggregation server must be able to process these events as they arrive. Due to the missing rate control, no damping of the sender is possible. Therefore, events must be dropped at the sender to avoid a growing backlog of outdated event messages.

Currently, our implementation does not distinguish between unreliable events that were received on the Mbus interface or the SIP interface, and reliable Mbus RPCs or HTTP requests. The reason for this treatment of incoming requests is a server abstraction layer that was introduced to simplify the integration of SIP servers and HTTP servers with the Mbus infrastructure. In the next section, we describe the design rationale of this *server abstraction layer* and present some of the coding paradigms we have followed to keep the implementation modular and extensible.

7.1.2 Server Abstraction Layer

To develop a single-threaded server application with multiple external interfaces, care must be taken to keep the high complexity of this system under control. Modules of manageable size with clear API descriptions as well as an object-oriented programming style make development easier, but are not sufficient if external software must be integrated as well. In addition to these, we had to introduce a number of generalized template classes to facilitate module integration. In particular, the object-oriented system design was enhanced by C++ template classes for specific

purposes such as user-space event handling using callback functions. As these template classes can be instantiated with nearly any custom event type, they are treated as a basis component for building specialized applications such as SIP proxies or HTTP servers. In this section, we describe the abstract event implementation and show a concrete example of its usage in an application on top of the server abstraction layer.

A fundamental pattern of template-based programming in C++ is the use of *traits* that define a specific flavor of more generic template-classes. In Example 7.1.1, a trait class is given that describes an abstract event, containing a unique identifier `Id`, a type symbol `Type`, the actual event object `Event`, and a predicate `Pred` that enables customized event masking functions.

Example 7.1.1: Definition of abstract event traits

```
template <class _Type, class _Event, class _Id=void*, class _Pred=std::equal_to<_Type> >
struct EvDesc {
    typedef _Id   Id;
    typedef _Type Type;
    typedef _Event Event;
    typedef _Pred  Pred;

    Id     id;
    Type   type;
    Event *event;

    bool match(Type mask);

    EvDesc(Id i, Type t, Event *e);
};
```

Note that the given template class uses generic type parameters only. To create a set of event traits to be used with our server abstraction, the template must be instantiated as depicted in Example 7.1.2.

Example 7.1.2: Declaration of `IOEvent`

```
struct IOEvent {
    enum Type {
        NONE           = 0,
        CREATED        = 1,         /** new connection has been created */
        ACCEPT         = 2,         /** new incoming connection */
        CLOSED         = 4,         /** closed by remote peer */
        CLOSE          = 8,         /** closed by local party */
        READ_READY     = 16,        /** data available */
        WRITE_FINISHED = 32,        /** write has successfully been performed */
        ERROR          = 64,        /** error */
        TIMEOUT        = 128        /** connection timeout */
    };

    /* ... some member functions omitted for brevity ... */

    Connection *connection;
    int _errno;
};

typedef EvDesc<IOEvent::Type, IOEvent, void*, binary_and<IOEvent::Type> > IOEventTrait;
```

With this declaration, the new type `IOEventTrait` contains all information that is required to handle input/output events. The event type is implicitly defined as a bit vector with an entry for any event of the given enumeration. In addition, the structure contains two members, `connection` and `_errno`, that are used to describe the I/O event encountered. The `connection` field refers to a `Connection` object that is managed by one of the low-level servers that are used for network operations. In general, this object can be used to identify the endpoints of a communication relationship, i.e. the IP addresses and port numbers of communicating peers. The second field is used to store the error status of the last input/output operation from the global variable `errno` for deferred delivery.

Having set up the class `IOEvent`, a new event trait is declared from the template `EvDesc` as type `IOEventTrait`. As the event type is identified by a bit vector rather than a number, we have changed the default predicate function to the function `binary_and` taken from the C++ *Standard Template Library* (STL).

In the next step, this trait structure is used to instantiate a generic user-space event server where applications can register a callback function that is invoked whenever a specific event occurs. A unique identifier allows for distinct handler functions to be registered with a single event type. The final declaration of the trait-based event server class is shown in Example 7.1.3.

Example 7.1.3: User-space event server

```
template <class EventTrait>
class EvImpl {
public:
    typedef typename EventTrait::Id    Id;
    typedef typename EventTrait::Type  Type;
    typedef typename EventTrait::Event Event;
    typedef EventTrait                 EvTrait;

    enum Result { OK, DONE, ERROR };
    typedef TCallBack<Result,EventTrait *> CB;

    bool reg(Id id, CB cb, Type mask);
    bool unreg(Id id);

    bool getMask(Id id, Type &mask) const;
    bool setMask(Id id, Type mask);

    void dispatch(Type type, Event *event);

    template <class Func> void apply(Func &f);
};
```

This example shows the use of the type declarations from the template parameter to specialize the classes' member functions. The functions `reg` and `unreg` are used to create and destroy an association of a handler function with a specific set of events. The event set is specified by the parameter `mask` of the function `reg`. Together with the predicate function specified with the event trait class, the `mask` parameter determines whether or not the handler function must be called for an event of a specific type. The event mask may be changed at any time with the function `setMask`, e.g. to ensure that a handler for a file write operation is called only if there is outstanding data to be written.

Most of the work is done by the function `dispatch`. Invoked with an event type and an instantiation of the specific event class handled by this event server, it traverses the set of registered handlers and invokes any handler function associated with this event type. Following the generalization paradigm, the traversal itself is done by the template function `apply`. Example 7.1.4 demonstrates the conciseness that results from strict use of the C++ template mechanism, hence making the code more compact and maintainable.

Example 7.1.4: Generalized application of function objects to registered clients

```
template <class EventTrait>
template <class Func>
void EvImpl<EventTrait>::apply(Func &f)
{
    // "copy" initial client set
    ClientSet cs;
    std::transform(clients.begin(),clients.end(),
                   std::insert_iterator<ClientSet>(cs,cs.begin()),
                   EXTNS identity<typename ClientSet::value_type>());

    // partition new client set:
    // [cs.begin(),b)  -- clients that did not return OK on f()
    // [b,cs.end()]    -- clients that returned OK on f()
    typename ClientSet::iterator b =
        std::remove_if(cs.begin(), cs.end(),
                  EXTNS compose1(std::bind1st(std::equal_to<Result>(),OK),f));

    // Call unreg() for clients that did not return OK.
    std::for_each(cs.begin(),b,
                  std::bind1st(EXTNS mem_fun1(&EvImpl<EventTrait>::unregClient),this));
}
```

Here, only three statements are necessary to traverse the internal data structure that contains the mappings from event masks to handler functions. The first statement, `transform`, creates a local copy of the mapping structure. After that, a given function is applied to each element in this set. The function `remove_if` in addition tests the function result for a given predicate. The result set then is ordered such that the first part contains every member of the initial set for which the predicate did not return true. The final statement of `apply` then destroys the registration for those handler functions.

With these definitions, a specialized event server can be constructed. The following declaration creates the type `IOEventSrv` that is used throughout our server abstraction library `libgensrv` to signal input/output related events.

```
typedef EvImpl<IOEventTrait> IOEventSrv;
```

In general, no object that implements the application logic ever needs to inherit from `IOEventSrv` directly. Instead, we have provided a hierarchy of classes that add different *aspects* to the base class. Figure 7.3 depicts the relevant classes for creating server objects for interworking with the dispatcher loop from the Mbus library.

In this figure, we have included a number of classes from the Mbus distribution to show how our parts fit into the Mbus event multiplexer. All external components have a white background and are designated by dashed lines. As they are defined in the library `libnotifier`, care

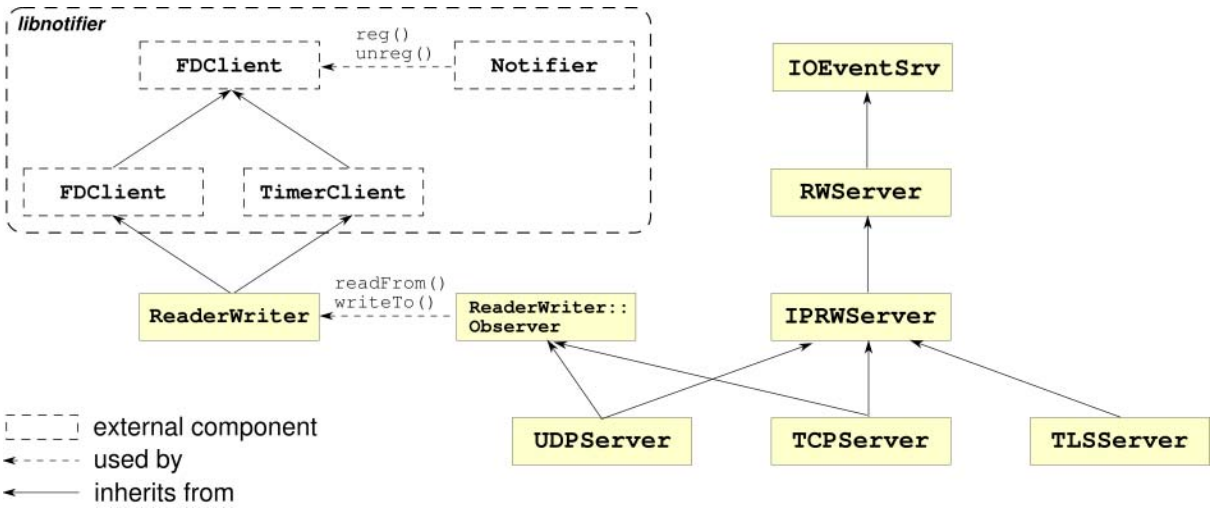

Figure 7.3: Partial class hierarchy from server abstraction layer

must be taken to configure the Mbus package to use this notification mechanism instead of its alternative dispatcher loop which cannot be extended this way.

The three classes `UDPServer`, `TCPServer` and `TLSServer` located at the bottom of this class hierarchy implement server objects for the transport protocols `UDP`, `TCP` and `TLS` over TCP, respectively. All three classes are derived from `IPRWServer`, a generic class for managing network connections (using objects of type `Connection` as seen before). `IPRWServer` provides pure virtual function interfaces for the operations `read` and `write` that have been inherited from `RWServer`.

The integration of these server classes with the dispatcher loop is realized via the class `ReaderWriter` that implements different strategies for buffered input and output (e.g. sending data packets with a capped binary exponential backoff as used in SIP). The class is derived from two `libnotifier` classes, `FDClient` and `TimerClient`, and thus can be registered with the unique `Notifier` object that provides the dispatcher loop.

To avoid too complex inheritance relationships, we have decided to use the *observer* design pattern for integration of the class `ReaderWriter` with our server objects. `ReaderWriter` therefore provides two functions, `readFrom` and `writeTo` that are used to trigger data input or output, respectively. Both functions take a reference to an object of type `ReaderWriter::Observer`, containing handler functions that are called after successful completion of the specific task, if an error occurred or the connection has been closed. These events in turn are fed into the event server by `UDPServer` and `TCPServer` to make them visible to the application logic via the interface of `IOEventSrv`.[36]

With this architecture, any protocol implementation at the application layer can use the interface definition of class `IPRWServer`, independent of the underlying network transport protocol. The event server in addition provides a generic interface used by applications to be notified from system events such as the receipt of data on a server port. At that point, the handling of events usually depends on the application layer protocol being used and thus

[36] `TLSServer` does not use the observer pattern because of the complexity introduced by the TLS handshake protocol. Therefore, the class is derived from `FDClient`.

individual handlers must be implemented for every server port. For protocols that use the text-based request/response syntax of HTTP, we have added a generic message parser that is capable of parsing the start line depending on the type of message (request or response), header fields, and the message body (with transparent support for MIME multipart bodies). An extension mechanism allows for customization of the header parser (e.g. to apply or avoid header field normalization) and for recognition of new URI schemes.

An example that makes extensive use of these mechanisms is the protocol engine we use for dissemination of presence information. Section 7.2.3 gives a detailed description of the transaction handling and application logic that are based on the generic server abstraction layer. The HTTP reference implementation in addition is used by WIPAM[37], a Web interface for management of PAL scripts at a presence aggregation server. WIPAM has been developed by Andreas Büsching as part of the research project PASST where initial ideas for the presence aggregation module have been evaluated.

7.2 Architectural Components

In this section, we discuss our implementation of the architectural components presented in Chapter 5. According to the requirements of the service architecture, the implementation covers three different areas:

1. a local infrastructure for the distribution of presence information;
2. the presence aggregation module (PAM); and
3. the presence distribution infrastructure.

A central design objective was the development of modular components that can be used in different environments. Hence, the presence aggregation module should be independent of the actual protocol used for wide-area distribution of the generated presence information, and the presence distribution infrastructure should be usable as standalone presence server without performing presence aggregation. Both components therefore have been designed as Mbus entities that are controlled via the relevant commands of Section 5.4.4. Figure 7.4 shows the resulting architecture.

The figure shows the *Aggregation Engine* which is de-coupled from the *SIP Protocol Gateway* and the *HTTP Protocol Gateway*, together with a number of additional Mbus entities such as sensors. The aggregation engine collects presence information sent by local sensors or received via SIP and forwarded on the Mbus by the SIP protocol gateway. After the aggregation process, the result is published via Mbus to the SIP protocol gateway. Depending on its configuration and the presence or absence of local presence sources, this Mbus-based system acts as a personal presence server in a local presence environment or as a public SIP server.

The implementation approach for a local presence environment is described in Section 7.2.1, the presence aggregation module is described in Section 7.2.2, and the presence distribution infrastructure is described in Section 7.2.3.

[37] Available at <ftp://ftp.tzi.org/tzi/dmn/wipam/>.

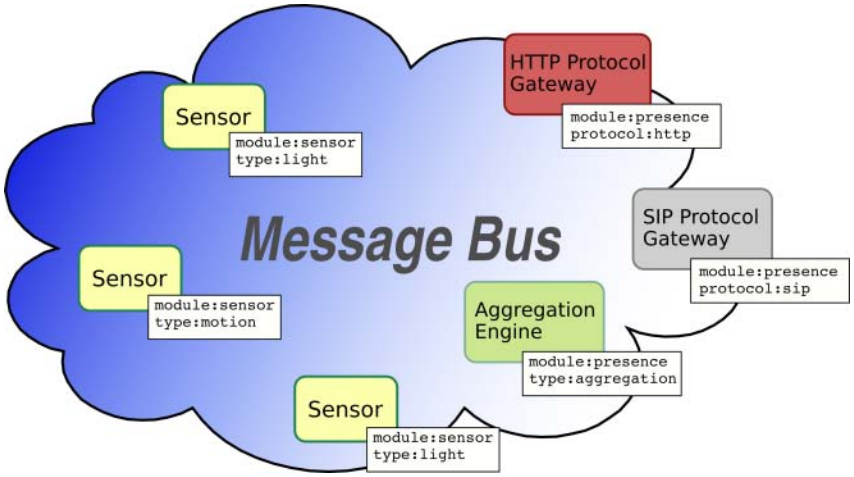

Figure 7.4: De-coupled implementation of a presence aggregation server

7.2.1 Local Presence Environment

A local presence environment is a set of devices that can generate presence information that is (at least partly) related to a specific user. The devices can be of different types, e.g., location awareness systems, personal (or even shared) telephones, or motion sensors. The presence information generated by these devices is evaluated and aggregated by a *presence aggregation module* (PAM). The PAM is a functional module of the personal presence server which is required for aggregation and dissemination of presence information on behalf of the user.

The local presence environment is an Mbus-based infrastructure as denoted on the left side of Figure 7.1. As a multicast-based messaging protocol, the Mbus facilitates the automatic location of new sensors and detection of removed sensors that provide an Mbus interface. As most technical devices do not even have IP connectivity, additional controllers may be used that interface with the sensor and forward the retrieved data onto the Mbus as visualized in Figure 7.5. For improved auto-configuration of Mbus entities, the *dynamic device association framework* (DDA) [KuOt03] might be used.

The Mbus communication channel is being used by the aggregation module to control the available presence data sources, and to retrieve descriptive information such as the sources' data schema. The vocabulary for these functions is specified in Appendix B. It provides an addressing scheme and Mbus commands for two types of entities: Mbus presence user agents (MPUAs, the presence sources), and the PAM. The tests of the local presence environment have been performed using several sensor modules that have been contributed by Andreas Büsching. The following sensors have been available for our tests:

- *Light Switch Sensor*

  A simulated sensor to simulate a light switch. The sensor is controlled with the keyboard. By pressing the **Enter** key the state of the light switch is toggled.

- *Chair Sensor*

  A simulated sensor simulating a weight-sensor placed on a chair's seat. The sensor's status is toggled after a random period of time.

- *Hardware Phone Simulator*

  A (simulated) hardware SIP phone that is capable of reporting its call state via Mbus to a control device such as the local presence server.

- *Bluetooth Device Detector*

  A sniffing tool that detects bluetooth devices. The sensor generates a presence notification containing the device name whenever a new bluetooth device was detected.

- *Idle Time*

  This sensor reports the idle time of a user, based on the information provided by the xscreensaver-command tool from the *XScreenSaver* project.[38]

The sensors listed here have been implemented in Python[39], an object-oriented programming language that is especially useful for rapid development of small applications. Using the Mbus messaging infrastructure, these lightweight scripts can be easily integrated with the complex aggregation service which is created in C++. The C++ implementation of the Mbus is built around the four libraries depicted in Figure 7.6. We have used these libraries for the implementation of the Mbus interfaces of the PAM and the presence distribution infrastructure. These two components are discussed in the following sections.

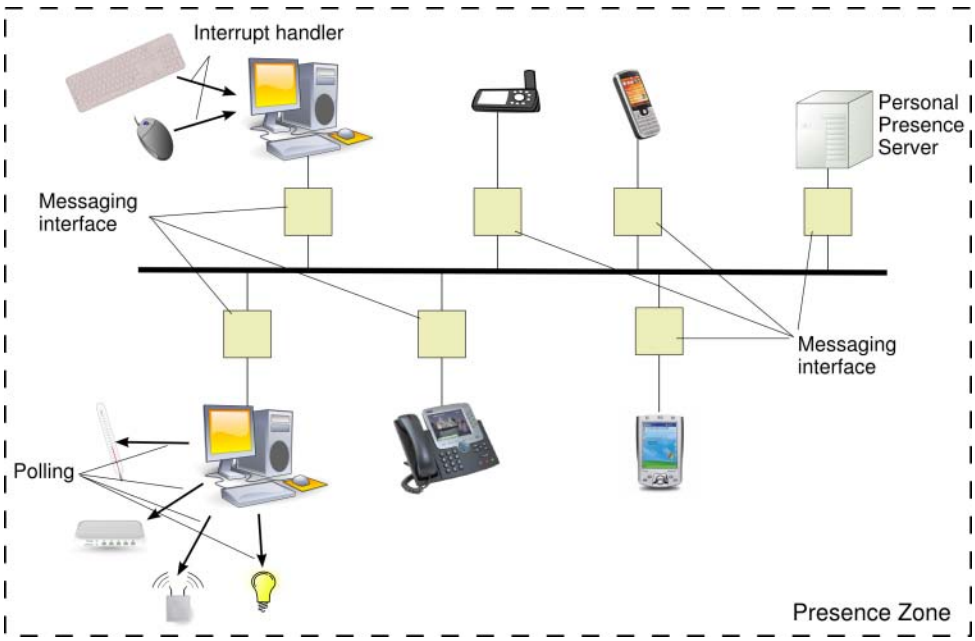

Figure 7.5: Mbus-communication of sensors within a local environment

[38]The project's homepage can be found at <http://www.jwz.org/xscreensaver/>.
[39]See <http://python.org>.

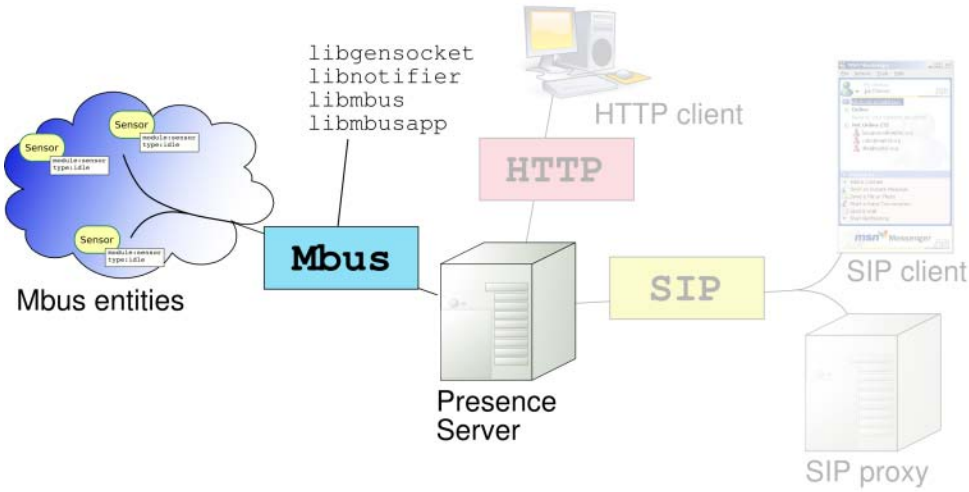

Figure 7.6: Generic libraries of the Mbus implementation

7.2.2 Presence Aggregation Module (PAM)

The core component of our aggregation service is the *Presence Aggregation Module* (PAM) which is responsible for aggregating presence data from multiple sources and publishing the aggregation result to authorized watchers. The aggregation process is controlled by user-provisioned aggregation specifications written in the *Presence Aggregation Language* (PAL) we have specified in Chapter 6. The integration of the PAM with existing presence servers is done via the standardized Message Bus as depicted in Figure 7.4, implementing the Mbus command set for controlling the PAM (cf. Section 5.4.4).

The following sections discuss various aspects that turned out to be essential for the implementation of the aggregation module. In Section 7.2.2.1, we describe the control flow for processing incoming presence documents as well as system-generated events that are relevant to the PAM. The internal representation of presence documents is explained in Section 7.2.2.2, followed by a discussion of the technologies we have used for evaluation of presence aggregation specifications (Section 7.2.2.3). A discussion of the serialization of presence data into client-specific document formats concludes this part in Section 7.2.2.4.

7.2.2.1 The Aggregation Process

The aggregation process is defined as a document transformation that is triggered by external or internal events. External events are signaled via the Mbus interface of the PAM, using the command `presence.publish` (see Section 5.4.4.1), while internal events usually occur after expiry of user-defined timers or triggers.

The respective event is used as an input parameter to the transformation process, together with the current status of a specific presentity, the user-provisioned aggregation specification and the associated runtime environment as well as the set of watchers that are subscribed to this presentity. This information is stored locally in an associative container of `Presentity` objects that is indexed by the URI of each presentity. If the presentity that is addressed in a publish command is not contained in this list, the event handler will automatically return an

error response for the RPC command. Otherwise, the presentity whose status record should be updated is found, and the provided scripts are evaluated.

During the script evaluation, the runtime environment can be updated using primitives from the PAL core function library defined in Chapter 6. To publish the transformation result, the aggregation specification must fill the global array variable `OUT` with one or more `Presentity` objects. These objects contain the definitions of available media channels together with their current presence status.

The presence attributes for a channel may be copied from the input set or created anew. Channel descriptions can also be annotated with authorization information to facilitate the generation of watcher-specific views. These are constructed from the output set after all aggregation scripts for the current presentity have been evaluated. To do so, the set of subscribed watchers is split into distinct partitions, each representing one authorization class. Now, a custom view on the result document is created for each authorization class, and then sent to the members of this class using the command `presence.notify`, as described in Section 5.4.4.1.

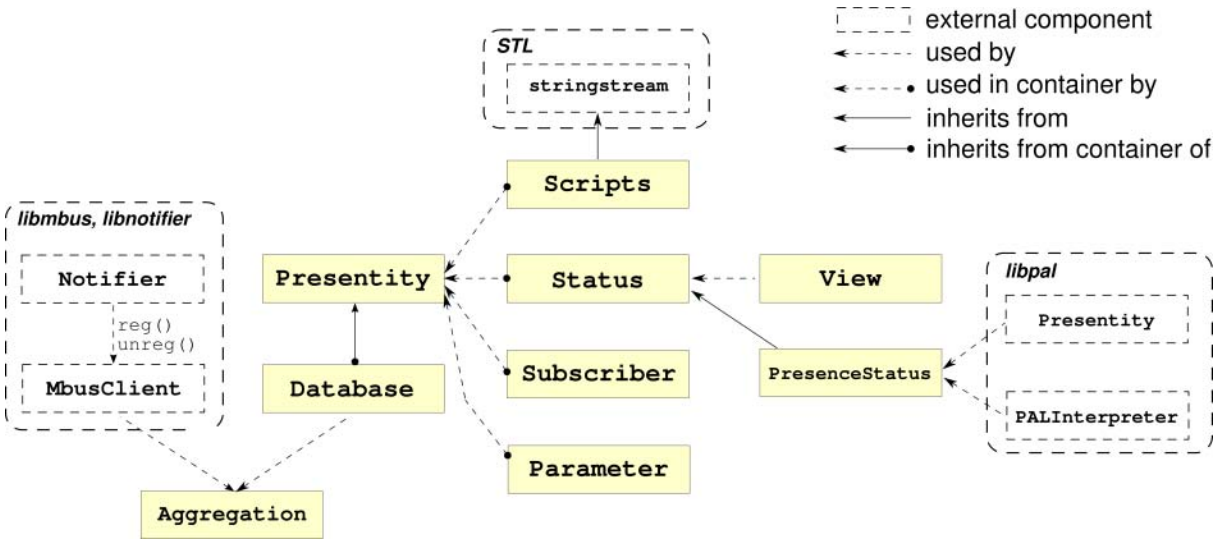

Figure 7.7: Class hierarchy of PAM components

Our implementation of this aggregation process is separated into two parts, separating the Mbus communication from the PAL language interpreter. Figure 7.7 depicts the core classes that are used to implement the Mbus communication. The central component here is the virtual class `Status` that provides a stub for integration with the actual aggregation engine. `Status` objects are used from `Presentity` objects that store status descriptions for every known event type, the user-provisioned aggregation scripts, the list of subscribed watchers, as well as global parameters to control the aggregation process.

The presentities are organized in a storage container `Database` that is used to store `Presentity` objects in local memory during runtime. The `Database` structure then is used by a single object of type `Aggregation` which serves as an interface to the Mbus implementation.

As said before, the class `Status` serves as an interface between Mbus communication and PAL interpreter. For the discussion of script evaluation in Section 7.2.2.3, it is important to note how this interface works: The abstract interface is defined by the class `Status` that provides three pure virtual functions that must be overwritten by an aggregation implementation

(i.e. `PresenceStatus` in Figure 7.7). Example 7.2.1 shows an abbreviated class definition containing the declaration of the three function interfaces.

Example 7.2.1: Declaration of stub functions for class `Status`

```
class Status {
  public:
    virtual View createView(const std::string &watcher, const std::string &type = "") = 0;
  protected:
    virtual bool _updateStatus(const std::string &data, const std::string &type) = 0;
    virtual std::string _getStatus(const std::string &mimeType) = 0;

  /* ... other declarations omitted for brevity ... */
};
```

The most important function of class `Status` is the function `_updateStatus`. The function is used to update the status information with the data given by the argument `data`. The parameter `data` is supposed to be formatted according to the rules of the MIME media type `type`. To retrieve the current status as a document of type `mimeType`, the function `_getStatus` is used. Both functions are protected, i.e. they are accessible only from derived classes. The public interface of the class `Status` provides two wrapper functions, `update` and `asString`, that use the protected functions if necessary.

The third function given in the class declaration of Example 7.2.1 is used to create watcher-specific views of the current presence status. A `View` consists of a string representation of the document according to the rules of the specified MIME media type, and the given watcher name that represents the corresponding authorization class.

To link against the PAL interpreter library, we have created a class `PresenceStatus` that implements the pure virtual functions inherited from `Status`. The aggregation engine is called from the function `_updateStatus` as denoted by Example 7.2.2.

The function definition in this example shows the implementation of the aggregation server's internal processing logic. First, the existing status descriptions are moved to a history queue for later retrieval. After that, the document that was passed to `_updateStatus` is parsed into the internal representation as discussed in the next section. The resulting object tree is stored as input to the evaluated script.

The next two steps invoke certain functions of the PAL interpreter stored as member variable `_pal`. These are necessary to convert the internal data structure to JavaScript objects for use with the aggregation scripts. If aggregation rules are given, the engine uses the function `execMultipleScripts` to evaluate the installed aggregation specifications with the current input set. On success, the output set will be populated with a presence status description according to [RFC3261]. If no aggregation rules have been specified, the aggregation engine literally copies the input set to the output set, hence performing an identity transformation.

Before discussing the implementation of the presence aggregation language in Section 7.2.2.3, we describe the creation of the internal representation of the input documents. This process includes the correct parsing of XML documents as well as the creation of persistent JavaScript objects to allow access from scripts.

Example 7.2.2: Implementation of PresenceStatus::_updateStatus

```
bool PresenceStatus::_updateStatus(const std::string &data, const std::string &type)
{
  moveQueue( _current[IN], _history[IN], historySize[IN] );
  moveQueue( _current[OUT], _history[OUT], historySize[OUT] );

  parsePresenceDocument(_pres.entity, data, INSERTER(_current[IN]) ); // fill input queue
  _pres.update(_current[IN].begin(), _current[IN].end(), false);      // update status

  _pal.setCurrentStatus(&_pres);
  _pal.setInput(_current[IN].begin(), _current[IN].end());

  // get all PAL rule-sets for this presentity
  typedef std::vector<std::string> Rules;
  Rules rules;
  getRules(_pres.entity, std::back_insert_iterator<Rules>(rules));

  // apply scripts: result is put to _history
  if (! rules.empty() ) {                                             // evaluate given scripts
    std::string result;
    PALInterpreter::Result code = _pal.execMultipleScripts(rules.begin(),rules.end(),result);

    /* ... error handling depending on result code ... */
  } else {                                                            // no scripts given
    cut(_current[IN], 0);                                             // default is to copy
    _current[OUT].push_back(new libpal::Presentity(_pres));
  }

  _type = type.empty() ? "application/pidf+xml" : type;
  return true;
}
```

7.2.2.2 Building the Internal Representation of Presence Documents

Whenever a presence source sends a status update notification, the data contained therein must be parsed, and an internal representation of the specific status description must be created. As seen in the previous section, the received document is passed to the function `_updateStatus` of an object that is derived from class `Status`. In our implementation, we have focused on the aggregation of enhanced PIDF documents, and therefore support only the MIME media type `application/pidf+xml`, with optional extension elements from the proposed XML namespace `urn:ietf:params:xml:ns:pidfxy` as defined in Section 5.3.

To parse the PIDF document syntax, we have used *expat*[40], a very lightweight open-source XML parser. Since expat is a stream-based parser that does not provide any support for XML namespaces, we have created a simple C++ wrapper around expat that creates a hierarchical tree with nodes that represent the element structure of the parsed document. If an element contains a namespace prefix declaration, the corresponding tree node is annotated with this information. The tree structure we are using is similar to the standardized XML *Document Object Model* (DOM), but restricted to the interfaces that are necessary for parsing presence documents into an object representation. For a more efficient implementation, only elements from PIDF and known extension elements are parsed. Other element types are discarded as they appear in the document.

[40] Available at <http://expat.sourceforge.net>.

After all relevant elements have been parsed, the internal object model is created from the element tree while it is traversed. We have defined a C++ representation of the conceptual objects specified in Section 6.2, to store a presentity's status description independent of its representation as XML document or ECMAScript object. This module hence could be used as a standalone presence server, without dependencies to external document formats.

C++ Class Interface

The library `libpal` provides three class definitions to carry the content information of a PIDF document. The first, `Presentity`, acts as a container for a set of `Channel` objects, together with a set of handler functions to fill the data structure from a given PIDF document and create a customized output document. The relevant members of this class are shown in Example 7.2.3.

Example 7.2.3: Declaration of class `Presentity`

```
class Presentity {
public:
  std::string entity;           // unique identifier of presence entity

  Channels channels;            // mapping from channel identifiers to objects
  Notes    notes;               // mapping from language tokens to text strings

  static Presentity *create(const Element *node, const std::string &pres = "");

  bool createView(const std::string &uri, Element &result) const;

  bool update(const Element *node, bool fullState = true);
  bool update(const Presentity *pres, bool fullState = true);

  /* ... several other functions not shown here for brevity ... */

protected:
  virtual void handleElement(const Element *node) throw (PALError);
};
```

As indicated in this example, each `Presentity` object is identified by the presentity's unique URI. The member variables `channels` and `notes` are used to store the channel information and the natural-language comments taken from the corresponding PIDF element types `tuple` and `note`. The data types `Channels` and `Notes` use the container type template `map` from the C++ Standard Template Library to map identifiers to object references. For a `Channel` object, the identifier is taken from the attribute `id` of the corresponding XML element `tuple` that should be constant over a series of presence status descriptions published by a single presence source. Natural-language comments usually do not have a unique identifier but may come in various languages. Therefore, the container `Notes` provides a mapping from language tags contained in the XML attribute `xml:lang` of the element type `note` to the contents of the particular element.

A `Presentity` object is typically created from an XML document tree by the static function `create`. This function takes an element node for a `presence` element in the PIDF namespace as its first argument, and an optional identifier that can be used to override the presentity URI of the PIDF document. After a new `Presentity` object was created, the function

`update` is called with the given element node as its object and the parameter `fullState` set to `true`, causing an update of the presence status description according to the given XML tree. If `fullState` was set to false, the XML tree would be treated as a partial update, i.e. channel descriptions not contained in the input document were not changed.

Handling PIDF Extension Elements

The transformation of the XML input format to an internal representation that is based on C++ object structures requires specific handling of extension elements that were not known a priori. The mapping from the XML tree structure to the internal representation thus uses a dispatcher that is called for every element node while traversing the XML input tree. The implementation therefore uses the template class `Dispatcher` contained in the library `libextcpp` that has also been used for event dispatching in the server abstraction layer as described in Section 7.1.2.

To use this mechanism, new handler functions for yet unsupported element types must be registered with the element dispatcher contained in each `Presentity` object. The new handler will be called by `handleElement` whenever the `update` function comes across an unknown event type. A derived class `MyPresentity` thus can register a new element handler as shown in Example 7.2.4.

Example 7.2.4: Registration of a custom element dispatcher

```
MyPresentity::MyPresentity(const std::string &ent) : libpal::Presentity(ent)
{
  elementDispatcher.insert("http://example.org/my-presence-namespace" NS_SEP "extstatus",
                           callback(*this, &MyPresentity::myStatusHandler));
}

bool MyPresentity::myStatusHandler(const Element *node)
{
  getCharContent(node, std::back_insert_iterator<std::string>(extstatus));
  return !extstatus.empty();
}
```

In the constructor of `MyPresentity`, a dispatcher for the element type `extstatus` in the namespace `http://example.org/my-presence-namespace` is registered with the dispatcher object `elementDispatcher` that is inherited from the parent class `Presentity`.

During runtime, the handler function for `extstatus` is invoked whenever an element of the given type is found in the input tree, with this particular node as function argument. In our example, the handler for this element type simply copies the textual contents into a string variable and returns `true` if this operation was successful.

This callback-mechanism provides a flexible interface to add new handler functions for extension elements. If extensions use nested element structures, the dispatcher must be called recursively to process these elements as well. As PIDF documents usually have a flat element structure, this recursive processing would rarely be used.

An advantage of the event-based processing of input trees is its independence of the actual input syntax. Since PIDF and most of its extensions use key/value-pairs for information encoding, no complex checking of the input document's element structure is necessary to extract the

content information. However, if a validation of the document structure were desired, the XML parser expat could be replaced easily by a validating XML parser.

Support of Different Input Document Formats

As shown in the previous sections, our implementation of the data model defined in Section 6.2 is restricted to the standardized presence information data format defined in [RFC3863] and its possible extensions. Since various presence clients do not support this format or support only outdated variants thereof, we have implemented an optional document transformation to map proprietary XML-based input formats to PIDF. The transformation engine is based on *libxslt*[41], an open-source implementation of [XSLT].

On startup, the format converter reads a set of XSLT transformation specifications that are annotated with the MIME media type identifiers of the supported input format and of the generated output format. These transformation specifications are stored in an internal data structure that allows for efficient retrieval of a specification for a transformation between specific document formats. This way, not only transformations into PIDF as input format for the presence aggregation process are supported, but also the creation of various output formats for publication of the aggregation result. For example, if clients have indicated support for the widely used XPIDF format during subscriptions, the aggregation module will produce this format instead of PIDF when sending presence update notifications to this particular clients. A detailed discussion of this aspect is contained in Section 7.2.2.4.

It is important to notice that the data structure used to store the transformation specifications is not bound to PIDF as source or target format in any way. Instead, it allows for transformations between arbitrary document formats. Hence, if the presence aggregation module in future versions should use another input document format, this could be achieved by changing the MIME media types being used as format specifiers. In addition, chained transformations could be used to support transformation specifications for specific extension vocabularies combined in a single presence document.

7.2.2.3 Evaluation of Aggregation Specifications

After the internal representation of the presence status is created, the aggregation specifications associated with the corresponding presentity must be processed. For interpretation of the presence aggregation language, our implementation uses *SpiderMonkey*[42], an interpreter for JavaScript 1.5 which is compliant to [ECMA-262]. SpiderMonkey is part of the Mozilla software suite and is used e.g. in the popular Web browser *Firefox*[43].

SpiderMonkey provides a single library `libjs` that is linked against our aggregation module. To bootstrap the application, a number of API calls are necessary that we have encapsulated in the C++ class `PALInterpreter`. In particular, every presentity is provided with its own JavaScript runtime environment and application context to avoid conflicts between different presentities. Figure 7.8 depicts this relationship between `libpal` and classes defined in `libjs` and the PAM main module `pam`.

[41] Available at <http://xmlsoft.org/XSLT/>.
[42] Available at <http://mozilla.org/js/>.
[43] See <http://mozilla.org/firefox/>.

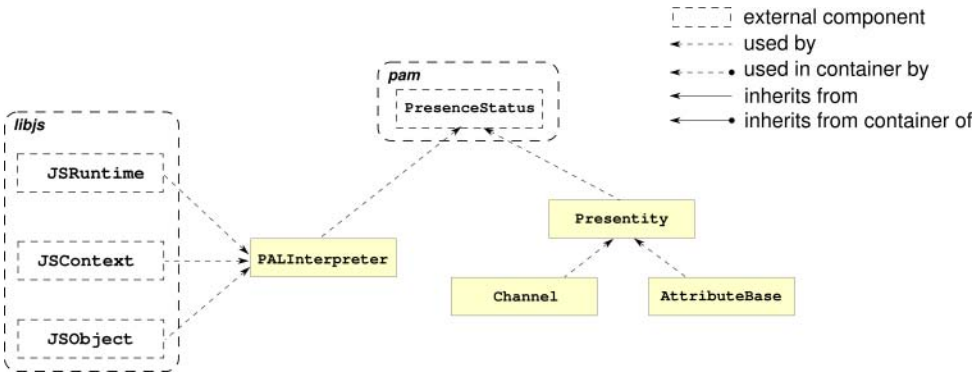

Figure 7.8: Class hierarchy of `PALInterpreter`

To prepare for the evaluation of JavaScript programs, the language processor must be initialized with a new `JSRuntime` object that holds the memory pool for dynamic object allocation and destructions as well as the list of defined objects and functions and several state variables that reflect the global state of the runtime environment. The specific control flow of a single program thread is managed by a context object of type `JSContext` which can be shared between subsequent scripts that are evaluated for a single presentity. The context holds the execution state of a script, including pending exceptions, errors, user-specific callback functions, and the function call stack for the particular script.

These two objects define the static frame for script execution with SpiderMonkey. In addition, the runtime environment must be initialized with the definitions of the PAL core function library (cf. Appendix D) and the native objects representing the current presence status (cf. Section 6.2). As the presence status can change between subsequent script runs, the status must be set before each run, while the remaining initialization steps are required only once.

To manage C++ definitions of native JavaScript objects, we have used a template class `PALObject` that is specialized with the *traits* of a specific object type. The template specialization then is resolved by the C++ compiler to a type-safe definition of the specific object type as required by the JavaScript engine. Example 7.2.5 shows the definition of the template class `PALObject`, together with the object constructor that generates a call to the function `JS_InitClass` for registration of the new object type with the JavaScript runtime environment.

The creation of prototypes for the native objects is done after initialization of the runtime environment by constructing local objects of the specialized type as shown below:

```
void PALInterpreter::initRuntime() {
  PALObject<PALPresentityTraits>(_ctx, _global);
  PALObject<PALChannelTraits>(_ctx, _global);
  PALObject<PALAttributeTraits>(_ctx, _global);
  PALObject<PALTriggerTraits>(_ctx, _global);

  /* ... other initialization steps, e.g.\ calls to JS_DefineProperty ... */
}
```

Here, `_ctx` and `_global` refer to the evaluation context for this script and the list of global objects, respectively. An example for an existing structure of traits that is used with the template class `PALObject` is given in Example 7.2.6.

Example 7.2.5: Declaration of the template class PALObject

```
template <class ClassTraits>
class PALObject {
public:
  PALObject(JSContext *ctx, JSObject *global);
  operator JSObject* () { return _obj; }           // returns the object prototype
  operator bool() const { return _obj != NULL; }   // object prototype exists
private:
  JSObject *_obj;                                  // pointer to the object prototype
};

template <class ClassTraits>
PALObject<ClassTraits>::PALObject(JSContext *ctx, JSObject *global) : _obj(NULL)
{

  _obj = JS_InitClass(ctx, global, NULL, &ClassTraits::classDefinition,
                      ClassTraits::constructor, ClassTraits::MIN_ARGC,
                      ClassTraits::propertyDefinition, ClassTraits::functionDefinition,
                      NULL, NULL);
}
```

The static definitions required by JS_InitClass follow the naming convention of PALObject, i.e. the structure provides a class definition for the new object type Presentity in the static member classDefinition, as well as the definition of associated properties and function objects in propertyDefinition and functionDefinition. The constructor function for creation of new objects of this type must be called constructor, and the number of arguments is given by the enumeration symbol MIN_ARGC.

The declaration of PALPresentityTraits together with its implementation of native functions such as the object constructor or the additional member functions setChannel, getChannel and removeChannel, illustrates how native objects can be created by instantiation of specialized templates. Extensions to this interface can be added as they are needed as the type conversion functions makeObject show.

The evaluation of PAL scripts is governed by the PALInterpreter object that is associated with every presentity. A subset of the PAL interpreter's API is shown in Example 7.2.7. Before a PAL script is evaluated, the runtime environment is updated with the current presence status of the associated presentity by calling the function setCurrentStatus. If new presence documents are available, the input queue IN is filled using the function setInput with a list of parsed presence documents (i.e. Presentity objects).

After initialization of the runtime environment, the function execute is called to evaluate an existing script. If a script is evaluated for the first time, it must be passed to the function executeScript instead, together with a string object that takes the literal evaluation result.

In our current implementation, the functions execute and executeScript block until the given script terminates. As the termination of a PAL script cannot be detected prior to its evaluation because of the language's Turing-completeness, the interpreter object can abort script execution after a configurable time. Initial tests with our environment yielded an average value of 20 ms required for execution of small PAL scripts. We envision that optimized servers should use this value as an upper limit before a script is abandoned.

The return value of the execution function indicates if the script evaluation was successful. In this case, the value PALOK is returned. Possible error values indicate a syntax error in the

Example 7.2.6: Declaration of `PALPresentityTraits`

```
struct PALPresentityTraits {
  static JSClass classDefinition;

  enum { FLAGS = JSPROP_ENUMERATE | JSPROP_READONLY };

  static JSPropertySpec propertyDefinition[];
  static JSFunctionSpec functionDefinition[];

  enum PropertyId { ID, CHANNELS };

  static JSBool constructor(JSContext *, JSObject *, uintN argc, jsval *, jsval *);

  enum { MIN_ARGC=1 };                // minimum number of constructor arguments

  static JSBool getChannel(JSContext *, JSObject *, uintN, jsval *, jsval *);
  static JSBool setChannel(JSContext *, JSObject *, uintN, jsval *, jsval *);
  static JSBool removeChannel(JSContext *, JSObject *, uintN, jsval *, jsval *);

  /* Creates JSObject from given Presentity object. */
  static JSObject *makeObject(JSContext *, const Presentity *);

  /* Creates Presentity object from given JSObject. */
  static Presentity *makeObject(JSContext *, const JSObject *);
};
```

provided PAL script (`PALSYNTAX`), a runtime error during script execution (`PALRUNTIME`), or a timeout as described before (`PALTIMEOUT`).

If the script execution succeeded, the output queue contains the presentity objects passed from the PAL script as shown in the examples of Chapter 6. To extract these objects, the function `getOutput` is called with an insert iterator as its argument. If one or more `Presentity` objects are returned by these functions, the subscribed watchers must be notified as described in the following sections.

7.2.2.4 Generating XML Output

After the evaluation of a PAL script has produced one or more objects representing the aggregation result, the aggregation module must process the set of watchers for this presentity and create one or more presence information documents that contain watcher-specific views of the result document. The generation of output documents is done by the member function `createView` of the particular presentity (see Example 7.2.3).

The actual number of different documents to be created depends on the authorization classes defined by the output document or by manual configuration. For every watcher, the output processor checks the authorization classes that contain the watcher's URI. Any channel description with an `authclass` attribute denoting an authorization class the watcher is not member, will be removed from the generated output document. For example, consider `bob@otherdomain.com` and `mallory@strangers.com` having subscribed Alice's presence status. The authorization policy for this presentity is configured as shown in Example 7.2.8.

Here, the potential watcher Bob is member of the class *friends* while Mallory is not contained in any authorization class definition. Now, the aggregation specification can use the authorization class `friends` to mark channel descriptions that contain sensitive data and there-

Example 7.2.7: Declaration of `PALInterpreter`

```
class PALInterpreter {
public:
  template <class Input> void setInput(Input start, Input end);
  template <class Output> void getOutput(Output out);

  void setCurrentStatus(const Presentity *);

  enum Result { PALOK, PALSYNTAX, PALRUNTIME, PALTIMEOUT };

  Result execute();
  Result executeScript(const std::string &script, std::string &result);

  /* ... */
};
```

Example 7.2.8: Authorization classes for Alice

```
<authdef xmlns="urn:ietf:params:xml:ns:pidfxy">
  <authclass name="friends">
    <authelem>bob@otherdomain\.com</authelem>
    <authelem>[^@]*@mysportsclub\.com</authelem>
    <authelem>jill@example\.net</authelem>
  </authclass>
</authdef>
```

fore should not be published to other watchers than Bob, Jill, and the members of the sports club. If no authorization class is given for a specific channel description, it is published to any subscribed watcher according to the semantics of PIDF.

The example document in Example 7.2.9 contains a basic presence description created from Alice's aggregation script. For simplicity, we have included only the elements that are necessary to understand this example. Documents created by the aggregation engine typically contain numerous extension elements in the PIDF-XY namespace as shown e.g. in Appendix C.

The document presented here contains two channel descriptions represented as element `tuple` for compatibility with PIDF. The first channel with the name `office` denotes the presence status of Alice's office phone. Currently, she is in a meeting and therefore the status is set to `CLOSED`. The second channel describes the instant messaging client installed on her notebook. The presence attribute `authclass` with the value `friends` restricts the information on this channel's presence status to the group of people contained in the authorization class with this name. Thus, Bob will be notified from the aggregation module with the presence document shown in Example 7.2.9. Mallory, in contrast, receives a presence document containing only the description of the first channel, i.e. the presence tuple with the unique identifier `op123m`. The second presence tuple is removed from the document during creation of the watcher-specific view.

It is important to notice that the function `createView` is called for every subscribed watcher to retrieve the presence document that must be sent to this particular watcher. As the number of watcher classes usually is very small, the presence documents are cached by the `Presentity` object that has created them. If another watcher to be notified is a member of the same authorization class, the function `createView` simply returns the document from the cache. Once a

presence document with at least one channel description is found for a specific watcher, it is passed to the SIP protocol engine that creates the SIP notification as described in the following section.

Example 7.2.9: Presence document with different authorization classes

```
<presence entity="pres:alice@example.net"
               xmlns="urn:ietf:params:xml:ns:pidf"
               xmlns:pidfxy="urn:ietf:params:xml:ns:pidfxy">

  <tuple id="op123m">
    <status>
      <basic>CLOSED</basic>
      <pidfxy:name type="string">office</pidfxy:name>
    </status>
    <contact priority="0.8">sip:alice@134.102.218.61:5060</contact>
  </tuple>

  <tuple id="j8sdv237">
    <status>
      <basic>OPEN</basic>
      <pidfxy:name type="string">jabber</pidfxy:name>
      <pidfxy:authclass type="string">friends</pidfxy:authclass>
    </status>
    <contact priority="0.8">
       sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE
    </contact>
  </tuple>

</presence>
```

7.2.3 Presence Distribution Infrastructure

After a set of watcher-specific status update notifications has been created, it is passed on the Mbus to the SIP protocol engine for dissemination. The SIP interface of the protocol engine is built around the libraries indicated in Figure 7.9, and uses the generic server abstraction layer we have described at the beginning of this section. Our implementation offers SIP message transport according to [RFC3261] over UDP and TCP with optional use of TLS. The implementation is split into four parts:

1. A generic server for message transport over UDP or TCP,
2. a SIP message parser,
3. the SIP transaction layer, and
4. the application logic.

As the creation of network servers and parsing of HTTP-style messages already has been addressed in Section 7.1, this section focuses on the management of SIP transactions and the application logic we have developed for multi-step aggregation of presence information.

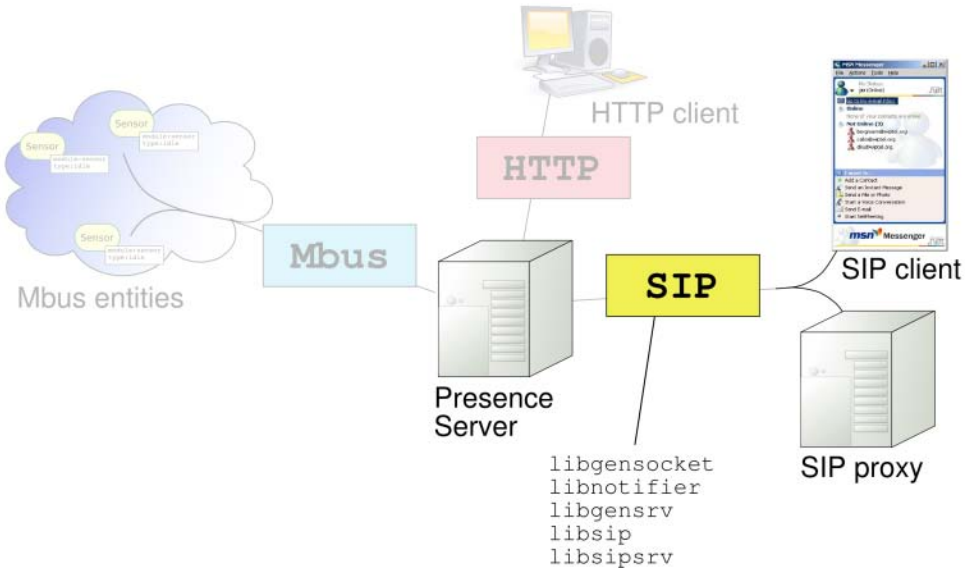

Figure 7.9: Generic libraries of the SIP implementation

7.2.3.1 Handling of SIP Transactions

SIP-based applications such as call signaling or event notification are defined in RFC 3261 as a series of atomic operations initiated by a user agent client sending a request. An operation is always completed with a final response returned either by the addressed user agent server or a proxy in the network (e.g. if the particular resource could not be found). If no response is received for a request that was sent over an unreliable transport protocol, the request is retransmitted with an exponential backoff to minimize the risk of a network congestion through retransmissions. A fixed maximum lifetime in addition assures that allocated memory for storing transaction state in proxies can be released after some time.

As the transport of SIP messages over UDP has become a de-facto standard for existing SIP clients, these timeouts must be considered when designing a SIP-based application. In particular, responses must be generated fast enough to avoid retransmissions when sufficient bandwidth is available with a reasonable delay, and a possible overlapping of transactions must be considered. Under perfect conditions, a response should be received not later than 500 ms after the request was sent.

In a congested network, delayed SIP messages may cause inconsistencies in presence information as messages are lost or come too late. Even worse, presence aggregation engines by definition operate on partial updates when combining presence documents from multiple sources. The server therefore has to store additional meta-data about the event notifications received by remote presence sources.

7.2.3.2 Application Logic

The application logic of our presence server implementation combines the SIP protocol handling according to the Mbus API defined in Section 5.4.4. The Mbus part is derived from the components we have described in Section 7.2.2 and is used to trigger the SIP-specific func-

tionality of the server. Depending on its actual configuration, our server can act as a personal presence server that disseminates presence documents published by the PAM to a configured next-hop server as depicted in Figure 7.10.

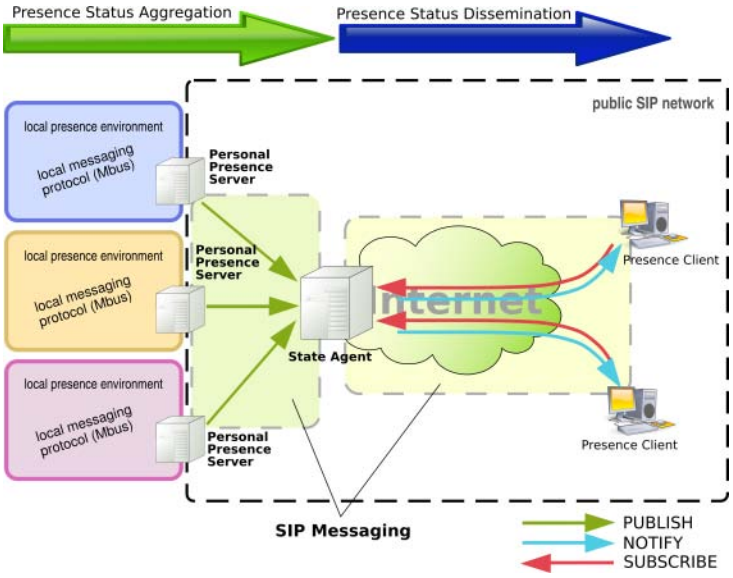

Figure 7.10: Abstract architecture of a presence aggregation system

The presence documents created from sources in the local presence environments are distributed to a central state agent using the method `PUBLISH` [RFC3903]. To protect the information carried in a `PUBLISH` message, we have set up a secure channel between the communicating peers using the TLS protocol. As a consequence, subscription management and the creation of watcher-specific views is done by the central state agent for this particular domain. The personal presence servers must pass the PIDF-XY documents generated from the aggregation script without filtering sensitive information because the subscribed watchers are known only at the state agent.

7.3 Evaluation

Due to the complexity of distributed real-time systems, no mechanisms exist to validate that the system behaves correctly at any point of time. Even if an appropriate specification for parts of the system is available, and only these isolated aspects are validated or—better—proved to be compliant to the specification. A conclusion on the system as a whole cannot be derived from these partial insights.

An alternative approach is the *evaluation* of the entire system by changing the input parameters and benchmarking the new overall system status. If done in a systematic fashion, different aspects can be checked together with the effect on other components or the system as a whole, respectively. The problem with this approach is to find a metric that can be used to determine the quality of the yielded results. For example, the presence service in Figure 7.11 aggregates the status information from three distinct presence zones for the user Alice. A subscribed watcher,

Jill, gets notified that Alice is currently `online` while another user, Bob, is blocked from getting notifications to his active subscription for Alice's status.

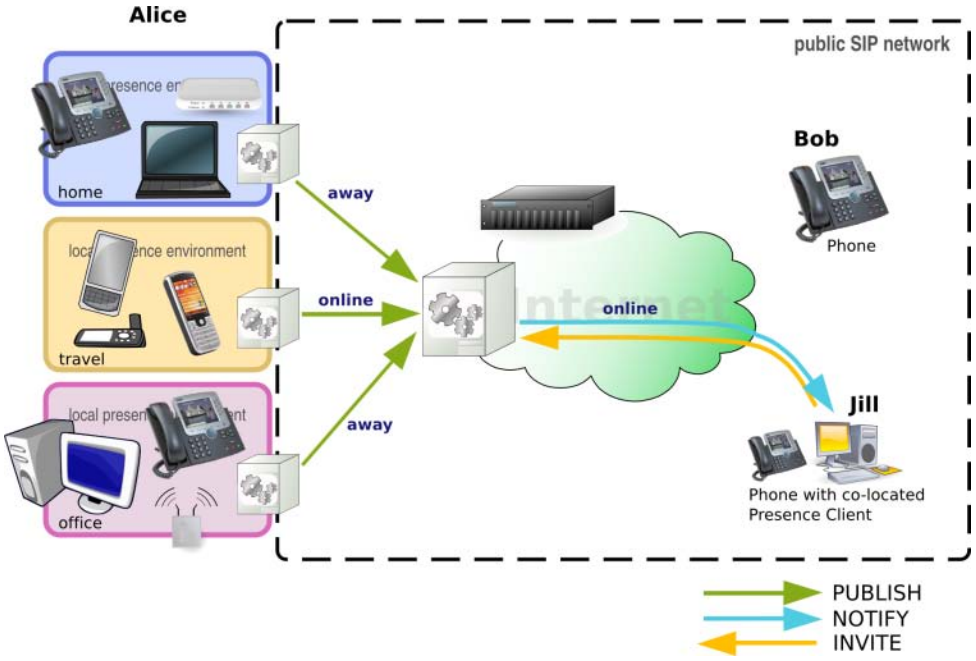

Figure 7.11: Aggregated information from a presence service

Looking at the status update notifications only, the notification of Jill seems to be correct (provided there is a mapping `m` of presence values that specifies `m(away,online,away)` to be `online`). The fact that Bob is blocked by a user-specific rule is not covered in this message-oriented evaluation scenario. Instead, the effect of user-provided aggregation rules must be considered as well. The use of a Turing-complete programming language for authoring aggregation specifications makes this more difficult, as the effect of rules authored in this languages cannot in general be predicted.

As a consequence, we were not yet able to perform an encompassing evaluation of our presence aggregation service. Instead, we have performed a number of tests with different scenarios and aggregation specifications for a small number of users as a first benchmark of the system's quality. The following section gives an example for a test scenario we have created for the research project *Presence Aggregation System* (PASST) to demonstrate the aggregation of presence information from multiple presence zones. Based on this test scenario, a review of our implementation is performed against the requirements of Chapter 3 to show that the system complies with the initial goals. Finally, we discuss possible strategies for further evaluation of the presence aggregation service and give some ideas for future research in this area.

7.3.1 Test Scenario

The test scenario described in this section is being used as a reference for evaluation of the presence aggregation service. The test scenario's main objectives are the demonstration of

- the aggregation of sensor data within the local environment, i.e. the generation and Mbus-based distribution of presence information,
- the configuration of a personal presence server, especially the installation and removal of aggregation specifications,
- SIP-based presence distribution, including the interoperability with third-party SIP user agents, and
- presence-aware call routing.

Section 7.3.1.1 and Section 7.3.1.2 give an overview of the reference scenario that was used to test the presence aggregation service, followed by a short description of the available user interface for script upload and management in Section 7.3.1.3. An example configuration of authorization classes and aggregation rules is explained in Section 7.3.1.4. The section concludes with a summary of our experience when performing the tests (Section 7.3.1.5).

7.3.1.1 Abstract Setup

The abstract setup for this scenario is shown in Figure 7.12. It consists of two distinct presence zones *office* and *home*, a presence-aware public SIP server, and a set of third-party SIP user agents. The presence zones *office* and *home* are part of the administrative domain "`example.net`" hosted by the public SIP server.[44] The local presence environments have been bound to the domain names `office.example.net` and `home.example.net` to indicate the tight relationship between local presence zones and the public SIP service. External SIP devices always use at least a different second-level domain name, e.g. "`wiptel.org`", making clear that requests from those devices originate in a different trust domain.

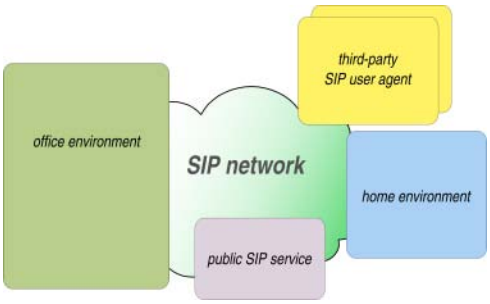

Figure 7.12: Abstract view of the demo scenario

Please note that these domain name conventions are a simplification we have introduced for this discussion only and that they are not required by our presence aggregation service. Instead, we anticipate that most presence zones will have domain names that are not related to the administrative domain of the public SIP server, even if operated by the same Internet Service Provider. Therefore, the authorization of requests and generation of presence information

[44] The domain names in this discussion are used only for illustration, specifically the domain name "`example.net`" which has been reserved by the IANA for this purpose.

always has to consider the successful identification of sender and receiver of a SIP message, which is usually done on the basis of authentication mechanisms defined in [RFC3261], or [RFC3325] for interconnections of closed SIP networks, respectively. As our main focus has been on the evaluation of the aggregation functionality, we have omitted the authentication of SIP requests, assuming that clients within the server's administrative domain are provided with credentials for a successful identification and have the permission to publish status update notifications for their own identity. External users must authenticate with their outbound proxy in its administrative domain as there is no global user database to check the identity of SIP users. SIP messages from these clients are sent to their outbound proxy, which then performs the necessary checks and eventually forwards them to our presence-aware SIP server as described in Section 3.4.

The communication of all components is done through the global Internet, i.e. all components have public IP addresses. If clients were located behind an intermediary that blocks requests from the Internet, care must be taken to keep a channel open where SIP notifications can be sent to the subscribed user agent. In particular, clients behind a packet filter or a network address translator usually send keep-alive messages in a regular interval to prevent the signaling connection with the SIP proxy for their administrative domain to be closed by an intermediary on the message path.

In this scenario, external users can subscribe to the presence state of a user who is located in the administrative domain `example.net` and initiate SIP calls with this user. The *office* environment as well as the *home* environment are presence enabled SIP environments providing multiple presence sources and a *Personal Presence Server* (PPS) that aggregates the presence information according to user-provisioned aggregation specifications. Depending on the subscribed user's personal preferences and the authorization configuration with respect to presence state distribution, the aggregated presence information is distributed to subscribers and is used for the call routing by the public SIP server.

7.3.1.2 Used Hardware and Software Components

The configuration of the network setup for demonstrating this scenario is depicted in Figure 7.13. The devices we have used for our tests are the following:

- A (simulated) hardware SIP phone with an integrated Mbus interface. The phone is capable of reporting its call state via Mbus to a control device such as the local presence server.
- A *Personal Digital Assistant* (PDA) that can be used to control or monitor the different presence sources in the environment via Wireless LAN.
- A user's mobile phone that can indicate presence by relying on short range radio communication (Bluetooth).
- A computer that hosts the personal presence server for the office environment, as well as a couple of software sensors for simulating additional sources for presence data.
- A Cisco 7960 hardware SIP phone that registers at the public SIP server from the home environment. As this phone does not generate any local presence information, there is no need for having a personal presence server in the home environment.

- A computer for running third-party SIP user agents as external clients. Most of our tests have been performed with the Microsoft Messenger, being one of the most popular soft phones with SIP-based presence support.
- A computer running the public SIP proxy and registrar. The integrated presence server manages subscriptions to known users and distributes presence information published by authorized clients or by the personal presence server located in the office environment.
- The components were connected to the local IP network of our test laboratory that provided basic services such as routing and name resolution.

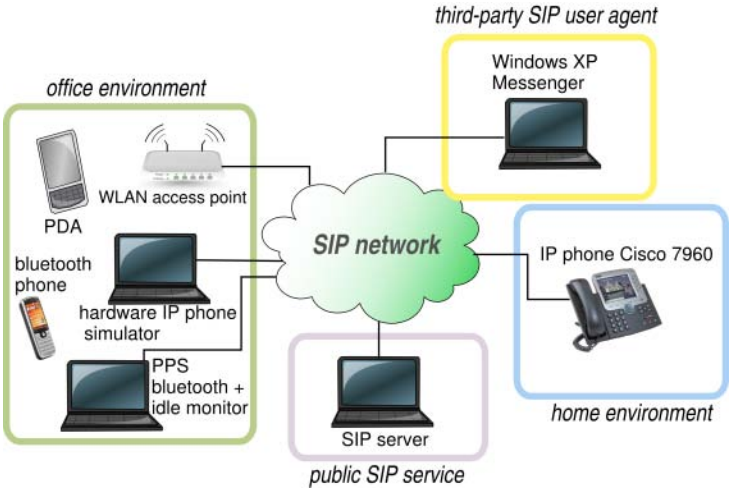

Figure 7.13: Network setup

The local presence environment in the user's office is an Mbus-based presence environment as described in Section 7.2.1. The presence information from multiple sources is collected and aggregated by the aggregation module as part of the personal presence server. The presence aggregation is governed by a user-provisioned aggregation specification that specifies how input from sensors is converted into a more abstract format that provides information on the user's reachability for communication.

The presence sources are Mbus-enabled services that use the *dynamic device association* (DDA) described in [KuOt03] to locate new sensors and associate them in a common Mbus session where they exchange presence information. The user's simulated hardware telephone is a SIP user agent that can be used to initiate or receive SIP calls. It is also a presence source that can distribute presence information via its Mbus interface. For example, it can signal the current phone state, i.e., `in-call` or `busy`. In addition, there is a set of simulated sensors that generally report on the user's physical presence, e.g., a keyboard activity sensor and a chair sensor. The bluetooth device detector detects the presence of a mobile phone owned by the user in order to determine if the user is in the vicinity of a specific bluetooth base station. In this demonstration scenario, the user is considered to carry his mobile phone anytime, so the presence of the phone is a very strong indication that the user is also present (keeping in mind

that we define presence information to be inexact by nature, so there might be exceptions to the given rule).

7.3.1.3 Authoring and Management of Aggregation Specifications

The implementation of the presence aggregation service discussed in this thesis was performed primarily to prove the feasibility of multi-step aggregation based on user-provisioned specifications. In this state of development, the usability of the authoring system for aggregation specifications was not in the implementation scope, hence PAL scripts must be edited by hand. As PAL is based on ECMAScript, existing developer tools for JavaScript could be used to create PAL specifications and validate the syntax of the scripts. For debugging of aggregation specifications, the PAM has an integrated command line interface that can be used to evaluate PAL script expression in the runtime environment of an aggregation server.

The installation of PAL scripts on a presence server is handled similar to the upload of HTML documents on a Web server. To simplify the configuration of presence servers, a Web-based interface WIPAM that supports the upload of PAL scripts was developed by Andreas Büsching in the research project PASST (see [OKB+04]).

WIPAM is designed as a standalone Mbus application that communicates with the presence aggregation module using the commands that were described in Section 5.4.4. Although the module offers a flexible template mechanism for dynamic inclusion of interchangeable content static in Web pages, the interface has been designed specifically to provide access to the configuration options of a presentity. As the configuration may have an immediate impact on the distribution of sensitive presence information, users must authenticate themselves with the Web server before access to the configuration data is granted.

In addition to the parameters that affect the runtime behavior of the presence server, the configuration section provides a management interface for the PAL scripts associated with the respective presentity. The interface lists the available scripts that have been uploaded to the server using the upload function of WIPAM. A single script then can be selected for activation at the presence server.

With this interface, the basic configuration options of the aggregation server can be set. In a future version, the interface may be enhanced by a simple mechanism for interactive combination of "macros" that represent simple PAL statements. After customizing these building blocks with user-provided parameter values, the building blocks form a basic aggregation specification.

7.3.1.4 Sample Aggregation Specification

In the following, we give two example specifications used for multi-step presence aggregation. The basic idea is that a user has defined two presence authorization classes, *public* and *protected*. Presence attributes associated with the class *public* may be published to any subscribed watcher, while *protected* information is restricted to friends and family. The authorization specification can be provided by uploading an `authdef` document together with the aggregation specification for the public SIP server. An example document defining the authorization classes *public* and *protected* is given in Example 7.3.1

The aggregation rules work as follows: When the user (Alice in this case) is in her office, the entire presence information from that presence environment should be published. But when at home, only the members of the authorization class *protected* should receive the details of Alice's

presence status. Any other subscriber should be notified that Alice is not at work and hence the presence status is `closed`. Example 7.3.2 shows the corresponding aggregation specification.

At the beginning of this specification, two local variables are bound to the presence channels *office* and *registration*. In addition, the output array `OUT` is initialized with a new presentity object for the current user. We use the expression `CURRENT_STATUS.entity` instead of `"pres:alice@example.net"` to allow for later re-use or change of domain names etc.

The custom rules are given in the `if`-statement following the initialization. If the local presence server of the office environment has signaled that the user is available, there is no need to check any other presence attribute. The corresponding channel object is literally copied to the output presentity as the only channel provided.

Example 7.3.1: Specification of authorization classes

```
<authdef>
  <authclass name="private">
    <authelem>alice@example\.net</authelem>
  </authclass>

  <authclass name="friends">
    <authelem>Bob@otherdomain\.com</authelem>
    <authelem>[^@]*@mysportsclub\.com</authelem>
    <authelem>jill@example\.net</authelem>
  </authclass>

  <authclass name="family">
    <authref name="private" />
    <authelem>philip@example\.net</authelem>
    <authelem>edward@example\.net</authelem>
    <authelem>anne@example\.net</authelem>
  </authclass>

  <authclass name="protected">
    <authref name="friends" />
    <authref name="family" />
  </authclass>

  <authclass name="public">
    <set-difference>
      <authref name="protected" />
      <authelem>.*</authelem>
    </set-difference>
  </authclass>
</authdef>
```

If instead the presence status of the office environment differs from `open` or is not known, the output object must be filled with two channel objects. First, a new channel object is created that takes the role of the office channel. The attribute `status` is set to "`closed`", with a substatus "`away`". The authorization class is set to "`public`". After that, the channel is added to the output object. The last step is to add the presence status description of the home environment as well. Here, the authorization class is explicitly set to `protected` to ensure that this information is passed only to members of that particular authorization class. The presence server then will omit this channel when creating PIDF documents for notification of subscribers from any class that is not *protected* or an authorization class referenced therein. The only information that might reveal the artificially generated presence information is the changed identifier of this channel. As the channel identifier is represented as the attribute `id` in the element `tuple` in

a PIDF document, it must be unique throughout that document and should be equal to the identifier of the corresponding presence tuples from previous update notifications.

While the aggregation specification installed at the server typically is used to control the dissemination of presence information and to provide a consistent view on inconsistent data from multiple presence sources, most of the status inference is done at the personal presence server. The Mbus-enabled aggregation engine receives low-level sensor data describing single aspects of the environment. The aggregation specification must provide rules to transform this data into an abstract status description. Therefore, most specifications begin with a declaration of local variables for the presence sources they know of as depicted in Example 7.3.3.

Example 7.3.2: Aggregation specification for public SIP server

```
var office = getChannel("office")          // presence status from office environment
    home   = getChannel("registration");   // IP phone at home

OUT[0] = Presentity(CURRENT_STATUS.entity);

if (office && office.status == "open")
  OUT[0].setChannel("1",office);        // forward status info from office
else {
  // Alice is not in her office, so appear closed or publish information from
  // her home phone, depending on the watcher's identity

  var result = Channel("office");
  result.status    = "closed";
  result.substatus = "away";
  result.authclass = "public";

  OUT[0].setChannel("2",result);

  // Add the registration status of the home environment as second channel to the output.  The
  // authclass "protected" ensures that this information is published to authorized users only.
  home.authclass = "protected";
  OUT[0].setChannel("1",home);
}
```

Example 7.3.3: Declaration of local variables for distinct presence sources

```
var phone = getChannel(/IP phone/i),              // hardware IP Phone
    reg   = getChannel("registration"),           // and its registration state
    bt    = getChannel(/bluetooth/),              // Bluez bluetooth sensor
    x11   = getChannel("xscreensaver");           // X11 idle time sensor

var result = reg;                                 // aggregation result
```

In this example, we have three devices that report presence-related data in regular intervals or whenever a status change was observed. The exact method depends on the implementation of the associated Mbus entity that transforms raw sensor data into Mbus messages. The hardware SIP phone, e.g., provides its own Mbus interface for sending regular update notifications indicating the current status of the phone. Additional update notifications are sent at any status

change, e.g. if the hook is lifted for placing a call. Other sensors such as the bluetooth sniffer or the screensaver daemon send notifications only in case of a status change.

The strategy we use to infer the presence status from sensors in the office environment is contained in a small set of `if`-statements shown in Example 7.3.4.

Example 7.3.4: Aggregation of sensor data

```
if (phone.hook == false) {

  // user is busy, just adapt reg and we're done

  result.status = "closed";
  result.substatus = "onthephone";

} else {

  if (bt.devices == "aphone" || x11.screensaver == false) {

    result.status = "open";
    result.substatus = phone.state == "idle" ? "online" : "busy";
    result.priority = "0.9";

  } else {

    result.status = "closed";
    result.substatus = "idle";
  }

}
```

Here, the hardware phone is exclusively used by Alice, and thus we assume Alice to be busy when the phone hook is lifted. In that case, only the status description must be adapted. If instead, the hook is in its place, the user's presence status must be inferred from the mobile phone we have said to be a nearly perfect indicator of Alice's presence at a specific place. As the bluetooth sensor reports only the name of the devices it detects, the corresponding channel's attribute `devices` is checked for the name of Alice's phone. If the device was found or the screensaver daemon has detected activity at Alice's desktop computer, the status description is adapted accordingly. If neither the phone is present nor activity at the computer was reported, the status is set to `closed`.

So far, the aggregated status is stored in a local variable only. To publish this information, the object has to be transferred into the output set as shown in Example 7.3.5.

Example 7.3.5: Setting the output channel

```
OUT[0] = Presentity(CURRENT_STATUS.entity);
result.id = "office";
OUT[0].setChannel(result);
```

A close look on our aggregation rules reveals that the registration status of the office phone is not considered at all. As a consequence, the phone may be indicated to be online (meaning

that it may be contacted) although no registration exists and therefore an `INVITE` message will never arrive at this device. Two possible solutions to this problem are:

1. Make sure that there is no attribute `contact` in the status description to indicate that this is a "blind" channel that does not offer a communication media.

2. Create a channel that indicates the status to be not available.

As the second option is more explicit we have selected it as the fallback if no registration is active. Example 7.3.6 depicts an `if`-statement that must surround the definitions of Example 7.3.4.

Example 7.3.6: Creating a default status description

```
if (reg && reg.status != "closed") {

  /* ... aggregation of sensor data ... */

} else {                          // last resort

  // create a default channel

  result = Channel("dummy");
  result.status = "closed";

  if (reg && reg.contact != "")
    result.contact = reg.contact;
  else {
    // redirect requests to our call center

    result.contact = Attribute("contact");
    result.contact.value = "sip:cc@example.net";
  }
}
```

7.3.1.5 Lessons Learned

Based on the test setup described in the previous sections, we have authored several aggregation specifications to explore the expressiveness of the specification language and investigate potential limits. Because of the flexibility of ECMAScript, we did not see any restriction caused by the language itself. A potential limitation of the object model was recognized when unknown extension elements were used in PIDF documents. Specifically, the flat object model does not yet support nested elements from unknown namespaces, as these cannot be parsed into an appropriate data structure without background knowledge of the underlying XML schema. Those elements therefore cannot be used in aggregation rules. An improved handling of XML structures built into the specification language therefore is advisable.

In addition, it was seen that the imperative programming style makes aggregation specifications more readable and—at least for experienced developers—easy to understand. Most of the complexity is introduced by the fact that a specification must have access to persistent information such as the current presence status of the presentity, while it is triggered from dynamic events. A developer of aggregation rules hence must have deep knowledge of the aggregation

server's distribution process to understand the effect of a specific rule when being evaluated with a given input document. The only development tool we have provided so far is a command-line shell integrated into the aggregation engine. With this shell, aggregation rules are evaluated on a given input document as they are entered. It provides access to the runtime environment, but no post-processing of output documents is performed. Thus, the runtime environment will not be updated after generating an output document.

As for any server-based scripting technology, there is only little support for debugging exceptions that occur at runtime, as the server provides no channel for giving immediate feedback to the rule author in case of errors. Even worse, errors during the aggregation process are quite crucial because they can finally lead to a complete failure of the presence state calculation, or lead to unwanted disclosure of personal presence information. To avoid this, authoring environments for aggregation specifications could be equipped with a *live track mode* to trace the execution of aggregation rules and deliver exceptions that have occurred while a specification was processed. A new SIP event type representing this debug information could be defined. Subscribed clients then would be notified whenever a relevant debug event was observed.

7.3.2 Review of the Initial Requirements

After having reported our experience from our tests with the aggregation service in the previous section, this section provides a detailed review of our system against the requirements stated in Chapter 3. We begin with the functional requirements in Section 7.3.2.1 as these assure that the system fulfils the intended functionality. Section 7.3.2.2 then reviews our implementation for compliance with the guidelines for presence application design we have postulated in Section 3.1. A brief summary of our evaluation concludes this section.

7.3.2.1 Functional Requirements

As described in this chapter, our implementation is built around a generic SIP protocol stack with specific extensions that allow for user-controlled presence aggregation and watcher-specific distribution of presence information. A few additional changes have been made to the code for routing of SIP `INVITE` messages to demonstrate presence-based call routing. Based on the requirements list of Section 3.2, the following overview shows how our implementation meets the functional requirements of a presence aggregation service:

Distribution of local presence status

Local presence environments are equipped with a personal presence server that disseminates the local presence status via SIP `PUBLISH` to a next-hop aggregation server, typically the public SIP server of the respective user's administrative domain. Status publication can be triggered explicitly via a dedicated command of the Mbus control interface, or implicitly by updating the presence status.

Presence status aggregation

Presence information is aggregated by user-provisioned aggregation specifications containing the required background knowledge to transform low-level data into abstract presence in-

formation. In our architecture, the aggregation process can be triggered at any presence server on the path between the status data's origin and the presence agent that manages presence subscriptions for the respective presentity.

To offer a more structured view on a person's presence status, we have enhanced the standard document format for conveying presence information with several attributes. These attributes can be used to denote distinct communication channels and describe their types. In addition, we have defined fixed vocabularies for available media types and for explicit status descriptions.

Publication of presence information

Our SIP protocol implementation supports the management of subscriptions and generation of event notifications as specified in [RFC3265], as well as the extension for event state publication defined in [RFC3903]. Specific support for user presence according to [RFC3856] and [RFC3859] is provided. The document format used in our implementation is backwards-compatible to the PIDF document format defined in [RFC3863].

Integration with a multimedia communication service

Being "an application-layer control (signaling) protocol for creating, modifying, and terminating sessions [including] Internet telephone calls, multimedia distribution, and multimedia conferences" [RFC3261], the Session Initiation Protocol intrinsically provides the signaling functions that are necessary for managing multimedia communication. Any SIP server in our architecture can be co-located with a registrar and a proxy to enable routing of arbitrary SIP messages according to [RFC3261].

Configuration interfaces for user-specific aggregation rules

Our implementation of the aggregation engine provides several interfaces for controlling aggregation specifications and for modification of global parameters for the runtime environment. The most intuitive interface is based on HTTP, providing Web-based dialogs for upload and deletion of aggregation specifications, or the modification of variables. In addition, all aspects of the aggregation engine can also be controlled via the Mbus interface, which is especially useful to add new protocols such as the *XML Configuration Access Protocol* [XCAP].

7.3.2.2 Usability

As the presence aggregation service is just one part of a more complex application—distributed real-time multimedia communication—, our review was performed in this specific context. In particular, the aggregation system implements a technical service that requires a certain amount of background knowledge to be used as intended. The usability aspects that have been investigated therefore focus primarily on the *robustness* of the aggregation language against configuration errors and the quality of information provided to enhanced multimedia communication services. The review is based on the guidelines we have developed in Section 3.1.

Define a standard vocabulary with a flexible extension mechanism

The basic vocabulary used for presence status descriptions defined in [RFC3863] provides an element type `basic` with the possible values `open` and `closed`. We have defined an additional attribute `substatus` in our object model that holds extended status descriptions that are read from an input document or set explicitly in an aggregation rule. The substatus attribute is mapped to an XML element that follows the element `basic` within the contents of `tuple` in a PIDF document. The exact mapping depends on the document type of the created output document. The proprietary document format of the Microsoft Messenger e.g. accepts the attribute `msnsubstatus`, while PIDF requires another namespace to be used for this element.

Due to the generalized definition of `substatus`, there is no need to prescribe a fixed vocabulary of presence status values. Instead, we use the keywords that are accepted by Microsoft Messenger clients in the attribute `msnsubstatus`, and thus assure backwards-compatibility with one of the most popular applications for SIP-based presence.

Provide support for uncertain status values

Our extension to the standard PIDF document format provides two mechanisms that are used to express the uncertainty of a given status description. The first is a presence attribute, `ownership`, indicates the degree to which a described communication channel is owned by a particular user, i.e. which relevance this information has in the context of the entire set of presence attributes.

The second mechanism consists of three XML attributes, `decay`, `timestamp` and `threshold`, that can be used with any presence attribute in a PIDF document. The attributes describe the decreasing dependability of the information they are associated with, starting from the given point of time until the specified threshold is passed. Aggregation engines can use this information to enforce re-calculation of the presence status even if no status update notification has been received.

Use a global presence data model

Our extensions of the standardized PIDF document format as well as the aggregation language's object model follow a concise semantical model that treats the `tuple` elements of PIDF documents as descriptions of communication channels rather than device-specific descriptions. A `tuple` element may be augmented by additional information on the channel's type (using abstract keywords that describe the channel's role in the particular environment), and the offered media type. The availability of users then is always combined with a description of the media that is available for communication.

Let users keep control of their communication relationships

This requirement includes two contradicting aspects of automated communication establishment that must be addressed at the same time: First, the calling user should be able to select a communication media of his choice, and second, the called user must be able to block call attempts based on the caller identity or the media being used. The generation of watcher-specific views on a user's presence status helps solving this trade-off as the presence service can be configured to present only those channels that are accepted for receiving calls at a particular time.

For example, a user who is in a meeting may have configured his presence service to show the phone status (more exact: "the media channel phone which is an interactive audio channel") as busy. For family members and colleagues, an additional channel of type "interactive text" is offered where the user can be contacted even in the meeting.

Generate alternate views specific to pre-defined subscriber groups

This requirement is met by associating channel descriptions in the output document with specific authorization classes. When the aggregation result is processed, the presence server generates a set of presence documents that are specific to a particular class of watchers. Each document contains only channel descriptions that are marked with the authorization class a particular watcher belongs to. The resulting documents then are sent to their specific class only, providing distinct views on distinct groups of subscribers.

Define an expression language for flexible filtering and throttling of notifications

Our implementation of the aggregation process currently does not provide explicit support for filtering incoming messages or throttling outgoing status notifications as both functions can be achieved through appropriate aggregation rules.

Allow for partial status updates

This is an intrinsic requirement for presence aggregation servers as update notifications originate at distinct sources and thus always contain only partial information. The aggregation server therefore stores the results of the aggregation process and treats received status updates as a patch to this information. If an aggregation specification is installed, it is processed with the current status and the received status update as its input.

7.3.2.3 Summary

To determine the quality of the design and the implementation of the presence aggregation service discussed in this thesis, we have performed a review against the catalogue of requirements that was defined in Chapter 3 as a working program to improve the quality of distributed multimedia communication services. The set of requirements is divided into two parts, *functional requirements* and *usability guidelines*.

The functional requirements aim at the feature completeness of the provided service infrastructure and thus must be met entirely. The review of our reference implementation has shown that all required features are implemented and work as specified. The aggregation service was designed such that no changes to the Session Initiation Protocol or its standardized extensions are necessary. The extended document format for conveying presence information through the Internet is backwards-compatible to [RFC3863] and does not prevent the use of other standards-compliant extensions to this format.

The usability guidelines ensure that the presence service delivers useful information for improving multimedia communication services. The review has revealed that our aggregation model addresses the guideline's recommendations in a standards-compliant and extensible way, based on the reference scenario that was presented in Section 7.3.1.

In summary, the review has shown that the presence aggregation service was built in compliance to our working program. To prove that presence aggregation is an appropriate solution regarding the initial problem statement *improvement of multimedia communication*, further investigation is required. In the next section, we give an overview of additional evaluation strategies that can be applied to determine the quality of presence aggregation and its applicability in existing working environments.

7.3.3 Further Evaluation Strategies

The test scenario we have described previously was created primarily for evaluation of the aggregation concept discussed in this thesis. The implementation therefore has been a proof of concept rather than a presence service that can be used without changes in a productive environment. Before deploying this service, we suggest further evaluation especially to determine the robustness of the aggregation language against malicious use, and to investigate graphical authoring environments for aggregation specifications. To enhance the user acceptance of presence services, the evaluation must address the following criteria:

- *robustness*

  Defines the impact that local changes to an aggregation specification have on the generated output. In the best case, changes affect only the objects on which these changes were made.

- *predictability*

  Gives an indication whether the result of a change made to an aggregation specification meets the expectations of a script author. In the best case, the actual result is equal to the anticipated result.

- *expressiveness*

  Indicates the level of details that can be changed in a presence document. While a document transformation allows to modify presence documents at the level of the XML information set, a domain-specific language might restrict changes to the objects defined in a particular ontology for this domain.

- *granularity*

  The granularity depends on the maximum frequency at which update notifications can be sent and is related to the level of abstraction provided by a particular presence vocabulary. If status changes occur faster than they can be processed by the presence service, some update messages might get lost, causing an inconsistent view on the presence status.

- *performance*

  A technical indicator that denotes the responsiveness of a presence service. If update messages are lost due to overloaded network components (including the aggregation server) or presence messages are deferred by the slow aggregation process, users might lose confidence in this technology.

This section addresses possible strategies for evaluating the implemented aggregation service. We refer to the methodology that has been used in related research projects in Section 7.3.3.1. Section 7.3.3.2 then discusses the requirements on the user interface of a simulation system that could be used as a playground for authoring of aggregation specifications. Finally, Section 7.3.3.3 discusses the performance implications we have seen during our tests, and gives a rough estimation of the cost per user for presence aggregation compared to a traditional presence service.

7.3.3.1 Field Studies

In the 1990s, several field studies have been performed to investigate the impact of groupware systems and computer-supported collaborative work on the productivity of users in their work environment. As presence services are used only as enabler technology for this type of applications, an isolated evaluation is hardly possible. The presence information hence is always related to the current environment of the principal. The studies listed in this section therefore address primarily the work environment where users are inclined to accept observation technologies more than in their home environment.

A possible strategy for the evaluation of context-determination is described by Wang et al. in [WCK04] for their experiments with the Solar system (see Section 4.1.1). The authors use two types of sensors to detect if a meeting is in progress in a specific office. The sensors detect motion of chairs and pressure on the seat of each chair. The data reported by these sensors is aggregated to a boolean value that indicates if there is a meeting.

To determine the quality of their aggregation function, the authors have defined possible metrics to measure the accuracy of the function's result (i.e. if reflects the real status within that particular office). In addition, the timeliness of reported status changes has been recorded to determine the time span between a real status change and its automatic detection. These metrics now have been applied to different event logs that were created during the test runs. The first was the *meeting log*, the reference of real meetings held in that office. The meeting log has been created by the chair person of the meeting who had to record start and end times manually by pressing a button on a dedicated logging device. Any record with inconsistent data (i.e. missing start or end events) has been discarded before evaluation.

The manually created meeting logs then were compared to the results of the aggregation function using the given metrics. The output of this process then can be used to analyze the factors that have an impact on the correctness of the aggregation result.

A similar approach is discussed by Hudson, Fogarty et al. in [HFA+03, FLC04b] for evaluating a sensor-fusion system to predict the interruptibility of users' current activities. The authors have performed a *Wizard of Oz* study to determine the conditions in which users report themselves to be interruptible. The study has combined an offline analysis of audio-visual recordings from the working environments of these users with self-reports of interruptibility that have been prompted at specific intervals. The results of this study have been analyzed with statistical methods to determine the deviation between self-reports and automatically detected information on the users' interruptibility.

The approach of Begole et al. described in [BTS+02] uses activity diagrams that have been synthesized from the presence information delivered by their test candidate. The data basis for these diagrams results from automatically collected data on the interaction of specific users with their computer. Besides keyboard and mouse activity, the logs contained information on

the users' location and scheduled activities from their calendar applications. The collected information then has been aggregated into a diagram that displays activity times for specific users together with the location detected by the presence system to evaluate (called *Awarenex*). Figure 7.14 shows an example graph generated from this information. The created graphs have been compared to the actual locations and activities of the users according to their meeting schedules.

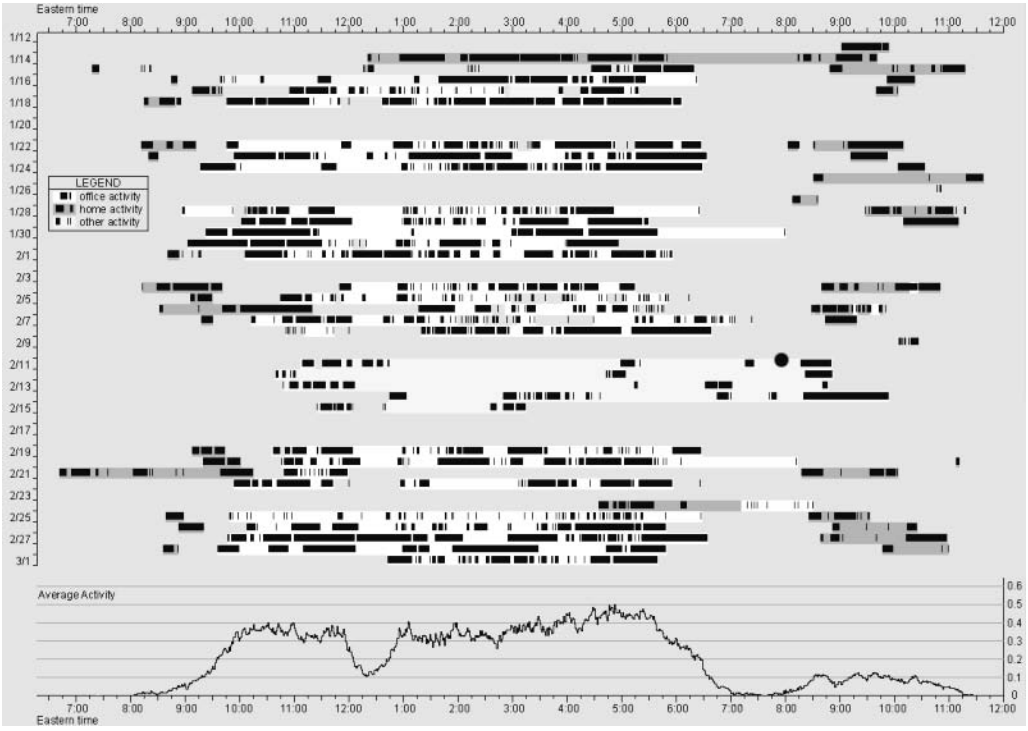

Figure 7.14: Activity diagram created from Awarenex presence information (source: [BTS+02])

The examples for evaluation strategies indicate that no absolute metric for the quality of presence information exists. The results of the presence aggregation process therefore must be evaluated in the context of a user's habits and preferences.

7.3.3.2 Simulating the Presence Service

Besides the offline analysis of presence information that is described in the previous section, presence events can be simulated to determine the effect of the aggregation service. In [Ran02] Rantanen describes a testbed for simulation of presence events he has used to evaluate a specific presence service. The simulation includes the generation of random events pretending the presence or absence of persons with a certain probability.

The evaluation is based on a specific presentity whose activities are controlled by a personal calendar that can be changed from the simulation environment. Events are induced by the transition between activities or by traversal through simulated locations (e.g. when returning from work). A certain amount of entropy is added by annotating a scheduled calendar task with a probability of occurrence (the simulated principal may "forget" a task).

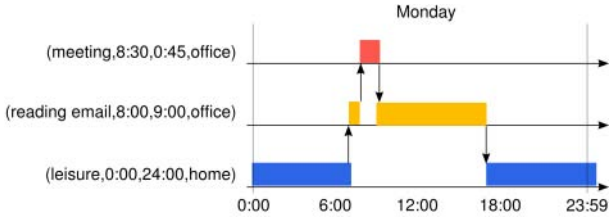

Figure 7.15: An example activity schedule (source: [Ran02])

Events in the simulation model are derived from activities that consist of a quadruple `(activity, time, duration, location)` as illustrated by the example activity schedule in Figure 7.15. To resolve conflicts caused by overlapping activities, each specification has a priority value. Activities with a low priority then can be interrupted by activities with higher priorities. The transition is observed as an event by the presence service.

The simulation model proposed by Rantanen allows for flexible creation of event traces that trigger a presence service. To evaluate the event aggregation service, a reference schedule could be processed with different levels of background noise and different aggregation specifications. If combined with a graphical user interface, the presence simulation environment even can be used for authoring of aggregation specifications.

In [Har04], Hartmann also gives an example for a simple simulation environment that allows for modification of parameters for pre-defined aggregation functions. The simulation environment contains a number of virtual sensors and can optionally be augmented with real sensors to test aggregation specifications with real-time data. A benefit of Hartmann's system is a virtual time axis that can be adjusted to fasten or slow down a test run. Unfortunately, this simulation environment cannot be used with existing presence services as the timer scheduling of the SIP protocol implementation and the PAL aggregation engine cannot be adjusted this way.

7.3.3.3 Performance

Besides the quality of aggregation results that was addressed in the previous sections, an economically interesting aspect is the performance of aggregation servers and the related question of scalability. Benchmarking of presence servers is a difficult task as there is not much experience with decentralized Internet-scale presence services. Depending on the acceptance of the presence service, watcher list may become very large and presence events may occur more frequently. In this section, we try to estimate the impact of presence aggregation based on a few assumptions on the average number of watchers and the frequency of status updates.

As a reference system we have used an *HP Proliant DL360G5* system with four CPU cores and a CPU clock frequency of 1.6 GHz. The server was equipped with 5 GB RAM that allowed for a concurrent registration of one million SIP users with a registration expiry of 1800 seconds. To benchmark the system, we have created three million concurrent presence subscriptions for 200,000 different users (i.e. every presentity held 15 subscriptions).

Based on this assumptions, we found that registrations and subscriptions require 1 kB of RAM each. A subscription update was handled by the system in less than 2.5 ms in average, resulting in 400 `SUBSCRIBE`/`NOTIFY` pairs that can be processed by a single CPU core per second. Assuming that a presentity's status changes once every three minutes in average (e.g.

because the principal takes the phone off hook and places a phone call with an average duration of three minutes), a single CPU core could handle 4500 users. If the estimated average cost for a server was 100 $ per month (according to the current rental costs for managed servers offered by Internet service providers these days), the average cost per user was about 0.03 $. Table 7.1 lists the results of our benchmarks and estimations as described before.

RAM per registration	1 kB
RAM per watcher	1 kB
average processing time per subscription	2.5 ms
users per CPU core	4500
average cost per user	0.03 $

Table 7.1: Reference benchmark for presence services

To determine the cost of presence aggregation, we have performed a number of benchmark tests against our reference implementation. As stated before, the main objective of our implementation was the demonstration of the aggregation service's functionality and the development of code that can be re-used in other research projects where high performance is not a primary goal. Because of this, we have benchmark-tested some functional components in isolation to yield comparable results. In particular, the evaluation of PAL scripts has been tested apart from the handling of Mbus messages and creation of SIP notifications.

A major drawback of the existing implementation is the exclusive use of the JavaScript runtime environment with a single context object for each status update of a specific presentity. The presentity structure thus requires additional 2 kB for the runtime environment and a minimal stack frame. To store the text version of the test script, we have added another 1 kB to the presentity structure. Thus, every presentity requires at lest 4 kB RAM storage. Since watchers may be provided with a custom view on the presence status, additional memory must be allocated for watcher-specific notifications. In average, this might eat up another 1 kB, resulting in 2 kB RAM required for each watcher structure.

The first observation from these figures is the increased amount of RAM required for each presentity. For 200,000 users, the memory allocation goes up from 3.2 GB to 6.8 GB (again assuming 15 watchers per presentity). To estimate the evaluation time for a specific script, we have performed a repeated evaluation of example scripts from Section 6.4. In these tests, we determined an average evaluation time for typical aggregation specifications of less than 20 ms.

RAM per registration	4 kB
RAM per watcher	2 kB
average processing time per subscription	20 ms
users per CPU core	560
average cost per user	0.18 $

Table 7.2: Performance benchmark for presence aggregation

In Table 7.2, we have collected the results of the benchmark tests with the presence aggregation service. Starting with the assumption that a script run does not take more than 20 ms,

we found that a single CPU will be able to handle about 560 users. Because of this small number, the increased amount of necessary RAM storage compared to the reference system can be neglected when estimating the cost per user of 100 $ / 560 = 0.18 $.[45]

The performance benchmarks indicate that the use of presence aggregation causes a significant increase of the average cost per user. A business model may justify this additional cost if the revenue is above that margin and maximum values can be given for the number of watchers per user to ensure linear scaling of the presence service. If this is the case, the mechanisms sketched in Section 8.4.5 can be used to increase the number of users that can participate in this service.

7.4 Summary

This chapter has given a brief overview of the major components we have developed to show the feasibility of the presence aggregation concept that has been discussed throughout this thesis. The primary goal was the creation of modular building blocks that implement the required functionality based on standardized architectures and protocols. The C++ implementation has been split into numerous libraries providing flexible classes and functions for asynchronous server applications. In particular, a generic server abstraction layer was constructed to integrate the selected Message Bus (Mbus) implementation with our event-based API for controlling server applications.

The central components of our implementation are the aggregation engine and a SIP protocol gateway. Both modules provide an Mbus API to enable the combination with other components of the presence service. As a result, the SIP protocol gateway can be co-located with an Mbus-based registrar and a SIP proxy controller to yield a standalone presence server. Adding the aggregation engine and our HTTP protocol gateway results in a full-featured presence aggregation server. Without SIP registrar and proxy controller, the setup can be used as a personal presence server to aggregate presence information within a single presence zone.

The implementation has been tested with a typical scenario that was motivated by the use case in Section 2.6. The local presence environment therefore has been equipped with several sensors, i.e. demons that publish raw sensor data on the Mbus for aggregation by custom PAL scripts.

Finally, we have summarized the lessons learned and pointed out possible strategies for further evaluation of the aggregation service to improve the system's usability and the quality of user-provisioned aggregation specifications.

[45] The cost per user for the reference system without presence aggregation has been rounded to the next cent value for readability. The exact number reflects the ratio $\frac{1}{8}$ of processing times and users per CPU.

Chapter 8

Conclusions

In this thesis, we have developed a framework for aggregation of presence information on a syntactic and semantic level. The two major objectives of our work were to give users control over the entire aggregation process and to use open and extensible standards for representation and distribution of presence documents. In summary, our research work has addressed the following goals, which have been discussed throughout this thesis:

- The first step was to perform a literature analysis to determine the expectations users have on computer-supported cooperation systems, followed by an investigation of documented socio-technical issues of existing groupware systems. Based on the assumption that interactive communication always requires a common set of communication channels between peers as well as their availability and willingness to communicate, we came to the conclusion that *social presence* is essential for interpersonal communication over the Internet.

- Based on the results of our analysis, we have determined the technical requirements on a presence service to leverage interpersonal real-time communication over the Internet. The envisioned presence system should offer social presence for human users independent of the devices or media they prefer. Fundamental requirements were the aggregation of presence documents from multiple sources in distinct *presence zones* as well as the modeling of degrading accuracy of presence values over time.

- The degradation of presence values was an outcome of reasoning about the quality of aggregated presence information and the overhead that is necessary to get presence status records up to date. Starting with our work on distributed presence services, we found that the standardized *presence information data format* (PIDF) did not contain sufficient information for automated re-calculation of presence status records based on the decreasing exactness of input values over time. In particular, the XML-based document format defined in [RFC3863], which is a common interchange format for presence information in the Internet, gives no indication of the exactness of listed presence values. Hence, updates of presence status records at a presence agent always require the entire aggregation process to be triggered by status update notifications from low-level data sources.

  To facilitate automatic re-calculation of presence status records at any level within the hierarchy of aggregation servers, our extensions should provide additional information on the quality of given presence values in a PIDF-document and allow for modeling the

decay of numeric values over time, together with a lower bound indicating when the data has become useless.

- Beyond its technical features, we wanted our presence service to be controllable by the users whose status information is disseminated. A powerful, but easy-to-learn specification language was invented to define rules for transforming low-level sensor data into more abstract presence status descriptions and to control the amount of presence information that is disclosed to specific watchers.

- The final objective was to develop an implementation of this presence service to prove the applicability of our concept even for existing presence infrastructure components. An important aspect of this design was the flexible integration of different types of low-level sensors in a local presence environment. A lightweight communication protocol should be used to facilitate the implementation of avatars for dumb sensors that provide no network connectivity at all.

In the following, we summarize the main conceptual achievements in Section 8.1, and we describe our main engineering results in Section 8.2. In Section 8.3, we compare our presence aggregation framework to other presence services. Section 8.4 then lists a number of open issues and possibilities for future enhancements of our approach, and shows the next steps we envision on the lessons learned so far.

8.1 Conceptual Achievements

Starting from our analysis of literature on existing computer-aided collaboration systems, we have identified the limitations and drawbacks that impeded global deployment of presence-based conference systems. An important result of the existing research efforts in our opinion was the observation that user acceptance of computer-supported groupware systems is still very low because of the systems' inflexibility and their lack of user control. Especially presence systems are criticized for their intrusiveness and missing transparency of what information is provided to others.

Addressing these problems, we have argued that *social presence* at Internet-scale can be provided only if users can participate in the creation of "their" presence data, and have control over the dissemination process. As a result of increased user acceptance, the presence service will offer higher data quality in terms of accuracy and reliability. Presence-aware applications such as interpersonal real-time communication (e.g. "multimedia conferencing") hence not only benefit from better call-completion ratios, but also from additional possibilities for user interaction.

To give users more control over the dissemination of their presence status, we introduced the concept of *presence aggregation* and developed a powerful language for specifying user-specific distribution rules for personal presence information. This specification mechanism is designed such that users with a certain background knowledge of scripting languages for the World Wide Web face a shallow learning curve and can easily understand how their authored aggregation rules will be processed.

We have shown the explicit rule language to be especially useful for high-level aggregation of abstract presence information while other techniques such as sensor-fusion are appropriate

for processing low-level data, e.g. numeric sensor values. Mathematical operations thus are a good choice when the operations' domains can be ordered, and when the result of each operation again is a member of that operation's domain. User-provided processing rules, in contrast, should be used if the domain is a discrete set with every element having certain semantics without a formal relationship to the other values of this set, i.e. there is no order of the domain.

An important result of our work is the observation that both approaches—mathematical fusion of low-level data and high-level aggregation of abstract presence information—can be used in combination to get the best of both worlds. Presence aggregation therefore must be seen as a multi-step process that extends over several levels of abstraction. Care must be taken that no accidental information loss occurs when going up from one level to another. As information items may be in conflict to each other, user-provided rules give a good indication which information item must be kept and which item is less important and thus can be neglected.

User-provided rules for processing presence information can also help solving the intrinsic conflict between information detail and user privacy: While the quality of the presence service grows with the higher detail of abstract presence information being disclosed to the public, the user whose sensitive personal data is exposed has an interest of getting as much privacy as possible. On the other hand, the less information is known about the user's current activities and habits, the more intrusive the communication attempts will be. The approach we have discussed in this thesis therefore enables the user to decide which level of privacy loss is acceptable to achieve a certain quality of presence service. Doing so, the information system is actively controlled by the user and hence will be less threatening.

Related to the trade-off between privacy and information detail, it is accepted that every user has a personal privacy policy that cannot be generalized. For example, one user might allow for his family to be notified about any status change during office hours, while other users might not want their relatives to know any details about their work day. To address this problem, we came to the conclusion that Internet-scale presence services must support user-specific notification policies, i.e. presence agents must be able to generate watcher-specific views on a principal's presence status record.

The multi-step aggregation of presence documents and the demand for watcher-specific views finally made clear that the standardized document format for conveying presence information over the Internet, PIDF, lacks important meta-data. In particular, most presence tuples need to be tagged with additional information on the exactness of the data represented as well as their degradation over time. Moreover, user-specific policies typically require information on user classes, black-lists, etc. to be carried in presence documents as well. To address both issues, we have worked on a number of enhancements to the PIDF document format to carry the additional information without restricting the use of other standardized PIDF extensions that provide more detailed presence values than the basis specifications.

Although call routing has not been in the inner focus of our work, we have shown how aggregated presence information may be used by presence-aware SIP servers to enhance the user lookup process. For example, if the presence information yields that a called user is currently only present in environment A but not in environment B, then `INVITE` messages could be routed to environment A only. This approach is preferred over more involved approaches such as using the *Call Processing Language* (CPL) [RFC3880] for the following reasons:

- In order to be effective, a CPL and presence enabled SIP server should be located in the public space, e.g., operated by a public SIP operator. However, the whole idea of pres-

ence aggregation pertains to the local domain, e.g., the living and working environment of the user, because the user has to provide specific knowledge—in form of an aggregation specification—to the presence processing system in order to achieve an accurate calculation of the presence state. The aggregated presence information (with different authorization classes for different watchers) is then forwarded to a public SIP server, where it might be aggregated with presence from additional environments of that particular presentity and then finally delivered to the subscribed watchers.

- The whole idea of presence aggregation is thus motivated by *hiding* information—either because it is only relevant for the local presence state calculation or because of privacy issues, i.e., when the corresponding watcher is not a member of the proper authorization class. In order to make presence information useful for CPL-based call processing, as much details as possible should be disclosed to the CPL-enhanced SIP server. Unfortunately, this would essentially require to *reverse* the presence aggregation in order to obtain the necessary level of detail. In any way, the presence-aware CPL must be provisioned with a CPL script that provides evaluations of detailed presence state, which might already be considered as a security breach. In summary, in our architecture, the information hiding approach pursued by the presence aggregation concept conflicts with the CPL approach of performing extensive tests on presence state in order to determine the call routing.

Based on the conceptual insights we have achieved during our research work, a new presence service was designed to give users better control over their personal information and its publication to subscribed watchers. The lessons learned from implementing this new presence aggregation service as a proof of concept are shown in the following section.

8.2 Engineering Results

In this thesis, we have discussed a number of application-layer communication protocols, control languages and document formats we have developed to prove the applicability of our concepts in an existing presence environment. In the following, we give a brief overview of the systems that are part of our work.

8.2.1 Component Architecture for Local Presence Environments

The modular architecture we have developed for collection and dissemination of presence information in local presence environments has been evaluated in several research projects. The telephony-oriented projects *Desktop Telephony Integration* (DTI) and *Functional Enhancements using External Telephony Applications* (FETA) were the first projects that have used our modular, Mbus-based SIP proxy. [OKB+02, OKB+03, Kut03]

To simplify the development and integration of new sensor modules within the local presence environment, our application design follows the *Mbus Guidelines for Application Profile Writers* defined in [Kut01]. Our implementation of the communication patterns of [Kut01] is available as an extension to the C++ reference implementation of the Message Bus, and was

demonstrated to be interoperable with the existing python-implementation of this specification as well.[46]

The SIP proxy implementation as well as the Mbus modules benefit from a strict separation of functional components, split up into a set of generic C++ code libraries. Later projects such as the *IPv6 Wireless Internet Initiative*[47] (6WINIT) or *Geo-based Services Enabling Cooperation* (GEOCOOP) [OKB+05] have re-used some or all of these code libraries.

The research project *Presence Aggregation System* (PASST) [OKB+04] has shown the applicability of our presence aggregation service to an enterprise communication system, with a special focus on geographically distributed organizations. The PASST presence service implements the multi-step architecture discussed in this thesis, with local presence environments that consist of presence information sources and a personal presence server to provide the aggregation of the different presence state descriptions. The presence sources within a local environment can be discovered dynamically and brought into a common Mbus session by a controlling device, such as a user's PDA or smart phone. Presence information is represented in a native representation that allows for efficient distribution over the shared multicast bus. For distribution over SIP, the data is aggregated and then converted into the (enhanced) PIDF format.

8.2.2 Object Model for Presence Documents

Having extended the basic representation of presence information by additional attributes to show the exactness of presence values, we have developed a presence server that interprets this additional meta-information and generates new presence documents whenever certain threshold values are passed. As the processing requires a concise internal representation of a principal's presence status, we have developed an extensible object model reflecting a presence document's semantics (rather than the pure syntax). The object model includes a well-defined set of data types for channel attributes with implicit processing rules, i.e., for later aggregation. In addition, it honors the authorization concept that allows to define authorization classes such as user groups and to associate channel definitions with authorization classes in order to control the distribution of presence information to subscribed watchers.

8.2.3 A Language for Controlling Presence Aggregation

Besides the static object model for representing presence information during the aggregation process, we have defined and implemented a language for controlling the dynamic aspects of this process. A presence aggregation server takes presence documents generated from one or more presence sources together with a user-provided set of processing rules and generates zero or more output documents to be sent to the subscribed watchers.

Our implementation of the aggregation language is based on the standardized ECMAScript language which has been extended with native objects that represent the logical components of a presence document as described in the previous section. A number of access functions and operators for these objects have been provided as well to simplify authoring of aggregation rules.

[46] The suite of Mbus reference implementations is available online at <http://www.mbus.org>.

[47] See <http://www.6winit.org>

The language interpreter can access attributes from channel definitions of input documents and can generate new channels with arbitrary attributes. The presence processing is not limited to a set of given transformation rules, but may rely on the complete feature set of ECMAScript and its provided function libraries.

The environment for processing the aggregation rules supports the PIDF extensions we have described in this thesis and provides for easy integration of additional presence vocabularies such as the rich presence extensions defined in [RFC4480]. In addition, a history concept is supported that allows access to previous input and output documents; one application of the history is the calculation of decay functions that increase the degree of fuzziness depending on the age of a presence value.

Due to the presence server's modular architecture, the aggregation engine can be replaced by any other language processor implementing the given object model, or switched off entirely. The engine is an Mbus component and provides an Mbus interface for control, for script provisioning, for managing watcher information, and for sending and obtaining presence status information. The Mbus interface can be used for Web-based configuration, for providing presence input to the presence aggregation engine, and for receiving presence output for later distribution.

8.2.4 Distribution of Aggregated Presence Information

As the integration with existing SIP components (i.e., proxies as well as user agents) has been a major goal of our work, special care was taken to follow existing IETF standards. Therefore, we have used the *Session Initiation Protocol* (SIP) for wide-area distribution of enhanced PIDF documents, as this protocol is widely deployed and has clear semantics of presence-related operations according to the definitions of [RFC2778] and [RFC3859].

Our implementation of SIP-based presence includes a presence agent with a co-located SIP proxy/registrar combination as well as a presence user agent for generation of status change notifications according to [RFC3265]. Any of these SIP components can be used either as a stand-alone application or as a SIP protocol module in a distributed Mbus application. When acting as an Mbus module, the SIP protocol stack is controlled by an external module via its integrated Mbus interface, parts of which have been described in Section 5.4.4. Example projects that made use of this function are described in [Har04] and [OKB+05]. In addition, our SIP stack is being evaluated as a technical platform for centralized multimedia conference services and for sending change notifications of Internet Media Guides.

While our application honors existing standards for message transport as well as the contents of published presence documents, many presence clients still use older formats that do not conform to the standardized *Presence Information Data Format* (PIDF) defined in [RFC3863]. To allow for interoperability between our presence aggregation service and those SIP user agents, we have implemented a number of additional transformation steps when distributing presence information to user agents that do not support this standard format. In particular, we have integrated an XML transformation engine to perform on-the-fly conversion of the widely deployed presence format of the *MS Messenger* to and from PIDF.

8.2.5 Presence-Based Call Routing

Having knowledge about the presence status of a principal, a SIP proxy server might use this information for routing incoming `INVITE` to an appropriate destination. In our tests, we have investigated the applicability of presence information to dynamic call routing by replacing the user lookup function of our SIP proxy server with a presence module. Whenever an `INVITE` message is received, the presence module creates a target set for these messages according to the presence status of the called user. Initial tests with this message routing based on dynamic availability information have shown that presence information in near future may become a valuable resource for more efficient call routing, and hence may lead to a better call-completion ratio and shorter call setup times.

To protect the principal's privacy, the routing engine must honor the authorization rules that have been provided with the presence aggregation specification. Incoming `INVITE` messages then are forwarded only to destinations that are indicated to be available in the watcher-specific view on the internal presence status. For example, when the principal is at home, incoming calls would be routed there as the presence service has determined this as the user's current location. Now, a user may want to hide the information that he is at home from co-workers and unknown persons (such as customers of the company the user works for). A specific aggregation rule thus would generate a channel description denoting absence from home and publish this information to any subscriber who is not contained in the authorization classes for friends and family.

Given this example, sensitive information would be disclosed if SIP messages generated by one of these blocked users were routed to the home phone. Even if the call was declined, the caller would know that the presence information is inexact and might even guess additional information about the principal's home network. Systematic calls to this address then could be used to infer the principal's presence status although the subscription has been blocked.

8.2.6 Securing Presence Information

Presence documents containing sensitive information are secured to avoid passive eavesdropping and active man-in-the-middle attacks. When using a state agent that handles subscriptions on behalf of a presence source and generates update notifications from published presence information, no end-to-end security between the originating source and the final receiver is available. To decide who is permitted to receive a specific presence document, the state agent is configured with explicit authorization policies controlling the information dissemination. Watchers then must authenticate themselves with the server to enable the determination of their specific authorization class.

The output generation of an aggregation server is governed by the installed authorization policy, ensuring the privacy of sensitive information. Security issues may arise from scripts that modify the original presence document, as sensitive data could be marked non-sensitive, copied into channels that are handled less restrictive, or the definition of authorization classes could be modified. Therefore, the configuration of the aggregation service regarding a particular presentity is permitted only for users that authenticate themselves with the credentials of the respective presentity.

8.3 Comparison With Other Approaches

To compare our system with existing approaches, we have to distinguish the aspects that should be compared. In Chapter 4, we have considered related work with respect to application scenarios, architectural concepts, and the technical implementation. The IETF's ongoing standardization effort for combination of presence documents is considered as orthogonal to our work, as we have carefully designed our presence service to fit into the common framework for instant messaging and presence defined by the IETF. Thus, our approach can be extended to integrate these new standards as they become available.

In the following, we compare our approach to two of the related approaches shown in Chapter 4, the *Solar* system (see Section 4.1.1) and the framework for *Context-aware Communication Services* (see Section 4.1.4). The comparison is based on the requirements we have discussed in Chapter 3 and highlights these approaches' specific application areas and design decisions. The two projects we have selected as a reference represent two essentially different ways of aggregating low-level sensor data.

The Solar system disseminates any data that has been published anywhere in the network as sets of key/value pairs which in turn may be combined to new events by any node of this content network. Applications subscribe to a specific event stream by placing filter expressions that match the key/value pairs they are interested in.

In the Solar system, the interpretation of events is up to the application using this system. Neither a fixed set of keys has been defined nor the types of publishing sources. In contrast, the *Context-aware Communication Services* provide a framework for automatic establishment of interactive communication channels based on the availability of participating users. The presence information transmitted by this system uses a fixed vocabulary with clear semantics. No indication of the raw sensor data is given after it has been aggregated to a more abstract context description. The application logic hence is more generic as it uses only the abstract context information for status inference, and does not need to know concrete sensor values this information is based upon.

After pointing out these fundamentally different views of presence information aggregation, we compare both approaches with our aggregation service to determine the degree of compliance to the initial requirements.

8.3.1 Solar

The Solar system described in Section 4.1.1 enables the creation of context-aware applications on top of a content-addressable network. Presence sources such as sensors or aggregating nodes publish data streams over an application-level multicast network that can be subscribed from any node in this overlay network.

Unlike our approach, the Solar system does not provide any scripting framework or object model that facilitates creation of custom aggregation functions. Instead, application writers have direct access to any sensor data stream that happens to be subscribed by the node that runs this particular application. As a consequence, there is no authorization framework that enforces protection from disclosure of sensitive data.

To a certain degree, the Solar approach is akin to the messaging infrastructure we use within the local presence environment. Here, self-descriptive entities may publish any data at any frequency. Solar nodes and Mbus entities even have a similar addressing scheme, except for the

fact that Solar allows for hierarchical nesting of attribute sets. The Solar system hence could have been an alternative to the local messaging framework. The Mbus, however, provides a more explicit command interface that allows for easy distinction between commands and data and thus is better suited for command-oriented applications.

8.3.2 Context-aware Communication Services

As seen in Section 4.1.4, the Context-aware Communication Services provide a framework for management and aggregation of context information, with a special focus on interpersonal communication. Similar to the approach discussed in this thesis, a tight integration with the SIP signaling infrastructure is envisioned, using an enhanced version of the Presence Information Data Format to convey context-related information with the presence document. This information includes an indication of a presence attribute's exactness and its degrading over time. In addition, the description of a media channel can be tagged with an authorization class that is used to protect sensitive data from being published to unauthorized persons.

While both approaches provide support for rule-based activation of contexts, they differ in their respective research and engineering focus: The Context-aware Communication Services framework addresses application integration to build context-aware applications that hide the complexity of the infrastructure from the user. Our approach, in contrast, fosters user participation by providing tools for aggregation of local presence information into coherent presence information sets that are published to subscribed watchers.

On the engineering side, both frameworks follow a different strategy. While the Context-aware Communication Services have defined a Web service for publication of aggregated context descriptions, our solution uses the existing SIP infrastructure for wide-area event publication, and a light-weight messaging infrastructure for local communication within a presence zone. This design especially gives the flexibility to deal with dynamically changing data sources in local presence environments where sensors are added or removed frequently, while keeping the advantages of SIP-based wide-area communication.

Finally, our presence aggregation service is entirely transparent to the SIP network used for interpersonal communication. No extensions to the Session Initiation Protocol are required to use our service. The aggregation process relies on standards-compliant enhancements to the Presence Information Data Format that is used to convey descriptions of media channels for interpersonal communication.

8.4 Open Issues and Next Steps

During our work on the presence aggregation service and its evaluation in several research projects, we have identified a number of open issues and possible extensions of our approach. In this section, we describe these issues and give some initial ideas for improving our system and adapting it for new application areas that we had not considered when starting our work.

8.4.1 Improving the User Interface

Aggregation rules in our approach take advantage of the Turing-complete ECMAScript programming language to avoid users being annoyed by any artificial restriction imposed by a new

specification language. Thus, users with high technical skills can control nearly every aspect of the aggregation engine that has no negative effect on the overall operation of the presence server. In particular, the only limits imposed on an aggregation specification are the code library provided by the aggregation server and the upper limit of the processor time granted to each aggregation specification.

We envision that users who are accustomed to creating their own JavaScript-based web pages will rapidly adapt the necessary knowledge for authoring their personal presence aggregation rules. For users with less technical skills, however, this approach will be too difficult, making the use of our system impossible. Although these users might benefit from a small number of skilled users who provide aggregation functions they have created and which only have to be customized to work for other users as well, other forms of user interfaces than script upload is appreciated.

Today, many users with moderate technical skills prefer graphical user interfaces even to perform system configuration tasks. As there is also a number of graph visualization tools, designers of specification languages typically consider graph-representations as a canonical form of written specification documents. Examples are the *Unified Modeling Language* (UML) or the *Call Processing Language* (CPL). For implementors, these languages have the advantage that the program flow constructed from a graph-like specification has the same characteristics as the graph itself. For example, recursive functions can be avoided by specifying that graphs have to be directed and non-cyclic.

Creating graphic user interfaces for graph-oriented specification languages is by far more simple than for free-form Turing-complete specifications. The latter typically requires certain program constructs and language features either to be let out of scope or to be offered in an "advanced mode" only. In 2001, Hüttner and Berger [Hüt01, Ber01] have demonstrated a prototype implementation of a graphical user interface for authoring transformation specifications that is capable of extension expressions in an arbitrary programming language. A future extension of our aggregation service might provide a similar interface specific to the ECMAScript-based specification language.

8.4.2 Generalized Document Transformations

Taking a close look on the common operations applied to input documents during the presence aggregation process, we came to the conclusion that presence aggregation is a specific kind of document transformation. As our system provides a fixed set of native objects and operations that are specific to these objects (cf. Chapter 6), the actual transformation step is covered by the aggregation engine and thus not visible to the user. As soon as presence documents are received that contain extension elements that have no corresponding native representation in our specification language, the modification of those objects is difficult, if possible at all.

An improved version of our specification language for aggregation rules should treat XML document transformation as a first-class operation by enabling manipulation of the documents' information set. The implementation options available for this task are the following:

- Definition of an XML *Document Object Model* (DOM) [DOM Level2] for (enhanced) PIDF documents. A DOM defines an interface for accessing components of an XML processor's internal representation of given input documents. The object model represents the conceptual components of an XML document such as elements, attributes,

namespaces, comments, and processing instructions. In addition, the hierarchical structure implied by the nesting of elements is retained in a tree-like representation. The core specification also provides bindings of the normative interface to the syntax of common programming languages such as Java and JavaScript. Therefore, an adaptation to ECMAScript would only require little effort.

In addition to the object model and specific access functions, a PIDF DOM could be integrated with the W3C eventing framework defined in [DOM Level2 Events]. Doing so, handler functions can be associated with external events on specific part of a PIDF document, such as partial updates or timer expiry.

- Another approach to a better support for document transformations during presence aggregations is to use a specification language with intrinsic support for manipulation of XML documents. For example, *ECMAScript for XML* (E4X) [ECMA-357] adds native XML datatypes to the ECMAScript language and provides additional operators for manipulating those XML objects. The transformation of PIDF documents then would be achieved by directly manipulating first-class objects in ECMAScript.

Note that a combination of both options does not seem to be a good idea as the E4X language provides its own internal representation of XML documents. A specialized document object model instead is a good starting point for advanced applications that make use of the presence-specific characteristics and operations the model provides. As a result, both implementation options are worth considering when extending our approach.

8.4.3 Location-Aware Resource Allocation for Conferencing Systems

The GEOCOOP project (see Section 2.4.2) has demonstrated that presence services are an integral part of enterprise conferencing systems to facilitate communication setup between members of a single organization. To be used efficiently, conferencing systems not only have to respect the presence status of invited participants but also must support resource allocation and dynamic re-negotiation of parameters. The attribute set of physical resources such as meeting rooms and presentation facilities can be augmented with geo-spatial coordinates to facilitate conference participation of individual users. Our presence aggregation service therefore provides the technical basis for implementing additional services such as the automated allocation of meeting rooms and presentation equipment based on location data and timed scheduling information.

In our opinion, automated scheduling of teleconferences yet suffers from the vast overhead that is required to set up and run the virtual conference between two or more sites of an organization. In addition to the technical experience, a conference organizer must be assigned who has to negotiate a time slot for the conference, allocate resources (accordingly equipped meeting rooms and audio/video equipment), and who must invite the designated participants. Whenever schedule changes occur or important members cannot participate, the setup process begins anew.

A solution to this problem is to use autonomous agents that track the current planning activities and re-negotiate conference parameters if necessary. User-specific preferences as well as descriptions of available devices' capabilities guarantee successful communication establishment once the conference is determined to start. Within the GEOCOOP project that made use of

our presence aggregation service, the concept of hybrid conferencing systems has been outlined, i.e. virtual conferences being combined with face-to-face meetings at one or more conference sites.

The presence service could be extended from its focus on interpersonal communication to a more general service that tracks the availability of arbitrary resources at a specific point in time. This generalized service then could be augmented with functions for dynamic resource allocation to support the initiation of teleconferences and face-to-face meetings.

8.4.4 Simulation of Aggregation Rules Using OMNet++

As multi-step aggregation scenarios can become very complex, new evaluation methods are required to show authors of aggregation rules the impact of their specification on the entire aggregation process. A better understanding of the system's behavior over time could be achieved with a simulation environment that represents all important components of the real system and visualizes the effect of changed input parameters such as status changes and modified aggregation rules.

During our work, we have evaluated OMNet++[48], an extensible network simulation engine with a graphical user interface to display simulation runs. As OMNet++ is written in C++, an integration of our modules into the test-bed would be possible without major change of the existing code base. In addition, the OMNet++ framework already provides a number of modules for simulating IP networks, including a stub mechanism for socket operations and DNS name resolution. Thus, the simulation could be performed at a very low abstraction level, i.e. on the basis of Mbus communication and SIP message flows between network nodes. This way, even new Mbus-enabled sensor modules could be tested before deployment.

A major development task to set up an OMNet++ simulation environment for our presence aggregation service therefore is the integration of our notifier-based socket library with the internal main loop of the OMNet++ system. This also includes the adjustment of internal timers to the scalable time axis if the simulation kernel. The first part already has been demonstrated for the integration of the Mbus library which is based on the same socket abstraction as our SIP protocol stack. The adaptation of timer ticks from the OMNet++ system requires some more changes in the Mbus implementation's internal notification loop, and therefore it is not finished at the time of writing.

8.4.5 Enhancing the System Performance

One of the lessons we have learned while implementing the presence aggregation system is that the server performance degrades badly with dynamic aggregation enabled. This observation is less obvious as it seems at the first glance, as the aggregation service is not tremendously different from the REST architecture of the World Wide Web (c.f. Section 3.3.1). However, aggregation servers need to store the presence status records of any presentity in their administrative domain, as incoming status update notifications must be related to the current presence status. In addition, the decay function provided in enhanced PIDF documents may trigger the re-calculation of presence information and thus must be processed in regular intervals as well.

[48] Short for *Objective Modular Network Test-bed in C++*, see <http://www.omnetpp.org>.

To avoid server overload, the user base could be distributed over a farm of servers that provides single ingress and egress points to appear as a single server with a unique IP address. Technically, this can be achieved at the application layer by providing a load balancer for even distribution of incoming SIP messages to a sufficient number of back-end servers. As most SIP servers still favor UDP as protocol for message transport, a simple stateless SIP proxy with an efficient distribution algorithm can be used for this task. The back-end servers then only need to process the subscriptions and change notifications for a small part of the user database, and thus have more resources available for evaluating aggregation rules.

8.4.6 Security

In Section 3.4, we have stated that presence services always face the trade-off between privacy of information and the increase of service quality with more detailed information being published. The design of our presence service therefore offers a powerful interface for users to stay in control of the information that is published on their behalf. This approach assumes that users are aware of what they are doing, i.e. authors of aggregation rules have high technical skills and—at least to a certain degree—programming experience. The complexity of multi-step aggregation, however, makes rule development a tedious, error-prone task. Given this, our approach holds the intrinsic risk of accidentally creating aggregation rules that leak information to unauthorized recipients.

We envision that Internet-scale real-time communication in general and presence services in particular in the near future will become a research area of high interest for security experts. The IETF policy framework discussed in Section 4.3.2 is a first step towards a more reliable security infrastructure for SIP-based presence. Many other issues such as reliable identification of communication peers, SPIT prevention, and prevention of presence profile creation come in mind. The issues regarding security and user acceptance we have discussed in this thesis may be a first guidance for further research.

8.5 Concluding Remarks

The main focus of this thesis has been on the improvement of interpersonal communication by providing information on the availability and willingness of users to engage in real-time multimedia conversation. In particular, we have extended an existing presence service to provide user-controlled aggregation of presence information from multiple sources. The conceptual work included the definition of a new and concise semantics for the existing IETF presence profile. In our analysis, we have stated the tight relationship of the IETF work with previous research in the area of computer-supported cooperative work and groupware systems. Possible application areas of our results also include ambient computing and ubiquitous computing (or pervasive computing, respectively).

An interesting aspect of our system architecture is the ability to extend local presence environments—i.e., the origins of basic presence information—with additional entities. New sensors could be added as they appear on the local Mbus, even without manual configuration if the role of this sensor was known to the installed aggregation specification. Once integrated in our implementation, the Mbus turned out to be useful for many other tasks as well. Its asynchronous messaging interface combined with several generic communication patterns providing

distinct levels of control leveraged the component-based design of our system and its modular implementation. Our SIP protocol implementation thus was used in many other projects in different application areas. Combined with a call control engine, it even served as a gateway (a *back-to-back user agent*, B2BUA, in SIP terminology) between IPv6- and IPv4-based SIP networks in the European research project 6WINIT.

Given this, the local presence environment with its scriptable SIP server is useful in itself, even without the multi-step aggregation we have proposed in this thesis. As a standalone application, this SIP server could generate outgoing requests, react on incoming requests, and send or process responses within SIP transactions. The advanced features of our presence aggregation service would be triggered only if the PIDF documents generated by this server were processed by a server with a user-provided aggregation specification in place for this particular presentity. The same holds for enhanced PIDF specifications sent by a local presence environment: If the recipient has no support for this extension, it will be processed like any other PIDF document compliant to [RFC3863].

We have chosen this design, having in mind the large number of presence applications with incompatible features. For many years, there has been a lack of concise standards for the integration of presence services with the Internet architecture defined by existing IETF standards. In our opinion, new features that change the fundamental way a presence service works, must be backwards-compatible without any negative effect on the basic service that was available without this feature. In particular, no performance decrease or lack of basic functions should be observed by a "feature-aware" presence engine when processing presence documents originated by clients that do not support this specific feature.

The major drawbacks we have identified during our work are the degrading performance of scriptable aggregation engines compared to presence servers with static aggregation rules, and the specification language's missing support of operations for direct manipulation of PIDF documents. While scalability of presence servers can be achieved to some degree using load balancing techniques we have discussed in this chapter, the definition of a presence DOM together with related manipulation functions could be a reasonable solution to the transformation issue. In summary, we believe that our approach to multi-step presence aggregation discussed in this thesis is a first step to more intelligent presence services we will see in multi-hop computing environments as motivated in the NSF consensus report on *Research Direction for Developing an Infrastructure for Mobile & Wireless Systems* [KMM02]. The results discussed in this thesis therefore could be taken as a basis for advancing the work on SIP-based presence aggregation and distribution as well as multi-hop computing in general.

Appendix A

Example RPID Document

The IETF has defined a set of extensions to the basic Presence Information Data Format to describe the presence status of persons, devices and communication services. This RPID extensions are standardized as RFC 4480, using the core data model of RFC 4479.

The following example shows a PIDF document with RPID extensions and additional capability descriptions that can be used to characterize communication endpoints. [Prescaps]

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
          xmlns:dm="urn:ietf:params:xml:ns:pidf:data-model"
          xmlns:rpid="urn:ietf:params:xml:ns:pidf:rpid"
          xmlns:caps="urn:ietf:params:xml:ns:pidf:caps"
          entity="pres:alice@example.net">

  <tuple id="x12">
    <status>
      <basic>closed</basic>
    </status>
    <dm:deviceID>mac:080046c95177</dm:deviceID>
    <contact priority="0.8">sip:134.102.218.61:5060;user=ip</contact>
  </tuple>

  <tuple id="t18">
    <status>
      <basic>open</basic>
    </status>
    <dm:deviceID>urn:uuid:abcdef00-1234-5678-beef-89234abe3f8a</dm:deviceID>
    <contact priority="0.8">sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE</contact>
    <caps:servcaps>
      <caps:message>true</caps:message>
      <caps:audio>false</caps:audio>
      <caps:methods>
        <caps:supported> <caps:MESSAGE/> </caps:supported>
      </caps:methods>
    </caps:servcaps>
    <timestamp>2006-10-01T17:13:12+02:00</timestamp>
  </tuple>

  <tuple id="v36">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.1">sip:alice@voicemail.example.net</contact>
    <caps:servcaps>
      <caps:audio>true</caps:audio>
      <caps:automata>true</caps:automata>
      <caps:methods>
        <caps:supported>
          <caps:ACK/> <caps:BYE/> <caps:CANCEL/> <caps:INVITE/> <caps:PRACK/>
```

```
        </caps:supported>
      </caps:methods>
      <caps:extensions> <caps:rel100/> </caps:extensions>
    </caps:servcaps>
    <dm:deviceID>mac:0013ce6bb427</dm:deviceID>
    <timestamp>2006-10-01T16:56:00+02:00</timestamp>
  </tuple>

  <dm:person id="me">
    <rpid:activities from="2006-09-15T17:00:00+02:00"
                     until="2006-10-04T08:00:00+02:00">
      <rpid:note xml:lang="de">Auf Dienstreise bis zum 4. Oktober.</rpid:note>
      <rpid:travel/>
    </rpid:activities>
  </dm:person>

  <dm:device id="phone">
    <dm:deviceID>mac:080046c95177</dm:deviceID>
    <rpid:sphere> <rpid:work/> </rpid:sphere>
  </dm:device>

  <dm:device id="imclient">
    <rpid:user-input last-input="2006-10-01T17:13:07+02:00">idle</rpid:user-input>
    <dm:deviceID>urn:uuid:abcdef00-1234-5678-beef-89234abe3f8a</dm:deviceID>
    <rpid:sphere>
      <rpid:note xml:lang="en">XY company SIP-to-Jabber gateway</rpid:note>
      <rpid:home/>
    </rpid:sphere>
    <rpid:relationship> <rpid:other/> </rpid:relationship>
  </dm:device>

  <dm:device id="vmbox">
    <dm:deviceID>mac:0013ce6bb427</dm:deviceID>
    <rpid:sphere> <rpid:work/> </rpid:sphere>
  </dm:device>

</presence>
```

Appendix B

An Mbus Command Set for the Local Presence Environment

The following document specifies the command set to be used by sensors within a user's local presence environment. This document has been initially published in [OKB+04]. Its original page-oriented Internet-Draft format has been adapted to the pagination of this thesis.

Network Working Group — Buesching
Internet-Draft — Kutscher
Expires: August 1, 2004 — Ott
Uni Bremen TZI
February 2004

An Architecture for the Local Distribution of Presence Information using Mbus

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 1, 2004.

Abstract

This memo specifies an architecture for the local exchange of presence information. The specification includes an authentication mechanism based on DDA [3], a communication profile based on Mbus [1], an addressing scheme for distinguishing presence entities based

their roles and types within a distributed presence application.

Version
$Revision: 1.3 $

Table of Contents

1. Introduction 3
2. Architecture 5
3. Locating a Channel Provider 7
4. Authenticate with Channels 8
5. Mbus Presence Profile 9
5.1 Addressing Scheme 9
5.2 Channel Description 9
5.3 Distributing Presence Information 10
References 12
Authors' Addresses 12
A. Generic Attributes 13
B. Change History 14
Intellectual Property and Copyright Statements 15

1. Introduction

Different devices in a user's environment (e.g., in an office or desk area environment) can assess cues about the user's presence and availability status automatically. These devices can transmit this information to a presence agent that compiles, evaluates and aggregates the information on behalf of the user to make it available to communication peers. The main motivation for generating presence information this way is to automate the process of determining and configuring presence state for a user.

Different types of devices can be deployed that provide information of different quality:

- The user may explicitly set her presence and availability state using dedicated tools.
- A device that is associated to a specific user, e.g., a personal telephone, can provide valuable and precise information about a user's current activities and her availability, especially with respect to her disposition to engage in new communication sessions.
- The user may deploy tools that are able to determine their current position with respect to certain coordinate systems, e.g., GPS receivers or WLAN-positioning-enabled NICs. If these devices are known to be carried by a user, they can provide valuable information about her current location.
- The user may also deploy dedicated presence tools that can generate very specific cues, e.g., a motion sensor that detects that somebody is present, or an intelligent chair that can tell whether somebody is sitting on it or not. The information that is generated by this kind of device may be rather qualified as vague hints about a person's presence that precise information but nevertheless be useful as cues that are considered for compiling the overall presence state.

A user's presence application that receives and processes this information has to select and to evaluate the presented cues based on the type of device that generated them, based on application specific rules and based on user preferences. The device may also provide additional information to facilitate the weighting of information, e.g., they might add "quality-attributes" that can describe the authenticity of the provided information.

The communication session between the user's presence application and the devices providing presence information should be established

dynamically. The devices should be able to locate the presence application automatically or vice versa and negotiate the session parameters during the connection process. Moreover, devices have to be authenticated in order to ensure that only legitimate devices provide information the presence application.

In this memo, we provide a specification of how such a distributed presence application can be realized using the DDA (dynamic device association) protocol and Message Bus, a message-oriented protocol for local coordination. The specification defines an mechanism for the session establishment, an addressing scheme for distinguishing Mbus entities based on their roles and additionally an Mbus profile is described.

2. Architecture

For creating a meaningful presence profile of a person, a set of different sources, providing any kind of information regarding the presence of its owner, is required. Sources (in the following also called channels) that are usable in such a scenario, can be of very different types.

For example, a phone in a conference room can provide the information that one of participants of the meeting is 'in call'. A video camera as an additional presence source can provide presence information that allows to uniquely determine the person. This example demonstrates that a collection of sensors is capable of providing a more detailed and useful description of a presence state. Other, more simpler channels, which may just provide a boolean value like 'the owner uses this chair' or 'the light in this office room is switched on', could be useful as part of a collection of sources, but do not give enough data to generate a meaningful presence profile. The chair could be used by someone else and the room cleaner may have forgotten to switch off the lights. In conjunction with a video camera or a telephone the information provided by the more simpler channels could be enhanced.

The previous examples have shown that one single channel is typically not sufficient for creating a useful presence profile for a person. By combining the data of a set of multiple channels more usable presence profiles can be generated. For collecting and evaluating the data for a user's profile another component, a presence aggregation module (PAM), is required. Before such an aggregation module can access the presence information the following steps need to be performed:

Locating a channel provider

Each channel announces its service via IP multicast as specified in DDA. The PAM listens to this well-known multicast group, to be able to locate the channels. Section 3 provides a more detailed description.

Authenticating a PAM

When a PAM has located a channel that should be integrated into the presence profile, the channel provider has to authenticate the PAM in order to authorize access to the channel's presence information. For example, if a phone of user x is providing presence information it should only be possible for a restricted group of people to access this information. A detailed description of this procedure is provided in Section 4

Requesting a description from the channel

In order to be able to evaluate the information distributed by a channel the PAM requires a detailed description of the type of information. This channel description is illustrated in detail in Section 5.

3. Locating a Channel Provider

In order to establish a communication session between the PAM and a set of channel providers several steps have to be performed. The first step is to locate the channel providers. This is done by listening for DDA announcements that are sent by the channel providers to a well-known IP multicast group. These announcements describe the provided service (presence channel) and access protocol parameters for connecting to the service.

Announcing the channel's service DDA can support the presence profile generation process in two ways:

- A user's presence aggregation module that is pre-configured to use all available channels for generating the presence profile can dynamically connect to new appearing channels and use their information. This allows for a "plug-and-play operation", where explicit configuration of PAM instances is not required in order to support new channel providers.

- When adding a user interface to the PAM, its owner can dynamically change the set of channels that are used to generate her presence profile.

4. Authenticate with Channels

During this connection procedure the PAM needs to authenticate via DDA authentication procedures. This ensures that the channels are only used by their owner and/or a group of authorized people. For example a telephone in a conference room may be used for presence information by all participants of the meeting and not just by one of them. If a channel has accepted the authentication data sent by the PAM, it returns a configuration description via Mbus. For privacy the connection procedure and the session communication should be encrypted.

5. Mbus Presence Profile

The following Mbus profile defines the channel configuration and Mbus commands used by the channel providers and the PAM.

The two types of entities in an Mbus presence session, MPUAs (Mbus presence user agents) that represent the channel providers and the MPA (Mbus presence agent) that is integrated with a PAM, MUST adopt the following addressing scheme to be identifiable on the Mbus.

5.1 Addressing Scheme

All Mbus presence entities MUST use the address element key "mpres" in their Mbus address. MPUAs MUST use the value "mpua" and MPAs MUST use the value "mpa" for this address element. Additionally, an MPUA MAY use the address element key "app" (application) to identify the type of device it is part of. The usage of other address elements is not precluded by this specification.

An example for an Mbus address of an MPUA/channel:

(app:ip-phone id:4711-0@192.168.1.1 mpres:mpua)

An example for an Mbus address of an MPA/PAM:

(app:pam id:4711-1@192.168.1.2 mpres:mpa)

5.2 Channel Description

Each MPUA MUST be able to provide a description of itself, when requested by an MPA. The description consists of a set of attributes as specified in the PIDF-XY document [4].

After completing the DDA connection procedure, an MPUA joins an Mbus session with the MPA and possibly other MPUAs. An MPUA MUST be able to accept other Mbus entities in the session as long as these entities follow the addressing scheme. When the MPUA is detected by the MPA in the Mbus session it requests a channel description from the new entity with the Mbus RPC command "mpres.mpua.describe". The answer to this request MUST contain the Mbus representation of a channel description.

The Mbus representation for the description consists of two lists containing the generic and the channel specific attributes:

o The generic attributes describe the basic configuration of a channel. Each attribute has a name, a type and its value that are all represented as strings. A complete list of the generic attributes and its respective type can be found in Appendix A.

(<name> <type> <value>)
Example: A generic attribute of a channel is the "interval" determining the period of time in which the channel repeats its status reports. A complete description:

("interval" "number" "30")

o The channel specific attributes describe the presence state information that can be provided by a channel. Each attribute provides a name, a type and a value. Additionally they provide a decay function and a time stamp. All these elements are represented as strings.

```
( <name> <type> <value> <decay> <timestamp>
)
```

Example:

```
( "INUSE" "bool" "no" "const()" "200404151046" )
```

A complete reply to an Mbus RPC request "mpres.mpua.describe" may look like the following example:

```
mbus/1.0 4 1085659424 U
(lang:python type:sensor name:chairsensor id:21873-0@134.102.218.127)
(lang:python org:tzi os:linux id:21880-0@134.102.218.127 ) ()
mpres.mpua.describe.return
((("ID" "3" )("RPC-STATUS" "OK"))
 ((OK OK "success")
  (((name string "chair-sensor")
    (type list(string) ("sensor" "switch" "chair"))
    (interval number 60)
    (ownership number 1.000000)
    (authclass string "protected"))
   ((inuse boolean "yes" "const()" 20040527140335)))))
```

5.3 Distributing Presence Information

An MPUA, which has just joined an Mbus session does not start reporting its presence state immediately. Instead it waits for an MPA requesting its channel description. After returning its description to the MPA the MPUA begins to report its presence state using the Mbus command "mpres.mpua.report". This report is repeated in an interval defined by the generic attribute "interval" (in seconds). An exception of this rule occurs if one or more presence state variables change before the interval is expired. In this case the new state is reported immediately and the timer is restarted.

```
( mpres.mpua.report <dynamic-attributes> )
```

References

[1] Ott, Perkins and Kutscher, "A Message Bus for Local Coordination", RFC 3259, April 2002.

[2] Kutscher, D., "The Message Bus: Guidelines for Application Profile Writers", Internet Draft draft-ietf-mmusic-mbus-guidelines-00.txt, February 2001.

[3] Kutscher, D., Ott and A. Buesching, "Dynamic Device Association (DDA)", February 2003.

[4] Kutscher, D., Bergmann, O., Buesching, A. and Ott, "PIDF-XY an extension to PIDF", February 2004.

Authors' Addresses

Andreas Buesching
Uni Bremen TZI

Dirk Kutscher
Uni Bremen TZI

Joerg Ott
Uni Bremen TZI

Appendix A. Generic Attributes

type list(string)
The list of strings should describe the type of the channel device in a generic way, e.g.\ a mobile phone is described by a list like ("COMMUNICATION" "INTERACTIVE" "PHONE" "MOBILE") and an e-mail channel could be described by ("COMMUNICATION" "ASYNCHRONOUS" "EMAIL").

name string
The name of a channel should a human readable description of the channel.

ownership number
The ownership of a channel is represented as a number in the range from zero to one. This number defines a percentage rate to which a channel is owned by a person. For example a phone in a conference room belongs to all participants of the meeting while a private mobile phone belongs to just one single person.

interval number
This number defines the interval of repetition (in seconds) for state reports.

authclass string
This attribute represents the name of a valid authorization class.

Appendix B. Change History

- Renamed Mbus commands
  mpres.server.* -> mpres.mpua.*
  mpres.client.* -> mpres.mpua.*
  All commands belong to the MPUA.

- Adjusted description of RPC "mpres.server.describe" return value to match current implementation. The type of a channel is a generic attribute does not need to be outside of the list of these attributes.

- Added appendix with complete list of generic attributes

- Merged mbus-presence and mpres

- Removed address element mpua-type.
- Added description of the Mbus commands mpres.client.report and mpres.server.change.
- Described architecture for local distribution of presence information.
- Added definition for address element "app" for MPUAs
- Added references to other documents.

Intellectual Property Statement

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.

Appendix C

Example for an Enhanced PIDF Document

The following example shows a PIDF document with extension elements that specify specific authorization classes as well as meta information for PIDF presence tuples. This document describes three media channels each of which has a specific type, authorization class, and presence status.

```
<impp:presence entity="pres:alice@example.net"
               xmlns:impp="urn:ietf:params:xml:ns:pidf"
               xmlns:pidfxy="urn:ietf:params:xml:ns:pidfxy">

  <!-- authorization definition -->

  <pidfxy:authdef>
    <pidfxy:authclass name="private">
      <pidfxy:authelem>alice@example\.net</pidfxy:authelem>
    </pidfxy:authclass>

    <pidfxy:authclass name="friends">
      <pidfxy:authelem>Bob@otherdomain\.com</pidfxy:authelem>
      <pidfxy:authelem>[^@]*@mysportsclub\.com</pidfxy:authelem>
      <pidfxy:authelem>jill@example\.net</pidfxy:authelem>
    </pidfxy:authclass>

    <pidfxy:authclass name="family">
      <pidfxy:authref name="private" />
      <pidfxy:authelem>philip@example\.net</pidfxy:authelem>
      <pidfxy:authelem>edward@example\.net</pidfxy:authelem>
      <pidfxy:authelem>anne@example\.net</pidfxy:authelem>
    </pidfxy:authclass>

    <pidfxy:authclass name="protected">
      <pidfxy:authref name="friends" />
      <pidfxy:authref name="family" />
    </pidfxy:authclass>

    <pidfxy:authclass name="public">
      <pidfxy:set-difference>
        <pidfxy:authref name="protected" />
        <pidfxy:authelem>.*</pidfxy:authelem>
      </pidfxy:set-difference>
    </pidfxy:authclass>
  </pidfxy:authdef>

  <!-- media channel: shared office phone  -->

  <impp:tuple id="x12">
    <impp:status>
      <impp:basic>CLOSED</impp:basic>
      <pidfxy:name type="string">office</pidfxy:name>
```

```
      <pidfxy:type type="tokenlist">COMMUNICATION, INTERACTIVE, AUDIO</pidfxy:type>
      <pidfxy:authclass type="string">public</pidfxy:authclass>
      <pidfxy:ownership type="number">0.5</pidfxy:ownership>
    </impp:status>
    <impp:contact priority="0.8">sip:134.102.218.61:5060;user=ip</impp:contact>
    <impp:note xml:lang="de">Bis zum 4. Oktober ist mein Büro nicht besetzt.</impp:note>
  </impp:tuple>

  <!-- media channel: IM client, user is currently typing -->

  <impp:tuple id="t18">
    <impp:status>
      <impp:basic>OPEN</impp:basic>
      <pidfxy:name type="string">jabber</pidfxy:name>
      <pidfxy:type type="tokenlist">COMMUNICATION, INTERACTIVE, TEXT</pidfxy:type>
      <pidfxy:authclass type="string">protected</pidfxy:authclass>
      <pidfxy:ownership type="number">1.0</pidfxy:ownership>
      <pidfxy:interval type="number">35</pidfxy:interval>
      <pidfxy:typing type="boolean" timestamp="2006-10-01T17:13:06+02:00"
          decay="linear(500)" threshold="0.4">true</pidfxy:typing>
    </impp:status>
    <impp:contact priority="0.8">
       sip:alice!jabber.org@jabbergw.example.net;method=MESSAGE
    </impp:contact>
    <impp:note xml:lang="en">
       You can contact me via our company's SIP-to-Jabber gateway.
    </impp:note>
    <impp:timestamp>2006-10-01T17:13:12+02:00</impp:timestamp>
  </impp:tuple>

  <!-- media channel: voicemail box -->

  <impp:tuple id="v36">
    <impp:status>
      <impp:basic>OPEN</impp:basic>
      <pidfxy:name type="string">voicemail</pidfxy:name>
      <pidfxy:type type="tokenlist">COMMUNICATION,AUDIO</pidfxy:type>
      <pidfxy:authclass type="string">public</pidfxy:authclass>
      <pidfxy:ownership type="number">1.0</pidfxy:ownership>
    </impp:status>
    <impp:contact priority="0.1">sip:alice@voicemail.example.net</impp:contact>
    <impp:note xml:lang="en">
      If everything else fails, leave me a note on my personal voice mailbox
    </impp:note>
    <impp:timestamp>2006-10-01T16:56:00+02:00</impp:timestamp>
  </impp:tuple>

</impp:presence>
```

Appendix D

PAL Function Library

The Presence Aggregation Language (PAL) comes with a number of pre-defined functions. This appendix gives a complete list of these functions together with a description of their semantics and simple usage examples.

getChannel

```
Channel getChannel(Id)
```

Returns the channel description with a given name. When called, this function first searches the input set, then the current status of the presentity. If no channel was found with this name, the `Null` value is returned. Otherwise

The function is typically used for quick access to a named channel description as seen in the following example:

```
var office = getChannel("office")          /* presence status from office environment */

if (office && office.status == "open") {
  /* ... create presence status ... */
}
```

setTimeout

```
Object setTimeout(Function, Number)
```

Registers the given `Function` object as callback function that is called after `Number` milliseconds. The function returns a handle to the created timer that can be passed to `clearTimeout` to remove the timer.

The following example is used to register a timer callback function that is called after 10 seconds.

```
var timer = setTimeout(function() {
  if (office.status == "closed") OUT.status.value = "away";
}, 10000 ) );
```

clearTimeout

```
Boolean clearTimeout(Object)
```

Removes the timer with the given handle. The function returns true if the timer has been removed, false if it does not exist. The PAL interpreter MUST always assure that the timer is not invoked after it has been removed, unless it is added again.

The following example removes a timer that has previously been added with `setTimeout`.

```
clearTimeout(timer);
```

setTrigger

```
Object setTrigger(Trigger)
```

Adds the specified Trigger object to the system's list of event sources. The function returns always the unique identifier of the trigger object.

The active triggers for the script are listed in the set `TRIGGERS`. When adding a Trigger object that has an identifier equal to any identifier of the triggers that are already registered, the old trigger will be replaced without activation.

The following example registers a new trigger object that fires when the attribute's exactness goes below 20 %.

```
var trigger = setTrigger(new Trigger(callbackFunction, attribute, 0.2 ) );
```

clearTrigger

```
Boolean clearTrigger(Object)
```

Description:

Removes the Trigger object with the given identifier from the system's trigger list. The function returns true if the trigger has been removed, false if it does not exist. The PAL interpreter must always assure that the trigger is not invoked after it has been removed, unless it is added again.

A trigger that has previously been registered with a call to the function `setTrigger` can be removed as follows:

```
clearTrigger(trigger);
```

authdef

```
Aset authdef(Presentity)
```

Used to access the specified presentity's authorization class definition. The function returns an associative set containing a parsed version of the authorization definition as specified for the given presentity. The keys used in this associative set are the names of the authorization classes. The values are `Set` objects containing the authorization definitions for the particular class as regular expression objects.

The following example shows how this function can be used to check for a specific URI in the authorization definition. If the value is not contained, the output definition will be adjusted accordingly.

```
var auth = authdef(IN[0])
    class = typeof auth == 'undefined' ? false : auth.get("friends");

if (!class || !class.foldl(function(obj,res) {
                             return res || obj.exec("Bob@otherdomain.com");
                           }, false))
{
  OUT[0].channel[0].status.value = "away";
}
```

Appendix E

XML Transformation Specification for Presence Documents

The presence aggregation service discussed in this thesis supports the widely deployed presence format of the Microsoft Windows Messenger that is used in various other applications as well. As this format is nearly feature compatible to the standardized PIDF format [RFC3863], we have provided the following two XML transformation specifications to convert between both formats.

E.1 Transformation from XPIDF to PIDF

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="urn:ietf:params:xml:ns:pidf"
        >

<xsl:output method="xml"
            indent="yes"
            encoding="UTF-8"
            media-type="application/pidf+xml"
            />

  <xsl:template match="presence">
    <presence entity="{presentity/@uri}">
      <xsl:apply-templates />
    </presence>
  </xsl:template>

  <xsl:template match="atom">
    <tuple>
      <xsl:copy-of select="./@*" />
      <status>
        <basic><xsl:value-of select=".//@status" /></basic>
        <xsl:copy-of select=".//msnsubstatus" />
      </status>
      <xsl:apply-templates select="address" />
    </tuple>
  </xsl:template>

  <xsl:template match="address">
    <contact>
      <xsl:apply-templates select="@priority" />
      <xsl:value-of select="@uri" />
    </contact>
```

```
  </xsl:template>

  <xsl:template match="address/@priority">
    <xsl:attribute name="priority">
      <xsl:value-of select="format-number(translate(.,',','.'),'0.000')" />
    </xsl:attribute>
  </xsl:template>

</xsl:stylesheet>
```

E.2 Transformation from PIDF to XPIDF

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:pidf="urn:ietf:params:xml:ns:pidf"
                xmlns:pidfxy="urn:ietf:params:xml:ns:pidfxy"
                xmlns=""
                exclude-namespace-prefix="pidf pidfxy #DEFAULT"
        >

<xsl:output method="xml"
            indent="yes"
            encoding="UTF-8"
            doctype-public="-//IETF//DTD RFCxxxx XPIDF 1.0//EN"
            doctype-system="xpidf.dtd"
            media-type="application/xpidf+xml"
            />

  <xsl:template match="pidfxy:*" />

  <xsl:template match="pidf:presence">
    <presence>
      <presentity uri="{@entity}"/>
      <xsl:apply-templates />
    </presence>
  </xsl:template>

  <xsl:template match="pidf:tuple">
    <atom>
      <xsl:copy-of select="./@*" />
      <address>
          <xsl:apply-templates select="@priority" />
          <xsl:attribute name="uri" select="."/>
        <xsl:apply-templates select="*" />
      </address>
    </atom>
  </xsl:template>

  <xsl:template match="pidf:contact" />

  <xsl:template match="pidf:basic">
    <status status="{.}"/>
  </xsl:template>

  <xsl:template match="msnsubstatus">
    <xsl:copy-of select="."/>
  </xsl:template>

  <xsl:template match="pidf:contact/@priority">
    <xsl:attribute name="priority">
      <xsl:value-of select="format-number(translate(.,',','.'),'0.000')" />
    </xsl:attribute>
  </xsl:template>

</xsl:stylesheet>
```

Bibliography

[AAS03] A.-S. Axelsson, Å. Abelin, and R. Schroeder: *Communication in Virtual Environments: Establishing Common Ground for a Collaborative Spatial Task*, Presented at PRESENCE 2003, 6th Annual International Workshop on Presence, 2003.

[ASB+99] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley: *The Design and Implementation of an Intentional Naming System*, Operating Systems Review, Prooceedings of the 17th ACM Symposium on Operating System Principles (SOSP '99), December 1999, 186—201.

[Alv02] Harald Alvestrand: *Instant Messaging and Presence on the Internet*, Internet Society, November 2002.

[Auth] J. Rosenberg: *Presence Authorization Rules*, Internet-Draft draft-ietf-simple-presence-rules-07, work in progress, June 2006.

[BAD04] N. Banerjee, A. Acharya, and S. K. Das: *Peer-to-peer SIP-Based Services over Wireless Ad Hoc Networks*, Proceedings of the Workshop on Broadband Wireless Multimedia at the 1st Annual International Conference on Broadband Networks (Broadnets 2004).

[BBM+72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson: *TENEX, a paged time sharing system for the PDP-10*, Communications of the ACM, March 1972, 135—143.

[BCB03] Rob von Behren, Jeremy Condit, and Eric Brewer: *Why Events are a Bad Idea (for high-concurrency servers)*, Presented at HotOS, 2003.

[BCL+94] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret: *The World Wide Web*, Communications of the ACM, August 1994, 76—82.

[BCM+99] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Storm, and Daniel C. Sturman: *An efficient multicast protocol for content-based publish-subscribe systems*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.

[BCT+96] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Aleyander E. Wise: *A framework for event-based software integration*, ACM Transactions on Software Engineering and Methodology (TOSEM), 1996, 378—421.

[BEG00] Michael Boyle, Christopher Edwards, and Saul Greenberg: *The Effects of Filtered Video on Awareness and Privacy*, Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work, December 2000, ACM Press, Philadelphia, PA, USA, 1—10.

[BOK+02] C. Bormann, J. Ott, D. Kutscher, and O. Bergmann: *Konzepte Content-Repräsentation & Markup-Sprachen*, SPC TEIA Lehrbuch Verlag, 2002.

[BOK05] Olaf Bergmann, Jörg Ott, and Dirk Kutscher: *A Script-based Approach to Distributed Presence Aggregation*, Proceedings of the IEEE WirelessCom 2005 Conference, June 2005, Maui, HI.

[BTS+02] James "Bo" Begole, John C. Tang, Randall B. Smith, and Nicole Yankelovich: *Work rhythms: analyzing visualizations of awareness histories of distributed groups*, Proceedings of the 2002 ACM conference on Computer supported cooperative work, 2002, 334—343.

[BaSc04] Salman A. Baset and Henning Schulzrinne: *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol*, Department of Computer Science, Columbia University, 2004.

[BeFa93] Steve Benford and Lennart Fahlén: *A Spatial Model of Interaction in Large Virtual Environments*, Proceedings of the Third European Conference on Computer Supported Cooperative Work, 1993, Kluwer, 109—124.

[BeGr97] Steve Benford and Chris Greenhalgh: *Introducing Third Party Objects into the Spatial Model of Interaction*, Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 1997, Kluwer, 189—204.

[BeSe93] Victoria Bellotti and Abigail Sellen: *Design for Privacy in Ubiquitous Computing Environments*, Proceedings of the Third European Conference on Computer Supported Cooperative Work, 1993, Kluwer, 77—92.

[Ber01] Thomas Berger: *Business-to-Business-Kommunikation mit strukturierten Dokumenten: Ein grafisches Tool für die interaktive Erstellung strukturorientierter Abfrageausdrücke zur Transformation von XML-Dokumenten*, Universität Bremen, 2001, Master's Thesis.

[Ber89] Tim Berners-Lee: *Information Management: A Proposal*, Organisation Européenne pour la Recherche Nucléare (CERN), 1989.

[Ber98] Tim Berners-Lee: *Web Architecture from 50,000 feet*, 1998.

[Bha02] Yudhijit Bhattacharjee: *A Swarm of Little Notes*, Time Magazine, September 2002.

[BlCl01] Marjory S. Blumenthal and David D. Clark: *Rethinking the design of the Internet: the end-to-end arguments vs. the brave new world*, ACM Transactions on Internet Technology (TOIT), August 2001, 70—109.

[BoTh88] Nathaniel S. Borenstein and Chris A. Thyberg: *Cooperative Work in the Andrew Message System*, Proceedings of the ACM 1988 Conference on Computer Supported Cooperative Work, September 1988, 306—323.

[Bry03] David A. Bryan: *SOSIMPLE — Self Organizing Simple. A Proposed P2P Instant Messaging System*, College of William and Mary, 2003.

[Bus45] Vannevar Bush: *As We May Think*, Atlantic Monthly, July 1945, 101—108.

[Bux95a] W. Buxton: *Integrating the Periphery and Context: A New Model of Telematics*, Proceedings of Graphics Interface '95, 1995, 239—246.

[Bux95b] W. Buxton: *Ubiquitous Media and the Active Office*, 1995.

[CCH00a] M. Czerwinski, E. Cutrell, and E. Horvitz: *Instant Messaging and Interruption: Influence of Task Type on Performance*, Proceedings of OZCHI 2000, 2000, 356—361.

[CCH00b] Mary Czerwinski, Edward Cutrell, and Eric Horvitz: *Instant Messaging: Effects of Relevance and Timing*, Proceedings of HCI 2000, September 2000, 71—76.

[CCH01] E. Cutrell, M. Czerwinski, and E. Horvitz: *Notification, disruption, and memory: Effects of messaging interruptions on memory and performance*, Proceedings of Interact 2001. IFIP. Conference on Human-Computer Interaction, 2001.

[CCS+03a] D. Scott McCrickard, C. M. Chewar, Jacob P. Somervell, and Ali Ndiwalana: *A model for notification systems evaluation: assessing user goals for multitasking activity*, ACM Transactions on Computer-Human Interaction (TOCHI), 2003, 312—338.

[CCW02] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf: *An Experience in Evaluating Publish/Subscribe Services in a Wireless Network*, ACM WOSP 02, 2002, 128—133.

[CCW03] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf: *Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications*, IEEE Transactions on Software Engineering, December 2003, 1059—1071.

[CDG+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach: *Secure routing for structured peer-to-peer overlay networks*, Proceedings OSDI, December 2002.

[CJT01] L. F. Cabrera, M. B. Jones, and M. Theimer: *Herald: achieving a global event notification service*, Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, 2001, 87—92.

[CLR95] *Ontario Telepresence Project*, John Chattoe, Peter Leach, Ron Riesenbach, Information Technology Research Centre, Telecommunications Research Institute of Ontario, March 1995.

[CPT+01] N. H. Cohen, A. Purakayastha, J. Turek, L. Wong, and D. Yeh: *Challenges in flexible aggregation of pervasive data*, IBM Research, January 2001.

[CRW03a] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf: *A Routing Scheme for Content-Based Networking*, Software Engineering Research Laboratory, University of Colorado, June 2003.

[CVJ+02] J. J. Cadiz, Gina Venolia, Gavin Jancke, and Anoop Gupta: *Designing and Deploying an Information Awareness Interface*, Proceedings of the CSCW 2002, 2002, 314—323.

[CaWo03] Antonio Carzaniga and Alexander L. Wolf: *Forwarding in a Content-Based Network*, Proceedings of the ACM SIGCOMM Conference, 2003, 163—174.

[Car98] Antonio Carzaniga: *Architectures for an Event Notification Service Scalable to Wide-area Networks*, Politecnico Di Milano, 1998, PhD Thesis.

[CarDay89] Dennis R. McCarthy and Umeshwar Dayal: *The architecture of an active database management system*, Proceedings of the 1989 ACM SIGMOD international conference on Management of data, 1989, 215—224.

[ChKo02] Guanling Chen and David Kotz: *Context Aggregation and Dissemination in Ubiquituos Computing Systems*, Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '02), 2002.

[Che03] Christa M. Chewar: *Modeling User Goals for Notification System Interfaces*, Virginia Polytechnic Institute and State University, 2003, PhD Thesis.

[Che04] Guanling Chen: *Solar: Building A Context Fusion Network for Pervasive Computing*, Dartmouth College, 2004, PhD Thesis.

[CoGr93] A. Cockburn and S. Greenberg: *Making Contact: Getting the Group Communicating with Groupware*, Proceedings of the ACM COCS'93 Conference on Organizational Computing Systems, November 1993.

[CoMa03] P. Cottone and G. Mantovani: *Grounding "subjective views" — Situation awareness and co-reference in distance learning*, Being There — Concepts, effects and measurements of user presence in synthetic environments, Ios Press, 2003, G. Riva, F. Davide, W. A. Ijsselsteijn, Amsterdam.

[CoTh93] Andrew Cockburn and Harold W. Thimbleby: *Reducing User Effort in Collaboration Support*, Intelligent User Interfaces, 1993, 215—218.

[DOM Level2] M. Champion, S. Byrne, A. Hors, G. Nicol, J. Robie, L. Wood, and P. Hégaret: *Document Object Model (DOM) Level 2 Core Specification*, World Wide Web Consortium Recommendation REC-DOM-Level-2-Core-20001113, November 2000.

[DOM Level2 Events] T. Pixley: *Document Object Model (DOM) Level 2 Events Specification*, World Wide Web Consortium Recommendation REC-DOM-Level-2-Events-20001113, November 2000.

[DZK+02] Frank Dabek, Nickolai Zeldovich, M. Frans Kaashoek, David Mazières, and Robert Morris: *Event-driven programming for robust software*, Proceedings of the 10th ACM SIGOPS European Workshop, September 02, 186—189.

[DaKr03] Laura Dabbish and Robert Kraut: *Coordinating Communication: Awareness Displays and Interruption*, Proceedings of the CHI 2003, 786—787.

[DaKr04] Laura Dabbish and Robert Kraut: *Controlling Interruptions: Awareness Displays and Social Motivation for Coordination*, Paper presented at CSCW 2004.

[Den82] Peter J. Denning: *Electronic Junk*, Communications of the ACM, March 1982, 163—165.

[DoBl92] Paul Dourish and Sara Bly: *Portholes: Supporting Awareness in a Distributed Work Group*, Proceedings of the CHI 92 Conference on Human Factors in Computing Systems, 1992, 541—547.

[DuMu98] Du Li and Richard Muntz: *COCA: Collaborative Objects Coordination Architecture*, Proceedings of ACM CSCW '98 Conference on Computer Supported Cooperative Work, November 1998, 179—1988.

[ECMA-262] *ECMAScript Language Specification, 3rd Edition*, December 1999.

[ECMA-357] *ECMAScript for XML (E4X) Specification*, June 2004.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein: *Groupware — Some Issues and Experiences*, Communications of the ACM, January 1991.

[Egi88] Carmen Egido: *Videoconferencing as a Technology to Support Group Work: A Review of its Failure*, Proceedings of the ACM 1988 Conference on Computer Supported Cooperative Work, September 1988, 13—24.

[FHL04d] James Fogarty, Scott E. Hudson, and Jennifer Lai: *Examining the Robustness of Sensor-Based Statistical Models of Human Interruptibility*, ACM Conference on Human Factors in Computing Systems, April 2004, 207—214.

[FKC90] Robert S. Fish, Robert E. Kraut, and Barbara L. Chalfonte: *The VideoWindow System in Informal Communications*, Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work, October 1990, 1—11.

[FLC04b] James Fogarty, Jennifer Lai, and Jim Christensen: *Presence versus availability: the design and evaluation of a context-aware communication client*, International Journal of Human-Computer Studies, September 2004, 299—317.

[Fal03] Deborah Fallows: *Spam — How It Is Hurting Email and Degrading Life on the Internet*, Pew Internet & American Life Project, October 2003.

[Fal05] Deborah Fallows: *CAN-SPAM a year later*, Pew Internet & American Life Project, April 2005.

[FiTa00a] R. T. Fielding and R. N. Taylor: *Principled Design of the Modern Web Architecture*, Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000), June 2000, Limerick, Ireland, 407—416.

[FiTa02] Roy T. Fielding and Richard N. Taylor: *Principled Design of the Modern Web Architecture*, ACM Transactions on Internet Technology, May 2002, 115—150.

[Fie00b] R. T. Fielding: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000, PhD Thesis.

[Fog04c] James Fogarty: *AmIBusy: High-Level Abstraction in Ubiquitous Sensing Environments*, Pervasive 2004 Workshop on Toolkit Support in the Physical World, 2004.

[GAM03] Manuel Görtz, Ralf Ackermann, Andreas Mauthe, and Ralf Steinmetz: *Using Context Information to Avoid Service Interactions in IP Telephony*, Proceedings of the first international workshop on Multimedia Interactive Protocols and Systems (MIPS), November 2003, 340—351.

[GAS04] Manuel Görtz, Ralf Ackermann, and Ralf Steinmetz: *Enhanced SIP Communication Services by Context Sharing*, Proceedings of the 30th EUROMICRO Conference, September 2004, 272—279.

[GBS00] H.W. Gellersen, M. Beigl, and A. Schmidt: *Sensor-based Context-Awareness for Situated Computing*, Workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC00), June 2000.

[GGW+00] Steve Greenspan, David Goldberg, David Weimer, and Andrea Basso: *Interpersonal trust and common ground in electronically mediated communication*, Proceedings of the 2000 ACM conference on Computer supported cooperative work, 2000, 251—260.

[GHJ+00] Patrice Godefroid, James A. Herbsleb, Lalita Jategaonkar Jagadeesan, and Du Li: *Ensuring Privacy in Presence Awareness Systems: An Automated Verification Approach*, Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work, December 2000, ACM Press, Philadelphia, PA, USA, 59—68.

[GMM+92] William Gaver, Thomas Moran, Allan MacLean, Lennart Lövstrand, Paul Dourish, Kathleen Carter, and William Buxton: *Realizing a Video Environment: EuroPARC's RAVE System*, Proceedings of the CHI 92 Conference on Human Factors in Computing Systems, May 1992, 27—35.

[GrBe97] Chris Greenhalgh and Steve Benford: *Boundaries, Awareness and Interaction in Collaborative Virtual Environments*, Proceedings of the 6th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), June 1997.

[GrPa02] Rebecca E. Grinter and Leysia Palen: *Instant Messaging in Teen Life*, Proceedings of the ACM 2002 Conference on Computer Supported Cooperative Work, November 2002, ACM Press, 21—30.

[Gre88] Irene Greif: *Computer-Supported Cooperative Work: A Book Of Readings*, Morgan Kaufmann, 1988, I. Greif.

[Gru88] Jonathan Grudin: *Why CSCW Applications Fail: Problems in the Design and Evaluation of Organization Interfaces*, Proceedings of the ACM 1988 Conference on Computer Supported Cooperative Work, September 1988, 85—93.

[Gör05] Manuel Görtz: *Effiziente Echtzeit-Kommunikationsdienste durch Einbeziehung von Kontexten*, TU Darmstadt, 2005, PhD Thesis.

[HCB+99] Mark Handley, Jon Crowcroft, Carsten Bormann, and Jörg Ott: *Very Large Conferences on the Internet: the Internet Multimedia Conferencing Architecture*, Computer Networks, The International Journal of Computer and Telecommunications Networking, Special Issue on Internet Telephony, Elsevier, North Holland, July 1999, 191—204.

[HFA+03] Scott E. Hudson, James Fogarty, Christopher G. Atkeson, Daniel Avrahami, Jodi Forlizzi, Sara Kiesler, Johnny C. Lee, and Jie Yang: *Predicting human interruptibility with sensors: a Wizard of Oz feasibility study*, Proceedings of human factors in computing systems, 2003, 257—264.

[HHP+97] Hyong Sop Shim, Robert W. Hall, Atul Prakash, and Farnam Jahanian: *Providing Flexible Services for Managing Shared State Collaborative Systems*, Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 1997, Kluwer, 237—252.

[HHS+02] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster: *The anatomy of a context-aware application*, Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking, 1999, 59—68.

[HKK+02] Eric Horvitz, Paul Koch, Carl M. Kadie, and Andy Jacobs: *Coordinate: Probabilistic Forecasting of Presence and Availability*, Proceedings of the Eighteenth Conference on Uncertainty and Artificial Intelligence, July 2002, Morgan Kaufmann, 224—233.

[HKP+03] Eric Horvitz, Carl Kadie, Tim Paek, and David Hovel: *Models of attention in computing and communication: from principles to applications*, Communications of the ACM, 2003, 52—59.

[HaHo94] Andy Harter and Andy Hopper: *A Distributed Location System for the Active Office*, IEEE Network, 1994, 62—70.

[Har04] Tobias Hartmann: *Entwurf und Implementierung eines Systems zur Aggregation und Distribution von SIP-basierten Presence-Informationen*, Universität Bremen, 2004, Master's Thesis.

[HeGr99] James D. Herbsleb and Rebecca E. Grinter: *Architectures, Coordination, and Distance: Conway's Law and Beyond*, IEEE Software, 1999, 63—70.

[HiBe03] Rosco Hill and James "Bo" Begole: *Activity Rhythm Detection and Modelling*, Proceedings of the ACM Conference on Human Factors in Computing Systems, 2003, 782—783.

[HiTu85] Starr R. Hiltz and Murray Turoff: *Structuring computer-mediated communication systems to avoid information overload*, Communications of the ACM, 1985, 680—689.

[HoSt92] Jim Hollan and Scott Stornetta: *Beyond being there*, Proceedings of the SIGCHI conference on Human factors in computing systems, 1992, 119—125.

[Hüt01] Brian Hüttner: *Business-to-Business-Kommunikation mit strukturierten Dokumenten: ein Tool zur Transformation von XML-Dokumenten mit prozedural erweiterbaren Transformationsvorschriften*, Universität Bremen, 2001, Master's Thesis.

[IMPPWG] M. Day and D. Atkins: *IMPP Working Group History and De Facto Charter*, Internet-Draft, draft-day-atkins-impp-defacto-00, work in progress, June 2003.

[ISO93] *Information Technology — Universal Multiple-octet coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*, ISO Standard 10646-1, 1993.

[IWR02a] Ellen Isaacs, Alan Walendowski, and Dipti Ranganthan: *Hubbub: a sound-enhanced mobile instant messenger that supports awareness and opportunistic interactions*, Proceedings of the SIGCHI conference on Human factors in computing systems, 2002, 179—186.

[IWR02b] Ellen Isaacs, Alan Walendowski, and Dipti Ranganathan: *Mobile instant messaging through Hubbub*, Communications of the ACM, 2002, 68—72.

[IWW+02c] Ellen Isaacs, Alan Walendowski, Steve Whittaker, Diane J. Schiano, and Candace Kamm: *The character, functions, and styles of instant messaging in the workplace*, Proceedings of the 2002 ACM conference on Computer supported cooperative work, 2002, 11—20.

[IjRi03] W. A. Ijsselsteijn and G. Riva: *Being There: The experience of presence in mediated environments*, Being There — Concepts, effects and measurements of user presence in synthetic environments, Ios Press, 2003, G. Riva, F. Davide, W. A. Ijsselsteijn, Amsterdam.

[JEP-0060] P. Millard, P. Saint-Andre, and R. Meijer: *Publish-Subscribe*, JSF JEP 0060, September 2006.

[JEP-0080] J. Hildebrand and P. Saint-Andre: *User Geolocation*, JSF JEP 0080, August 2006.

[JEP-0107] P. Saint-Andre and R. Meijer: *User Mood*, JSF JEP 0107, October 2004.

[JEP-0108] R. Meijer and P. Saint-Andre: *User Activity*, JSF JEP 0108, October 2004.

[JEP-0118] P. Saint-Andre: *User Tune*, JSF JEP 0118, November 2004.

[JEP-0163] P. Saint-Andre and K. Smith: *Personal Eventing via Pubsub*, JSF JEP 0163, September 2006.

[JEP-0166] S. Ludwig, J. Beda, P. Saint-Andre, J. Hildebrand, S. Egan, and R. McQueen: *Jingle*, JSF JEP 0166, September 2006.

[JLG91] M. Jirotka, P. Luff, and N. Gilbert: *Participation Frameworks For Computer Mediated Communication*, Proceedings of the Second European Conference on Computer Supported Cooperative Work, L. Bannon, M. Robinson, K. Schmidt, September 1991, Amsterdam, NL, 279—291.

[KMM02] Birgitta König-Ries, Kia Makki, S. A. M. Makki, Charles E. Perkins, Niki Pissinou, Peter Reiher, Peter Scheuermann, Jari Veijalainen, Alexander Wolf, and Ouri Wolfson: *Research Direction for Developing an Infrastructure for Mobile & Wireless Systems: Consensus Report of the NSF Workshop Held on October 15, 2001 in Scottsdale, Arizona*, IMWS 2001, B. König-Ries et al., Springer, Berlin, Heidelberg, 2002, 1—37.

[Kan01] Theo Kanter: *Adaptive Personal Mobile Communication, Service Architecture and Protocols*, Kungl Tekniska Høskolan (KTH), 2001, Stockholm, PhD Thesis.

[Kan02] Theo Kanter: *Extensible Mobile Presence*, Proceedings of the 4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN02), September 2002, Arlanda.

[Kan03a] Theo Kanter: *Attaching Context-Aware Services to Moving Locations*, Internet Computing, IEEE Computer Society, March 2003, 43—51.

[Kan03b] Theo Kanter: *Cooperative Mobile Ambient Awareness*, Proceedings of the MobEA Workshop collocated with WWW2003 conference 2003, May 2003, Budapest, Hungary.

[KhRi98a] Rohit Khare and Adam Rifkin: *Scenarios for an Internet-Scale Event Notification Service (ISENS)*, Internet Draft draft-khare-notification-00.txt (Work in Progress), November 1998.

[Kha03] Rohit Khare: *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems*, University Of California, Irvine, 2003, PhD Thesis.

[KrEg88] Robert Kraut and Carmen Egido: *Patterns of Contact and Communication in Scientific Research Collaboration*, Proceedings of the ACM 1988 Conference on Computer Supported Cooperative Work, September 1988, 1—12.

[KuOt03] Dirk Kutscher and Jörg Ott: *Dynamic Device Access for Mobile Users*, Proceedings of the Eighth International Conference on Personal Wireless Communications, September 2003, 53—64.

[Kul03] Joanna Kulik: *Fast and flexible forwarding for Internet subscription systems*, Proceedings of the 2nd international workshop on Distributed event-based systems, 2003.

[Kut01] Dirk Kutscher: *The Message Bus: Guidelines for Application Profile Writers*, Internet Draft draft-ietf-mmusic-mbus-guidelines-00.txt. Work in Progress, February 2001.

[Kut03] Dirk Kutscher: *Local Coordination for Interpersonal Communication Systems*, Universität Bremen, 2003, PhD Thesis.

[LAL01] Kristof Van Laerhoven, Kofi A. Aidoo, and Steven Lowette: *Real-time Analysis of Data from Many Sensors with Neural Networks*, Proceedings of the 5th IEEE International Symposium on Wearable Computers, 2001, 115—122.

[MBS+91] Marilyn M. Mantei, Ronald M. Baecker, Abigail J. Sellen, William A. S. Buxton, Thomas Milligan, and Barry Wellman: *Experiences in the Use of a Media Space*, Proceedings of the CHI 91 Conference on Human Factors in Computing Systems, 1991, 203—209.

[MGL+92] Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Ramana Rao, and David A. Rosenblitt: *Readings in Groupware and Computer Supported Cooperative Work*, Morgan Kaufmann, 1992, 461—473.

[MGT+87] T. Malone, K. Grant, F. Turbak, S. Brobst, and M. Cohen: *Intelligent Information Sharing Systems*, Communications of the ACM, 1987, 484—497.

[MIH+03] P. Markopoulos, W. Ijsselsteijn, C. Huijnen, O. Romijn, and A. Philopoulos: *Supporting Social Presence Through Asynchronous Awareness Systems*, Being There — Concepts, effects and measurements of user presence in synthetic environments, Ios Press, 2003, G. Riva, F. Davide, W. A. Ijsselsteijn, Amsterdam.

[MaCo90] M. Lynne Markus and Terry Connolly: *Why CSCW Applications Fail: Problems in the Adoption of Interdependent Work Tools*, Proceedings of the ACM 1990 Conference on Computer Supported Cooperative Work, October 1990, 371—380.

[Mac88] Wendy E. Mackay: *More Than Just a Communication System: Diversity in the Use of Electronic Mail*, Proceedings of the CSCW 1988, September 1988, 344—353.

[Mac90] Wendy E. Mackay: *Patterns of Sharing Customisable Software*, Proceedings of the CSCW 1990, October 1990, 209—221.

[Mar02] Joshua A. Marshall: *An Essay on Effective Telepresence*, University of Toronto, April 2002.

[McMo94] John C. McCarthy and Andrew F. Monk: *Channels, Conversation, Cooperation And Relevance: All You Wanted To Know About Communication But Were Afraid To Ask*, Collaborative Computing, March 1994, 35—60.

[MiSm00] Allen E. Milewski and Thomas M. Smith: *Providing Presence Cues to Telephone Users*, Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work, December 2000, ACM Press, Philadelphia, PA, USA, 89—96.

[MoLa99] R. Movva and W. Lai: *MSN Messenger Service 1.0 Protocol*, draft-movva-msn-messenger-protocol-00.txt. Work in progress, August 1999.

[Moo02] T. Moors: *A critical review of "End-to-end arguments in system design"*, Proc. International Conference on Communications (ICC), 2002, 1214—1219.

[NEL91] William M. Newman, Margery A. Eldridge, and Michael G. Lamming: *PEPYS: Generating Autobiographies by Automatic Tracking*, Proceedings of the Second European Conference on Computer-Supported Cooperative Work, September 1991, 175—188.

[NLM+97] T. Narine, A. Leganchuk, M. Mantei, and W. Buxton: *Collaboration awareness and its use to consolidate a disperse group*, Proccedings of Interact '97, Sydney, Australia, July 1997.

[NWB00] Bonnie A. Nardi, Steve Whittaker, and Erin Bradner: *Interaction and Outeraction: Instant Messaging in Action*, Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work, December 2000, ACM Press, Philadelphia, PA, USA, 79—88.

[Nel81] Theodor Holm Nelson: *Literary Machines*, Mindful Press, 1981.

[Nie03] Jakob Nielsen: *IM Not IP (Information Pollution)*, ACM Queue: Instant Messaging: The Net's Real Killer App?, November 2003, 75—76.

[OKB+02] Jörg Ott, Dirk Kutscher, Andreas Büsching, and Olaf Bergmann: *DTI: Desk-area Telephony Integration — Final Project Deliverable*, Technologie-Zentrum Informatik, November 2002, Bremen.

[OKB+03] Jörg Ott, Dirk Kutscher, Andreas Büsching, and Olaf Bergmann: *FETA: Functional Enhancements using External Telephony Applications — Final Project Deliverable*, Technologie-Zentrum Informatik, October 2003, Bremen.

[OKB+04] Jörg Ott, Dirk Kutscher, Andreas Büsching, and Olaf Bergmann: *PASST: Presence Aggregation SySTem — Final Project Deliverable*, Technologie-Zentrum Informatik, 2004, Bremen.

[OKB+05] Jörg Ott, Dirk Kutscher, Eilert Brinkmann, Andreas Büsching, and Olaf Bergmann: *GEOCOOP: Geo-based Services Enabling Cooperation — Final Project Deliverable*, Technologie-Zentrum Informatik, December 2005, Bremen.

[OtKu05] Jörg Ott and Dirk Kutscher: *A Disconnection-Tolerant Transport for Drive-Thru Internet Environments*, Proceedings of Infocom 2005, March 2005, Miami.

[Ous96] John K. Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*, January 1996.

[Ous98] John K. Ousterhout: *Scripting: Higher Level Programming for the 21st Century*, IEEE Computer, March 1998, 23—30.

[Part] M. Lonnfors: *Presence Information Data format (PIDF) Extension for Partial Presence*, Internet-Draft, draft-ietf-simple-partial-pidf-format-07, work in progress, July 2006.

[PeSo97] Elin Rønby Pedersen and Tomas Sokoler: *AROMA: abstract representation of presence supporting mutual awareness*, Proceedings of the SIGCHI conference on Human factors in computing systems, 1997, 51—58, ACM Press New York, NY, USA.

[Ped98] Elin Rønby Pedersen: *People presence or room activity supporting peripheral awareness over distance*, CHI 98 conference summary on Human factors in computing systems, 1998, 283—284.

[Policy] H. Schulzrinne: *Common Policy: A Document Format for Expressing Privacy Preferences*, Internet-Draft, draft-ietf-geopriv-common-policy-11, work in progress, August 2006.

[Prescaps] M. Lonnfors and K. Kiss: *Session Initiation Protocol (SIP) User Agent Capability Extension to Presence Information Data Format (PIDF)*, Internet-Draft, draft-ietf-simple-prescaps-ext-07, work in progress, July 2006.

[Pri99] Wolfgang Prinz: *NESSIE: An Awareness Environment for Cooperative Settings*, Proceedings of the Sixth European Conference on Computer Supported Cooperative Work, 1999, 391—410.

[RDF] G. Klyne and J. Carroll: *Resource Description Framework (RDF): Concepts and Abstract Syntax*, World Wide Web Consortium Recommendation REC-rdf-concepts-20040210, February 2004.

[RDR98] Devina Ramaduny, Alan Dix, and Tom Rodden: *Getting to Know: the design space for notification servers*, Proceedings of CSCW'98, 1998, 227—235.

[RFC1034] P. Mockapetris: *Domain names - concepts and facilities*, STD 13, RFC 1034, November 1987.

[RFC1035] P. Mockapetris: *Domain names - implementation and specification*, STD 13, RFC 1035, November 1987.

[RFC1288] D. Zimmerman: *The Finger User Information Protocol*, RFC 1288, December 1991.

[RFC1633] B. Braden, D. Clark, and S. Shenker: *Integrated Services in the Internet Architecture: an Overview*, RFC 1633, June 1994.

[RFC1831] R. Srinivasan: *RPC: Remote Procedure Call Protocol Specification Version 2*, RFC 1831, August 1995.

[RFC2045] N. Freed and N.S. Borenstein: *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, RFC 2045, November 1996.

[RFC2222] J.G. Myers: *Simple Authentication and Security Layer (SASL)*, RFC 2222, October 1997.

[RFC2246] T. Dierks and C. Allen: *The TLS Protocol Version 1.0*, RFC 2246, January 1999.

[RFC2396] T. Berners-Lee, R.T. Fielding, and L. Masinter: *Uniform Resource Identifiers (URI): Generic Syntax*, RFC 2396, August 1998.

[RFC2475] S. Blake, D.L. Black, M.A. Carlson, E. Davies, Z. Wang, and W. Weiss: *An Architecture for Differentiated Services*, RFC 2475, December 1998.

[RFC2616] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee: *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616, June 1999.

[RFC2617] J. Franks, P.M. Hallam-Baker, J.L. Hostetler, S.D. Lawrence, P.J. Leach, A. Luotonen, and L. Stewart: *HTTP Authentication: Basic and Digest Access Authentication*, RFC 2617, June 1999.

[RFC2778] M. Day, J. Rosenberg, and H. Sugano: *A Model for Presence and Instant Messaging*, RFC 2778, February 2000.

[RFC2779] M. Day, S. Aggarwal, G. Mohr, and J. Vincent: *Instant Messaging / Presence Protocol Requirements*, RFC 2779, February 2000.

[RFC2817] R. Khare and S. Lawrence: *Upgrading to TLS Within HTTP/1.1*, RFC 2817, May 2000.

[RFC2824] J. Lennox and H. Schulzrinne: *Call Processing Language Framework and Requirements*, RFC 2824, May 2000.

[RFC2865] C. Rigney, S. Willens, A. Rubens, and W. Simpson: *Remote Authentication Dial In User Service (RADIUS)*, RFC 2865, June 2000.

[RFC3259] J. Ott, C. Perkins, and D. Kutscher: *A Message Bus for Local Coordination*, RFC 3259, April 2002.

[RFC3261] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler: *SIP: Session Initiation Protocol*, RFC 3261, June 2002.

[RFC3263] J. Rosenberg and H. Schulzrinne: *Session Initiation Protocol (SIP): Locating SIP Servers*, RFC 3263, June 2002.

[RFC3265] A.B. Roach: *Session Initiation Protocol (SIP)-Specific Event Notification*, RFC 3265, June 2002.

[RFC3311] J. Rosenberg: *The Session Initiation Protocol (SIP) UPDATE Method*, RFC 3311, October 2002.

[RFC3325] C. Jennings, J. Peterson, and M. Watson: *Private Extensions to the Session Initiation Protocol (SIP) for Asserted Identity within Trusted Networks*, RFC 3325, November 2002.

[RFC3339] G. Klyne and C. Newman: *Date and Time on the Internet: Timestamps*, RFC 3339, July 2002.

[RFC3340] M.T. Rose, G. Klyne, and D.H. Crocker: *The Application Exchange Core*, RFC 3340, July 2002.

[RFC3341] M.T. Rose, G. Klyne, and D.H. Crocker: *The Application Exchange (APEX) Access Service*, RFC 3341, July 2002.

[RFC3343] M. Rose, G. Klyne, and D. Crocker: *The Application Exchange (APEX) Presence Service*, RFC 3343, April 2003.

[RFC3401] M. Mealling: *Dynamic Delegation Discovery System (DDDS) Part One: The Comprehensive DDDS*, RFC 3401, October 2002.

[RFC3402] M. Mealling: *Dynamic Delegation Discovery System (DDDS) Part Two: The Algorithm*, RFC 3402, October 2002.

[RFC3403] M. Mealling: *Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database*, RFC 3403, October 2002.

[RFC3404] M. Mealling: *Dynamic Delegation Discovery System (DDDS) Part Four: The Uniform Resource Identifiers (URI)*, RFC 3404, October 2002.

[RFC3466] M. Day, B. Cain, G. Tomlinson, and P. Rzewski: *A Model for Content Internetworking (CDI)*, RFC 3466, February 2003.

[RFC3470] S. Hollenbeck, M. Rose, and L. Masinter: *Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols*, BCP 70, RFC 3470, January 2003.

[RFC3530] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck: *Network File System (NFS) version 4 Protocol*, RFC 3530, April 2003.

[RFC3568] A. Barbir, B. Cain, R. Nair, and O. Spatscheck: *Known Content Network (CN) Request-Routing Mechanisms*, RFC 3568, July 2003.

[RFC3680] J. Rosenberg: *A Session Initiation Protocol (SIP) Event Package for Registrations*, RFC 3680, March 2004.

[RFC3694] M. Danley, D. Mulligan, J. Morris, and J. Peterson: *Threat Analysis of the Geopriv Protocol*, RFC 3694, February 2004.

[RFC3725] J. Rosenberg, J. Peterson, H. Schulzrinne, and G. Camarillo: *Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)*, BCP 85, RFC 3725, April 2004.

[RFC3761] P. Faltstrom and M. Mealling: *The E.164 to Uniform Resource Identifiers (URI) Dynamic Delegation Discovery System (DDDS) Application (ENUM)*, RFC 3761, April 2004.

[RFC3764] J. Peterson: *enumservice registration for Session Initiation Protocol (SIP) Addresses-of-Record*, RFC 3764, April 2004.

[RFC3840] J. Rosenberg, H. Schulzrinne, and P. Kyzivat: *Indicating User Agent Capabilities in the Session Initiation Protocol (SIP)*, RFC 3840, August 2004.

[RFC3842] R. Mahy: *A Message Summary and Message Waiting Indication Event Package for the Session Initiation Protocol (SIP)*, RFC 3842, August 2004.

[RFC3856] J. Rosenberg: *A Presence Event Package for the Session Initiation Protocol (SIP)*, RFC 3856, August 2004.

[RFC3857] J. Rosenberg: *A Watcher Information Event Template-Package for the Session Initiation Protocol (SIP)*, RFC 3857, August 2004.

[RFC3858] J. Rosenberg: *An Extensible Markup Language (XML) Based Format for Watcher Information*, RFC 3858, August 2004.

[RFC3859] J. Peterson: *Common Profile for Presence (CPP)*, RFC 3859, August 2004.

[RFC3860] J. Peterson: *Common Profile for Instant Messaging (CPIM)*, RFC 3860, August 2004.

[RFC3861] J. Peterson: *Address Resolution for Instant Messaging and Presence*, RFC 3861, August 2004.

[RFC3863] H. Sugano, S. Fujimoto, G. Klyne, A. Bateman, W. Carr, and J. Peterson: *Presence Information Data Format (PIDF)*, RFC 3863, August 2004.

[RFC3880] J. Lennox, X. Wu, and H. Schulzrinne: *Call Processing Language (CPL): A Language for User Control of Internet Telephony Services*, RFC 3880, October 2004.

[RFC3903] A. Niemi: *Session Initiation Protocol (SIP) Extension for Event State Publication*, RFC 3903, October 2004.

[RFC3920] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Core*, RFC 3920, October 2004.

[RFC3921] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, RFC 3921, October 2004.

[RFC3922] P. Saint-Andre: *Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM)*, RFC 3922, October 2004.

[RFC3923] P. Saint-Andre: *End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP)*, RFC 3923, October 2004.

[RFC3986] T. Berners-Lee, R. Fielding, and L. Masinter: *Uniform Resource Identifier (URI): Generic Syntax*, STD 66, RFC 3986, January 2005.

[RFC4122] P. Leach, M. Mealling, and R. Salz: *A Universally Unique IDentifier (UUID) URN Namespace*, RFC 4122, July 2005.

[RFC4189] K. Ono and S. Tachimoto: *Requirements for End-to-Middle Security for the Session Initiation Protocol (SIP)*, RFC 4189, October 2005.

[RFC4287] M. Nottingham and R. Sayre: *The Atom Syndication Format*, RFC 4287, December 2005.

[RFC4479] J. Rosenberg: *A Data Model for Presence*, RFC 4479, July 2006.

[RFC4480] H. Schulzrinne, V. Gurbani, P. Kyzivat, and J. Rosenberg: *RPID: Rich Presence Extensions to the Presence Information Data Format (PIDF)*, RFC 4480, July 2006.

[RFC4481] H. Schulzrinne: *Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals*, RFC 4481, July 2006.

[RFC4482] H. Schulzrinne: *CIPID: Contact Information for the Presence Information Data Format*, RFC 4482, July 2006.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker: *A Scalable Content-Addressable Network*, Proceedings of the ACM SIGCOMM 2001.

[RKC+01] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel: *SCRIBE: The Design of a Large-Scale Event Notification Infrastructure*, Networked Group Communication, 30—43, 2001.

[RSS02] Sylvia Ratnasamy, S. Shenker, and I. Stoica: *Routing Algorithms for DHTs: Some open questions*, IPTPS, March 2002.

[Ran02] Matti Rantanen: *A Presence Service for Ubiquitous Computing*, Helsinki Institute of Information Technology, 2002, PhD Thesis.

[Rat02] Sylvia Ratnasamy: *A Scalable Content-Addressable Network*, University of California at Berkeley, 2002, PhD Thesis.

[ReGo03] J. Rennecker and L. Goodwin: *Theorizing the Unintended Consequences of Instant Messaging (IM) for Worker Productivity*, Sprouts: Working Papers on Information Environments, Systems and Organizations, 2003.

[RiKh98b] Adam Rifkin and Rohit Khare: *The Evolution of Internet-Scale Event Notification Services: Past, Present, and Future*, August 1998.

[RoWo97] David S. Rosenblum and Alexander L. Wolf: *A Design Framework for Internet-Scale Event Observation and Notification*, Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering, 1997, 344—360.

[Roo88] Robert W. Root: *Design of a Multi-Media Vehicle for Social Browsing*, Proceedings of the ACM 1988 Conference on Computer Supported Cooperative Work, September 1988, 25—38.

[Rou99] Nicolas Roussel: *Mediascape: A Web-Based Media Space*, IEEE Multimedia, 1999, 64—74.

[RyWo03] Nathan D. Ryan and Alexander L. Wolf: *Using Event-Based Parsing to Support Dynamic Protocol Evolution*, University Of Colorado, Department of Computer Science, March 2003.

[SAT+99] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven, and Walter Van de Velde: *Advanced Interaction in Context*, Lecture Notes In Computer Science, Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, 1999, 89—101.

[SBB97] Ovidiu Sandor, Cristian Bogdan, and John Bowers: *Aether: An Awareness Engine for CSCW*, Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 1997, Kluwer, 221—236.

[SCP91] Eve M. Schooler, Steven L. Casner, and Jon Postel: *Multimedia Conferencing: Has it Come of Age?*, University of Southern California — Information Sciences Institute, August 1991.

[SGF02] Rüdiger Schollmeier, Ingo Gruber, and Michael Finkenzeller: *Routing in Mobile Ad Hoc and Peer-to-Peer Networks. A Comparison*, Lecture Notes In Computer Science. Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing, Springer, 2002, 172—186.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, Proceedings of the ACM SIGCOMM 2001, 2001, 149—160.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark: *End-to-end arguments in system design*, ACM Transactions on Computer Systems (TOCS), November 1984, 277—288.

[SSR03] Roberto S. Silva Filho, Cleidson R. B. De Souza, and David F. Redmiles: *The design of a configurable, extensible and dynamic notification service*, Proceedings of the 2nd international workshop on Distributed event-based systems, 2003.

[SVV97] Cheri Speier, Joseph S. Valacich, and Iris Vessey: *The effects of task interruption and information presentation on individual decision making*, Proceedings of the eighteenth international conference on Information systems, 1997, 21—36.

[Sal04] Peter Salin: *Mobile Instant Messaging Systems: A Comparative Study and Implementation*, Helsinki University of Technology, September 2004, Master's Thesis.

[ScBa92] Kjeld Schmidt and Liam Bannon: *Taking CSCW Seriously — Supporting Articulation Work*, Computer Supported Cooperative Work, 1992, 7—40.

[Sch02] Kjeld Schmidt: *The Problem with 'Awareness': Introductory Remarks on 'Awareness in CSCW'*, Computer Supported Cooperative Work (CSCW), 2002, 285—298.

[Scho01] Rüdiger Schollmeier: *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications*, Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001), August 2001.

[SiSc04a] Kundan Singh and Henning Schulzrinne: *Peer-to-Peer Internet Telephony using SIP*, Department of Computer Science, Columbia University, 2004.

[SiSc04b] Kundan Singh and Henning Schulzrinne: *SIPpeer: A Session Initiation Protocol (SIP)-based Peer-to-Peer Internet Telephony Client Adaptor*, Department of Computer Science, Columbia University, 2004.

[Sik97] Klaas Sikkel: *A Group-based Authorization Model for Cooperative Systems*, Proceedings of the Fifth European Conference on Computer Supported Cooperative Work, 1997, Kluwer, 345—360.

[Sin92] Alok Sinha: *Client-server computing*, Communications of the ACM, July 1992, 77—98.

[TBH+04] Joe Tullio, James "Bo" Begole, Eric Horvitz, and Elizabeth D. Mynatt: *Forecasting Presence and Availability*, ACM Conference on Human Factors in Computing Systems, April 2004, 1713—1714.

[TMA+02] Loren Terveen, Jessica McMackin, Brian Amento, and Will Hill: *Specifying Preferences Based On User History*, Proceedings of the SIGCHI conference on Human factors in computing systems, 2002.

[TRB93] Jonathan Trevor, Tom Rodden, and Gordon Blair: *COLA: A Lightweight Platform for CSCW*, Proceedings of the Third European Conference on Computer Supported Cooperative Work, 1993, Kluwer, 15—30.

[TYB+01] John C. Tang, Nicole Yankelovich, James "Bo" Begole, Max Van Kleek, Francis Li, and Janak Bhalodia: *ConNexus to Awarenex: Extending Awareness to Mobile Users*, Proceedings of the SIGCHI conference on Human factors in computing systems, 2001, 221—228.

[TaBe03] John C. Tang and James "Bo" Begole: *Beyond Instant Messaging*, ACM Queue: Instant Messaging: The Net's Real Killer App?, November 2003, 28—37.

[WCK04] Jue Wang, Guanling Chen, and David Kotz: *A sensor-fusion approach for meeting detection*, Proceedings of the Workshop on Context Awareness at the Second International Conference on Mobile Systems, Applications, and Services, 2004.

[XCAP] J. Rosenberg: *The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)*, Internet-Draft draft-ietf-simple-xcap-11, work in progress, May 2006.

[XML] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler: *Extensible Markup Language (XML) 1.0 (2nd ed)*, W3C REC-xml, October 2000.

[XML Names] T. Bray, D. Hollander, and A. Layman: *Namespaces in XML*, W3C REC-xml-names, January 1999.

[XPIDF] Jonathan Rosenberg, Dean Willis, Robert Sparks, Ben Campbell, Henning Schulzrinne, Jonathan Lennox, Bernard Aboba, Christian Huitema, and David Gurle: *A Data Format for Presence Using XML*, draft-rosenberg-impp-pidf-00. Work in progress, June 2000.

[XPath] J. Clark and S. DeRose: *XML Path Language (XPath) Version 1.0*, World Wide Web Consortium Recommendation REC-xpath-19991116, November 1999.

[XSL] S. Deach, J. Caruso, T. Graham, P. Grosso, E. Gutentag, S. Parnell, R. Milowski, J. Richman, S. Zilles, A. Berglund, and S. Adler: *Extensible Stylesheet Language (XSL) Version 1.0*, World Wide Web Consortium Recommendation REC-xsl-20011015, October 2001.

[XSLT] J. Clark: *XSL Transformations (XSLT) Version 1.0*, W3C REC-xslt, November 1999.

[YiPl03] Ying Liu and Beth Plale: *Survey of Publish Subscribe Event Systems*, Computer Science Dept., Indiana University, IN, 2003.