

Introducing the XVSM Micro-Room Framework

Creating a Privacy Preserving Peer-to-Peer Online Social Network in a Declarative Way

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Johann Binder

Matrikelnummer 0727950

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ.-Prof. Dipl.-Ing. Dr. Eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Stefan Craß

Wien, 27. August 2013

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Introducing the XVSM Micro-Room Framework

Creating a Privacy Preserving Peer-to-Peer Online Social Network in a Declarative Way

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Johann Binder

Registration Number 0727950

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ.-Prof. Dipl.-Ing. Dr. Eva Kühn
Assistance: Projektass. Dipl.-Ing. Stefan Craß

Vienna, 27. August 2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Johann Binder
Grimmgasse 41/28, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgements

I wish to thank several persons for their contribution to this work. First of all, I thank Eva Kühn for her useful and constructive recommendations during our regular meetings. Second, I am particularly grateful for the technical assistance and guidance given by Stefan Craß. Special credit belongs to Evelyn Binder for proof-reading the thesis. Finally, I would like to acknowledge the support provided by my family during the whole work.

Abstract

Nowadays, all popular online social networks in the World Wide Web are based on a client-server architecture. This leads to severe privacy issues as the provider of the online social network holds all data and could easily misuse it. Other contributions already showed how to resolve this problem by decentralizing the data with a peer-to-peer architecture. Yet these solutions have their own drawbacks, e.g. many complex technologies, bad extensibility or a lack of important privacy and security features.

A cause for these drawbacks is that no satisfying high-level peer-to-peer frameworks exist, allowing users to develop complex peer-to-peer applications in an easy way. Available frameworks are rather low-level, concentrating on the communication layer. Therefore, all of the peer-to-peer online social networks analysed have been created more or less from scratch.

To overcome this issue, we introduce the XVSM Micro-Room Framework which focuses on ensuring those aspects, not covered sufficiently by the existing peer-to-peer online social networks. This framework allows to create peer-to-peer applications based on configurable, shared data rooms. We then use the XVSM Micro-Room Framework for creating a proof of concept peer-to-peer online social network settled in an e-learning environment.

By comparing our solutions (both the XVSM Micro-Room Framework and the generated peer-to-peer online social network) with the analysed related work, we demonstrate the benefits of our solutions regarding functionality, simplicity, extensibility, privacy and security. Finally, several performance benchmarks show the practicability of the XVSM Micro-Room Framework even in heavy-load scenarios.

Kurzfassung

Alle bekannten sozialen Netzwerke des World Wide Webs basieren auf einer Client-Server-Architektur. Das führt zu ernststen Datenschutzproblemen, da der Anbieter des sozialen Netzwerks über alle Daten seiner Kunden verfügt und sie ohne Probleme missbräuchlich verwenden kann. Andere wissenschaftliche Beiträge haben bereits gezeigt, dass dieses Problem gelöst werden kann, indem die Daten mithilfe einer Peer-to-Peer-Architektur dezentralisiert werden. Nichtsdestotrotz haben alle diese Lösungen Nachteile, z.B. die Verwendung von vielen, komplexen Technologien, schlechte Erweiterbarkeit oder das Fehlen von wichtigen Datenschutz- und Sicherheitsmerkmalen.

Eine Ursache dafür ist, dass es keine hochrangigen Peer-to-Peer-Frameworks gibt, die es erlauben, komplexe Peer-to-Peer-Applikationen auf einfache Art und Weise zu erstellen. Die verfügbaren Peer-to-Peer-Frameworks sind eher auf einer niedrigen Ebene angesiedelt und konzentrieren sich hauptsächlich auf die Kommunikationsschicht. Deshalb mussten alle in dieser Arbeit analysierten sozialen Netzwerke auf Peer-to-Peer-Basis mehr oder weniger von Grund auf neu erstellt werden.

Um dieses Problem zu lösen, führen wir das neue XVSM-Micro-Room-Framework ein, welches Lösungen für alle Probleme der existierenden sozialen Netzwerke auf Peer-to-Peer-Basis bereitstellt. Dieses Framework erlaubt es, Peer-to-Peer-Applikationen zu erzeugen, die auf konfigurierbaren, geteilten Daten-Räumen basieren. Anschließend erstellen wir mithilfe dieses Frameworks ein soziales Netzwerk auf Peer-to-Peer-Basis, welches sich mit E-Learning beschäftigt.

Durch Vergleichen unserer Lösungen (sowohl des XVSM-Micro-Room-Frameworks, als auch des erstellten sozialen Netzwerks auf Peer-to-Peer-Basis) mit den eingangs analysierten Lösungen, werden wir die Vorteile unseres Ansatzes hinsichtlich Funktionalität, Einfachheit, Erweiterbarkeit, Datenschutz und Sicherheit demonstrieren. Schlussendlich zeigen einige Leistungstests, dass das XVSM-Micro-Room-Framework auch in Hochlast-Szenarien verwendbar ist.

Contents

1	Introduction	1
1.1	Motivation & Problem Description	1
1.2	Methodology & Expected Results	2
1.3	Structure of the Thesis	2
2	Related Work	4
2.1	Client-Server Online Social Networks	4
2.2	P2P Online Social Networks	12
2.3	P2P Frameworks	24
2.4	Conclusions	27
3	Approach	29
3.1	Micro-Room Concept	29
3.2	Technologies Used	30
3.3	Fulfillment of Requirements	35
4	Design	39
4.1	Architecture Overview	39
4.2	Components	40
4.3	Request Handling	43
4.4	Security	44
4.5	Peer Management	45
4.6	Replication	46
4.7	Workflow Support	51
5	Implementation	56
5.1	Structural View	56
5.2	Sequential View	64
5.3	Implementation Details	69
6	Deployment	78
6.1	End User Manual	78
6.2	Application Developer Manual	87
6.3	User Interface Designer Manual	93

6.4	Deployment Possibilities	102
7	Evaluation	105
7.1	Use Case Overview	105
7.2	Comparison with Related Work	119
7.3	Evaluation of Architecture and Technologies	123
7.4	Performance Benchmarks	125
8	Future Work	131
8.1	Security	131
8.2	Peer Management	131
8.3	Replication	132
8.4	Workflow Support	132
8.5	Business Logic & Module Adaption	132
8.6	XML Modeler	133
8.7	Case Study	133
9	Conclusion	134
	References	135
	Web References	141
	Appendix	I

List of Figures

2.1	Number of active users per OSN	4
2.2	Percentage of global internet users actively using an OSN	5
2.3	Facebook timeline	6
2.4	Facebook promotion feature	6
2.5	Personal data provided to Facebook	8
2.6	Privacy settings in Facebook	9
2.7	Diaspora architecture	13
2.8	Mailbook architecture	15
2.9	PeerSoN architecture	16
2.10	SafeBook user interface	18
2.11	LifeSocial user interface	19
2.12	Comparison of P2P OSNs	20
2.13	TOMSCOP communication channels	25
2.14	TOMSCOP shared web browser example	25
2.15	The P2P Application Framework peer network	26
2.16	The P2P Application Framework digital library example	27
3.1	Micro-room example	30
3.2	XVSM communication	31
3.3	XVSM container	32
3.4	XVSM aspects	32
4.1	XVSM Micro-Room Framework component view	40
4.2	XVSM Micro-Room Framework request handling	43
4.3	XVSM Micro-Room Framework correlation example	51
4.4	XVSM Micro-Room Framework standard correlation rules	54
5.1	Class diagram - Components	58
5.2	Class diagram - Modules	60
5.3	Class diagram - Objects	62
5.4	Sequence diagram - Startup	65
5.5	Sequence diagram - Login	66
5.6	Sequence diagram - Request handling	67
5.7	Sequence diagram - Logout	68

5.8	Sequence diagram - Shutdown	69
5.9	File download process	76
5.10	File upload process	77
6.1	Visualized XSD of business logic files	81
6.2	Visualized sample business logic file	83
6.3	Deployment possibilities	103
7.1	P2P OSN scenario - Borrowing books	106
7.2	P2P OSN user interface - Borrowing books	107
7.3	P2P OSN scenario - Chatting	108
7.4	P2P OSN user interface - Chatting	109
7.5	P2P OSN scenario - Discussions	110
7.6	P2P OSN user interface - Discussions	111
7.7	P2P OSN scenario - Voting	112
7.8	P2P OSN user interface - Voting	113
7.9	P2P OSN scenario - Profile	114
7.10	P2P OSN user interface - Profile	115
7.11	Blob URL example	115
7.12	P2P OSN scenario - Homework 1/2	116
7.13	P2P OSN scenario - Homework 2/2	117
7.14	P2P OSN user interface - Take homework assignment	118
7.15	P2P OSN user interface - Upload homework submission	119
7.16	P2P OSN user interface - Correct homework submission	119
7.17	Comparison of P2P OSNs including our implementation	122
7.18	Benchmark - Total CPU usage	126
7.19	Benchmark - Total RAM usage	127
7.20	Benchmark - Request acceptance time	128
7.21	Benchmark - Replication delay	129

List of Tables

2.1	Real names provided to Facebook	9
2.2	Number of vulnerable Facebook profiles	11
4.1	Single-master vs. multi-master replication	47
4.2	Full vs. partial replication	47
4.3	Synchronous vs. asynchronous replication	48
5.1	XVSM Micro-Room Framework package structure	57
5.2	XVSM Micro-Room Framework containers	64
6.1	All possible entries for config.properties	80
6.2	All relevant attributes of a business logic file	86
6.3	Abstract methods derived from BaseMicroRoom	89
6.4	Instance variables derived from BaseMicroRoom	90
6.5	Instance variables derived from BasePlugin	92

List of Listings

5.1	CORS Header sent by the framework	76
6.1	Sample configuration file - config.properties	78
6.2	Sample business logic file - Chatting.xml	82
6.3	Sample micro-room file - Reception.java	87
6.4	Sample plugin file - Creatable.java	91
6.5	Sample action implementation file - CreateAction.java	93
6.6	REST API - Request container lookup	94
6.7	REST API - Response container lookup	94
6.8	REST API - Send request	95
6.9	REST API - Read response	96
6.10	JavaScript wrapper library - Initialization	98
6.11	JavaScript wrapper library - Oneway call	99
6.12	JavaScript wrapper library - Asynchronous request-response call	99
6.13	JavaScript wrapper library - Synchronous request-response call	100
6.14	JavaScript wrapper library - Publish-subscribe registration	100
6.15	JavaScript wrapper library - Publish-subscribe deregistration	100
6.16	JavaScript wrapper library - Clean up	100
6.17	JavaScript wrapper library - File download	101
6.18	JavaScript wrapper library - File upload	101
A.1	XSD of business logic files	I

List of Abbreviations

2PC	2-Phase Commit
ABE	Attribute-Based Encryption
API	Application Programming Interface
CA	Certificate Authority
CORS	Cross-Origin Resource Sharing
CPU	Central Processing Unit
CSS	Cascading Style Sheets
CVS	Concurrent Versions System
DHT	Distributed Hash Tables
DNS	Domain Name System
FAQ	Frequently Asked Questions
FIFO	First In – First Out
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ID	Identifier
IP	Internet Protocol
JAR	Java Archive
JID	Jabber Identifier
JMS	Java Message Service
JS	JavaScript
JSON	JavaScript Object Notation
JSONP	JSON with Padding
JVM	Java Virtual Machine
JXTA	Juxtapose
LIFO	Last In – First Out
MEP	Message Exchange Pattern
NOYB	None Of Your Business
OSN	Online Social Network
P2P	Peer-to-Peer
PC	Personal Computer
PDF	Portable Document Format
PDS	Personal Data Server
PKI	Public-Key Infrastructure

POP3	Post Office Protocol 3
RAM	Random Access Memory
RDBMS	Relational Database Management System
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SOP	Same Origin Policy
SQL	Structured Query Language
SSO	Single Sign-On
TOMSCOP	Technology of Multi-user Synchronous Collaboration Platform
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WCF	Windows Communication Foundation
XACML	Extensible Access Control Markup Language
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSD	XML Schema Definition
XVSM	Extensible Virtual Shared Memory

Introduction

In this chapter we describe our motivation and introduce the problem. Furthermore, we outline the expected results and present the overall structure of this work.

1.1 Motivation & Problem Description

Today, several big *Online Social Networks* (OSN) exist in the World Wide Web. What they all have in common is that they are based on a client-server architecture. This causes serious privacy issues, as the OSN provider holds all data provided by its users and could therefore easily misuse it. As the author of the thesis uses OSNs regularly, it is a personal concern to overcome this problem and thus restrict the possibilities of data abuse.

Currently, many different approaches to resolve these privacy issues are evolving (cf. section 2.1.2). A very promising possibility is to decentralize the client-server architecture to a *Peer-to-Peer* (P2P)-based approach. However, most of the currently developed P2P OSNs suffer from several problems. First of all, they tend to be very complex as they need to make use of several different technologies (e.g. *Extensible Messaging and Presence Protocol* (XMPP), *Juxtapose* (JXTA), *Java Message Service* (JMS), *Distributed Hash Tables* (DHT) and WebFinger) to enable P2P communication. As a consequence thereof, the desired functionality of a social network is more difficult to implement in such a P2P OSN than in a classical client-server OSN.

What many of the actual P2P OSNs also neglect, is support for adapting their functionality in the future, e.g. due to new requirements. They are implemented rather statically for their special purpose, not providing any interfaces for further plugins or modules.

Another problem of many P2P OSNs is that their P2P architecture is not strict enough to achieve real privacy benefits in contrast to centralized OSNs. E.g. Diaspora¹, one of the most famous P2P OSNs, just connects its servers via P2P, while the users still need to communicate with these servers like in a centralized OSN. Therefore, the server administrators again possess large amounts of user data they could misuse without notion.

¹<http://diasporafoundation.org>

The last problem of actual P2P OSNs we deal with, is security. Some approaches, such as SafeBook [CMO11], concentrate on this issue and perform well in providing a secure P2P OSN. On the other side, those solutions, not explicitly focusing on security, again have several shortcomings, e.g. no satisfying protection against malicious users or requests.

1.2 Methodology & Expected Results

To overcome these issues (i.e. high complexity, bad extensibility and a lack of privacy and security features), we implement a new P2P framework, i.e. the XVSM Micro-Room Framework. This framework doesn't need to mix together several different technologies but is based on only one, namely *Extensible Virtual Shared Memory* (XVSM). XVSM is a space-based middleware that supports many of the different interaction patterns needed for P2P communication.

The created framework does not only abstract the communication layer (such as common P2P frameworks do) but the whole business logic as well as the data layer. Therefore, it allows designers to easily create the whole back-end of P2P applications by configuring so-called "micro-rooms" in a descriptive way (i.e. via *Extensible Markup Language* (XML)), thus reducing the overall complexity of such systems. This model-based approach also allows to easily adapt the generated P2P applications by modifying the underlying model. Additionally, the framework's capabilities themselves are extensible by modules.

A special focus is on privacy issues and security. Therefore, designers can specify in detail how data is distributed across peers, whereas users can see where their data is located and who is able to access it. Also, the designers have the possibility to define fine-grained access rules on each micro-room to further increase privacy. Concerning security, all external interfaces of the generated P2P application are rigorously secured automatically against malicious users and forged requests.

With the developed P2P framework a prototype of a P2P OSN is implemented which overcomes the drawbacks of all other P2P OSNs. The developed P2P OSN has a focus on collaboration and targets common interactions in secondary and high schools. Based on this e-learning environment several complex scenarios have to be developed, such as the correction of a student's homework or the borrowing of a book from the library.

By comparing both, the framework itself as well as the implemented P2P OSN, with the analyzed related work, we show the benefits of our solution. Some of them are the extensibility with arbitrary communication and collaboration features, the provision of additional privacy information, and the support of various technologies for UI generation. A final performance benchmark demonstrates the framework's practicability even in high-load scenarios.

1.3 Structure of the Thesis

The thesis is structured as follows: In chapter 2, we present related work on P2P OSNs and P2P frameworks, and derive the requirements for our work. Chapter 3 describes our approach (e.g. basic technology decisions) to meet these requirements. After that, chapter 4 outlines the architecture of the XVSM Micro-Room Framework, whereas chapter 5 contains details about

its implementation. In chapter 6, the deployment of applications developed with the XVSM Micro-Room Framework is presented. The e-learning P2P OSN and the evaluation are discussed in chapter 7. Future work, covering missing aspects of our solution, is outlined in chapter 8. Finally, chapter 9 concludes the thesis.

Related Work

In the fields of (P2P) OSNs and P2P frameworks several other contributions exist. We will outline the most promising approaches and their problems in this chapter. Based on the analyzed problems, we will derive a list of requirements for a satisfying P2P OSN.

2.1 Client-Server Online Social Networks

First of all, we will take a look at common client-server OSNs. The most popular representatives are Facebook, Google+ and Twitter. In figure 2.1, the total number of active users in millions for December 2012 is shown.

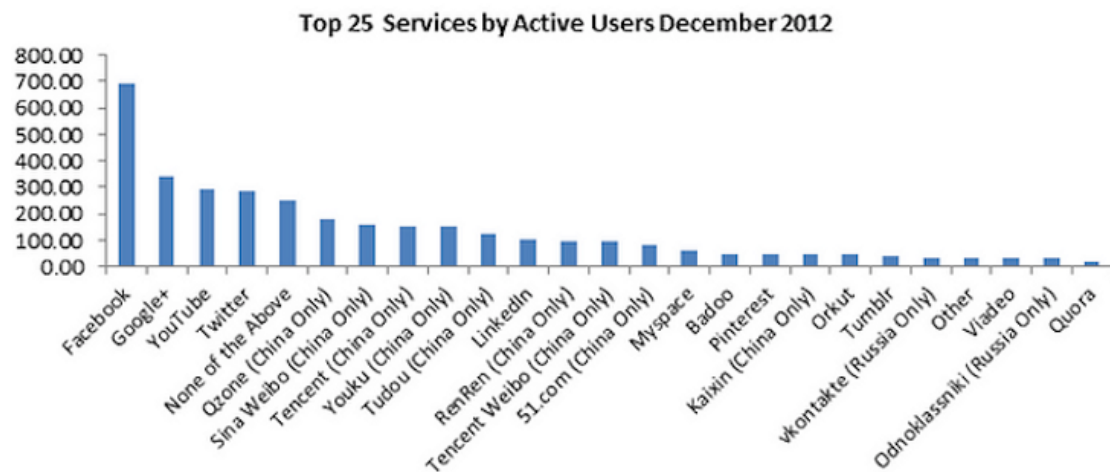


Figure 2.1: Number of active users per OSN [1]

Google+ has been introduced in 2011 by Google as an alternative to Facebook. Interestingly, even though its number of active users increased in the last few months, it is still far behind Facebook. These numbers, compared with the total count of overall online users, can be seen in figure 2.2.

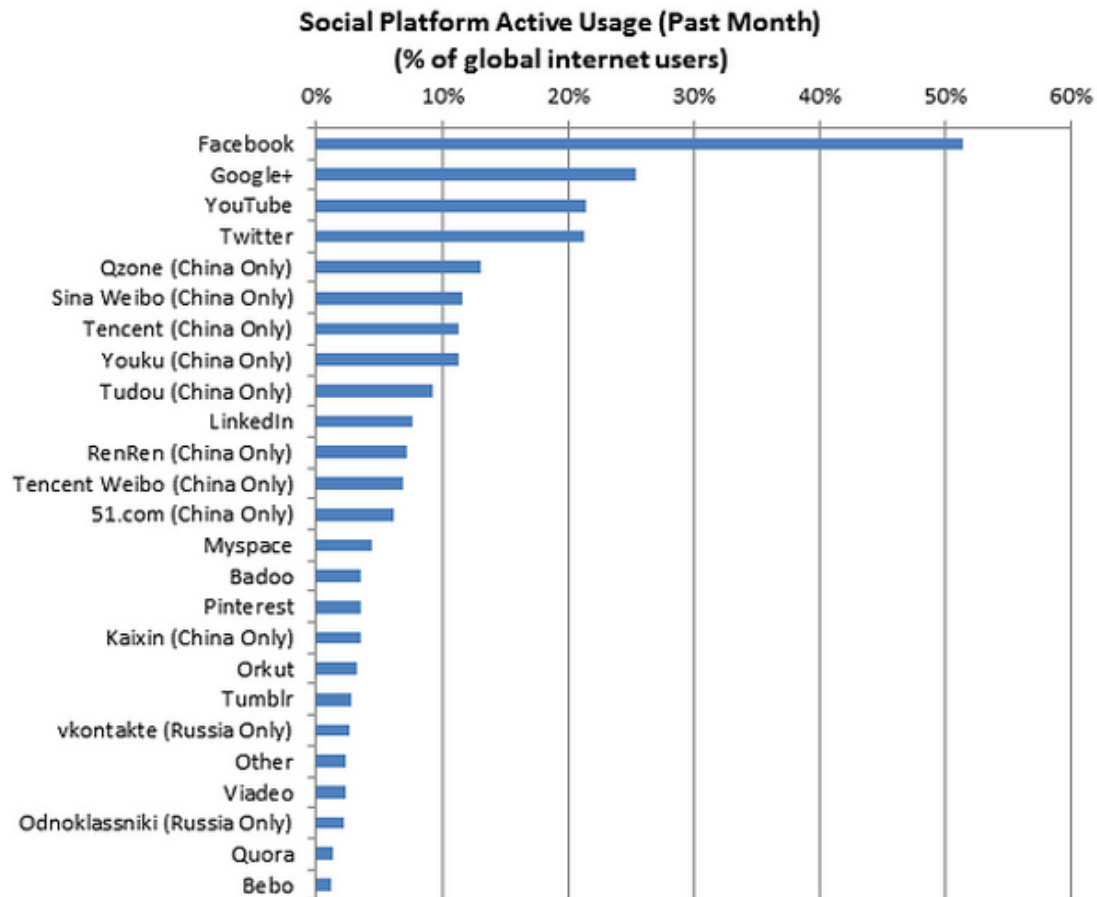


Figure 2.2: Percentage of global internet users actively using an OSN [1]

Thereby, it can be concluded that every 2nd internet user actively uses Facebook, which underlines the importance of the overall topic. As Facebook is by far the most popular client-server OSN, we will focus only on it and its problems.

2.1.1 Facebook

As the first and nowadays biggest OSN, Facebook mainly defines the features users expect from an OSN. These are mainly connecting with friends, exchanging messages, photos and videos as well as organizing events. A typical view after the login can be seen in figure 2.3.

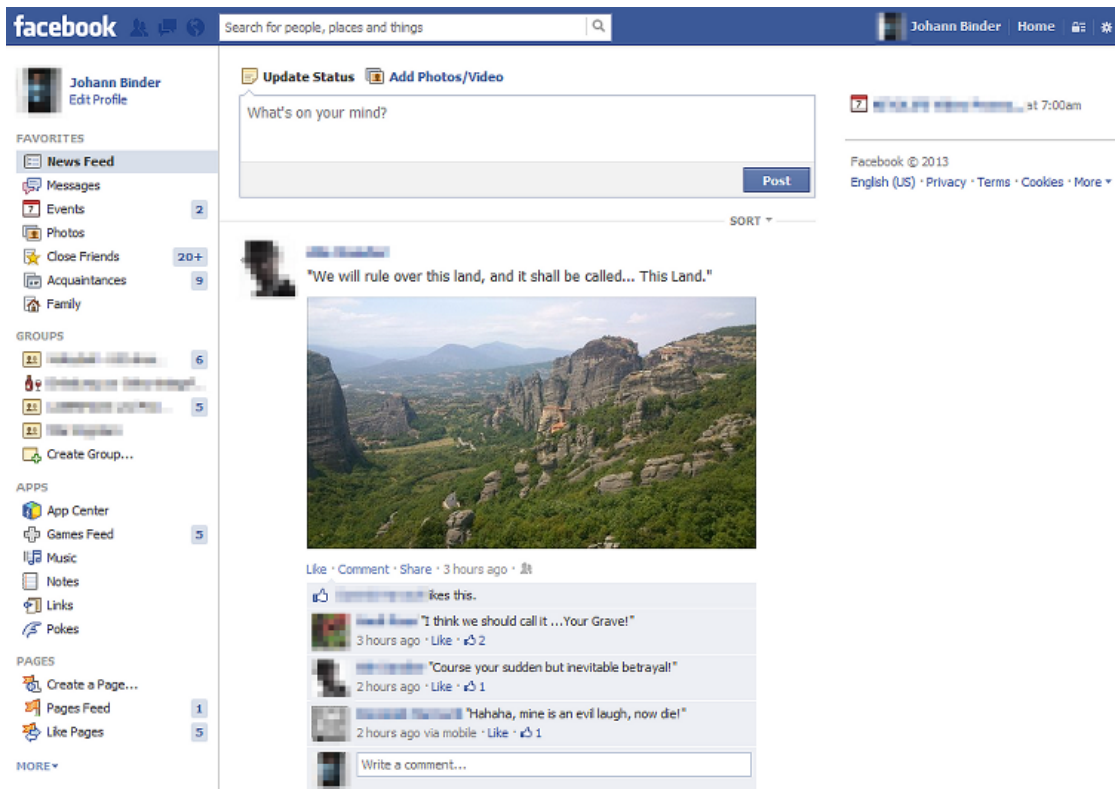


Figure 2.3: Facebook timeline

The features can be accessed via the menu on the left, upcoming events are shown on the right, and your personal news feed in the center. It consists of current messages and activities of your friends. The button for the privacy settings is unobtrusive and located in the top right corner.

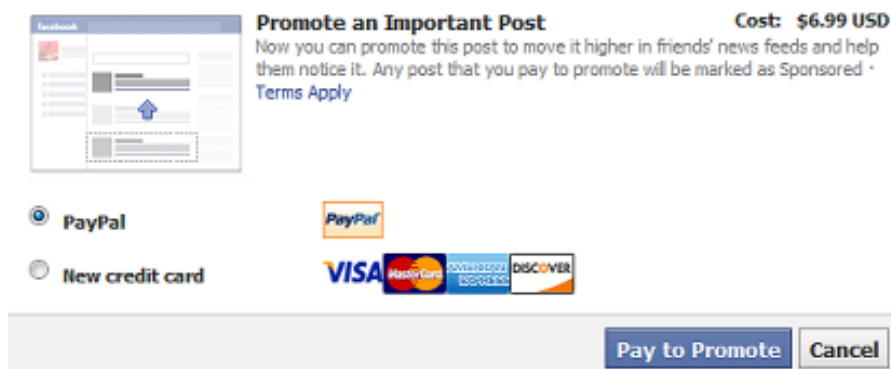


Figure 2.4: Facebook promotion feature

As all other popular OSNs, Facebook is free-to-use. However, in the last months Facebook introduced a new feature to promote own posts. This means that the desired post will be showed on top of the users' walls just as a paid advertisement in Google search. Nevertheless, this is a first try to obtain money from its users directly. But considering the expensive price of \$6.99 for promoting a single post to all of one's friends (cf. figure 2.4), its success is highly doubtful – at least regarding casual users and not advertisers.

So, neglecting this one new fee-based feature, Facebook as a profit-oriented company can only prevail by using its most valuable asset, i.e. its users' personal data. For monetizing this data it could be used for customized advertisements or sold to third parties. To get a feeling about how valuable this data actually is, here is a short overview of what a user is encouraged or even obliged to reveal about him/herself on Facebook:

- real name
- gender
- date of birth
- employer
- visited schools
- residence
- home town
- sexual orientation
- relationship status
- anniversary
- friends including their “closeness” to the user
- spoken languages
- religious views
- political views
- email address
- phone number
- visited places
- favorite movies, tv shows, music, books & sports
- other interests and performed activities

Almost all of the data listed is very problematic to disclose. Date of birth, home town, anniversary, or the favourite book are often considered as security questions for accounts. So, in combination with the person's email address one could easily compromise other accounts of this person, e.g. his/her mailbox. Information about the residence, employer, activities, and the phone number could be easily misused for stalking. Not to mention sexual orientation, religious and political views, which are considered the data most worthy to protect in the Austrian data protection act [2].

Now, all of this wouldn't be a problem if users simply didn't provide the data. But as has been shown by Gross and Acquisti in [GA05] the prevailing amount of users does so. Their findings, based on the analysis of 4,000 Carnegie Mellon University students, can be seen in figure 2.5.

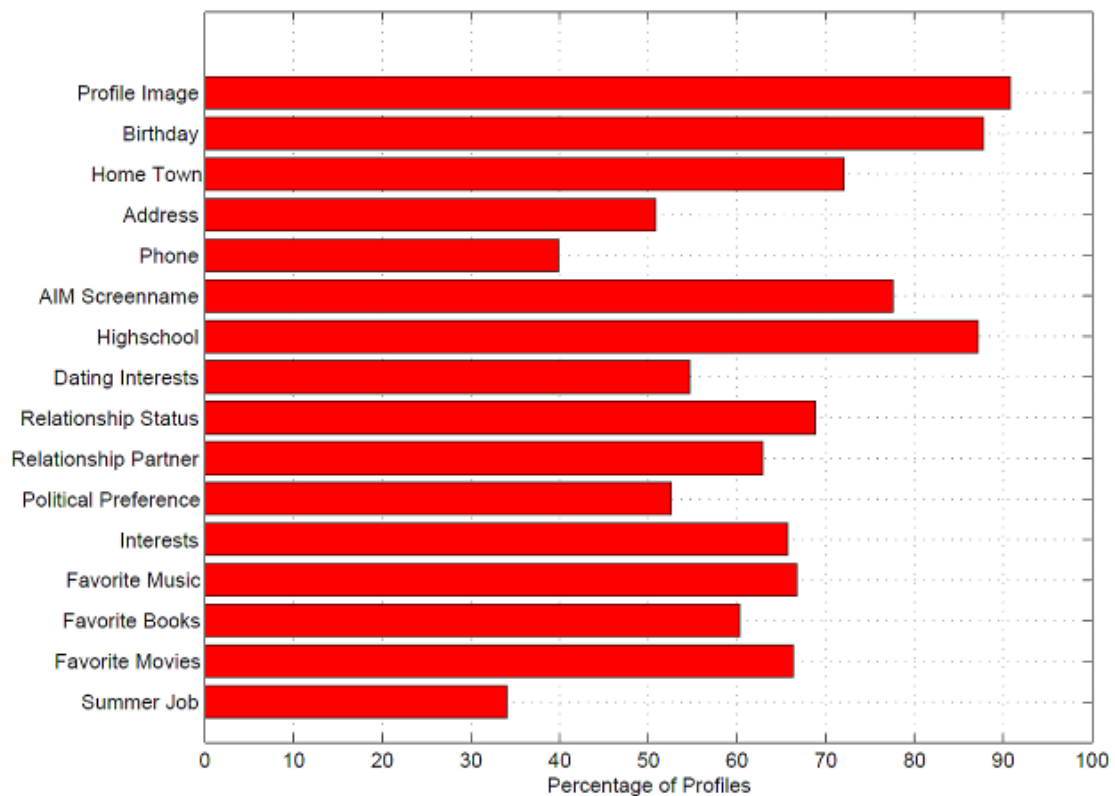


Figure 2.5: Number of personal data provided to Facebook [GA05]

As can be seen, almost 90% provide their birthday, 40% their phone number, and about 55% reveal even their political preferences and sexual orientation (= dating interests), some of the most sensitive data in the Austrian Law, as already described. Of course, one could ask whether all of this data is valid. Gross and Acquisti also covered this question on several types of data. E.g. concerning the real name, they came to the results shown in table 2.1.

Category	Percentage of Facebook Profiles
Real Name	89%
Partial Name	3%
Fake Name	8%

Table 2.1: Real names provided to Facebook [GA05]

One can assume that at least the users providing their real names also don't lie about the other data they disclose. Also, it is not very likely that users provide wrong optional data (e.g. phone number or interests) as this is senseless. The rational user wouldn't provide any data at all as this is the most convenient method. Several types of data also would have negative effects on the user if he/she forges it. An inaccurate sexual orientation or relationship status is likely to cause misunderstandings with other users.

But besides that, Facebook strictly forbids its users to provide invalid data. Section 4. *Registration and Account Security* of Facebook's terms of use includes the following two points:

- “1. You will not provide any false personal information on Facebook, or create an account for anyone other than yourself without permission.
[...]
- 7. You will keep your contact information accurate and up-to-date.” [3]

This means that one is not only obliged to provide only valid data but also to keep it (at least e-mail address, phone number, messenger id, etc.) up to date.

However, even if you are encouraged to provide valid data, you could at least hide it from others by Facebook's privacy settings. The problem is that – again – most users don't do that. Figure 2.6 shows the default privacy settings directly after profile creation.

Privacy Settings and Tools

Who can see my stuff?	Who can see your future posts?	Public	Edit
	Review all your posts and things you're tagged in		Use Activity Log
	Limit the audience for posts you've shared with friends of friends or Public?		Limit Past Posts
Who can look me up?	Who can look you up using the email address or phone number you provided?	Everyone	Edit
	Do you want other search engines to link to your timeline?	On	Edit

Figure 2.6: Privacy settings in Facebook

The most critical points are that everyone is allowed to see the user's future posts, everyone is able to find the user, and search engines like Google can directly index the user's timeline. The authors of [GA05] stated that only 0.6% of all users changed the first setting and 1.2% changed

the second one. Unfortunately, they didn't mention anything about the other settings, including who is allowed to see which data. Nevertheless, considering only these two values, one can conclude that most users really don't care about privacy settings and therefore also don't change the default values there. Of course, the referenced paper is relatively old, but newer contributions confirm the observations made.

In a more recent paper [BC09], Becker and Chen asked 105 Facebook users whether they have used any of the privacy mechanisms provided by Facebook – all of them denied. In [BRCA09], Benevenuto et al. analyzed the user behavior of more than 37,000 OSN users over a time period of twelve days. Only 0.4% of them changed something in the user settings. In their work, Krishnamurthy and Wills measured various privacy aspects of OSNs and came to the conclusion that *“users willingly provide personal information without a clear idea of who has access to it or how it might be used”* [KW08]. All of these contributions support the assumption that the findings of Gross and Acquisti are still valid today – at least to a predominant extent.

The amount of provided data alone might already be a disaster concerning privacy. But this is not all, if one looks at Facebook's data use policy, which states:

“We receive data about you whenever you interact with Facebook, such as when you look at another person's timeline, send or receive a message, search for a friend or a Page, click on, view or otherwise interact with things, use a Facebook mobile app, or purchase Facebook Credits or make other purchases through Facebook. When you post things like photos or videos on Facebook, we may receive additional related data (or metadata), such as the time, date, and place you took the photo or video. We receive data from the computer, mobile phone or other device you use to access Facebook, including when multiple users log in from the same device. This may include your IP address and other information about things like your internet service, location, the type (including identifiers) of browser you use, or the pages you visit. For example, we may get your GPS or other location information so we can tell you if any of your friends are nearby. We receive data whenever you visit a game, application, or website that uses Facebook Platform or visit a site with a Facebook feature (such as a social plugin), sometimes through cookies. This may include the date and time you visit the site; the web address, or URL, you're on; technical information about the IP address, browser and the operating system you use; and, if you are logged in to Facebook, your User ID. Sometimes we get data from our affiliates or our advertising partners, customers and other third parties that helps us (or them) deliver ads, understand online activity, and generally make Facebook better. For example, an advertiser may tell us information about you (like how you responded to an ad on Facebook or on another site) in order to measure the effectiveness of - and improve the quality of - ads.” [4]

So basically Facebook collects all data it is able to. This includes

- all actions the user performs on Facebook him/herself,
- all meta data that is included in uploaded photos and videos,

- user connections derived from the shared usage of the same machine,
- data deducible by your IP address including your real location,
- machine-relevant data including browser and operating system version,
- URL, visit date and time of all pages you open that contain just one simple *like* or *share* button and
- information about visited pages that contain ads of Facebook's advertising partners including which ads you clicked.

To sum it all up, the masses of data Facebook collects from not less than 50% of all online users (cf. figure 2.2) are alarming. In the next section we will discuss the different privacy risks and how to overcome them.

2.1.2 Privacy Risks & Solutions

In [GA05], the authors analyzed the profiles according to several privacy risks and came to the results shown in table 2.2.

Risk	Percentage of Facebook Profiles Vulnerable
Real-World Stalking	36.9%
Online Stalking	77.7%
Face Re-Identification	55.4%

Table 2.2: Number of vulnerable Facebook profiles [GA05]

Clearly, most profiles are vulnerable for online stalking, which means that the stalker can track the user's online activity, e.g. login and logout times, message histories, and pictures. For real-world stalking more information is required, e.g. real name, profile image, residence, phone number, activities and employer. That's why it is more unlikely that a profile can be used for it, even though 36.9% is still an alarming number. Face re-identification means that, by using face recognition software, one could easily link the profile with those of other OSNs or even with more unpleasant services, e.g. dating sites.

Cranor describes further privacy risks of e-commerce applications in [Cra03]. Those also affecting OSNs are unsolicited marketing, identity theft and government surveillance. Unsolicited marketing means that users receive advertisements that they might feel uncomfortable about, as they reveal private data about them (e.g. about their interests or their sexual orientation). Just as stalkers, identity thieves scan all information they can get about a user. But instead of molesting their victim, they impersonate it e.g. for social engineering. An approach, about how such identity theft attacks on OSNs can be performed automatically, has been presented in [BSBK09]. Finally, governments all over the world increase their efforts in observing their citizens in the name of war against terrorism. By providing data to an OSN, one could easily be classified automatically by grid investigation and thus might get into a very inconvenient situation. The

last point is especially concerning if you consider the current exposure of the NSA having direct access to all of Facebook's data [5].

To reduce the privacy risks described, two measures are necessary: First, the users need to be in full control of the data they provide. This is also stated in [Cra03] where the author suggests client-side profiles (among others) to reach this aim. Second, users need to be shown who is able to see which of their data, in order to raise the currently very low level of user awareness.

One general approach to counter privacy risks has been presented in [AAB⁺10] by Allard et al., where they introduced so-called *Personal Data Servers* (PDS). These are tiny, secure devices that contain all personal data of a person in an encrypted format. They provide interfaces for communication with other PDS devices and should be used among all possible services, e.g. healthcare or traveling. Considering OSNs, all profile data would rely on your PDS, and if any other user of the OSN wants to know, e.g. when your birthday is, his/her PDS would have to communicate with your PDS and ask for the data. Your PDS would then either accept the request or reject it, depending on the configurations you made. As one can see, this requires every user to have such a device in order to work. That's why even the authors title their work as a "vision paper", as the approach is not realizable in the near future.

Another approach targeting OSNs is called *None Of Your Business* (NOYB) and has been presented by Guha et al. in [GTF08]. It is a Firefox plugin that replaces own data on Facebook with random data of other NOYB users. Thereby, Facebook can't detect that data has been forged as it is puzzled together by valid data of other users. The user him/herself doesn't see any difference in his/her data or the data of his/her friends, as the NOYB plugin "decrypts" all the changes for him/her. However, the "key" (i.e. the mechanism) for decrypting a profile needs to be exchanged via a second channel, e.g. email. As this needs to be done for every profile, the whole solution is quite impracticable. Embedding the key in Facebook itself would remove this restriction but, on the same time, compromise the whole system, as Facebook could use NOYB itself to gather the real data again.

The most promising approach is to change the client-server architecture of the OSN to a P2P-based architecture. This includes most of the benefits of PDS (e.g. that users can autonomously define which data they want to provide with whom) without the hardware restriction and would lock out the OSN provider like NOYB, as all peers exchange their data directly. A lot of contributions have been made concerning this field, which are presented in the next section.

2.2 P2P Online Social Networks

In the following, we will discuss some of the most promising P2P OSNs already existing, compare them and outline missing features.

2.2.1 Diaspora

Diaspora first gained attention in April 2010 when it raised over \$200,000 on Kickstarter, which made it the most successful Kickstarter project of this time [6]. Since then, it has been the most popular P2P OSN until now, promising its users a privacy preserving alternative to Facebook.

Now, if we take a closer look at Diaspora’s architecture (cf. figure 2.7), we can see that it contains multiple servers (i.e. “Pods”) connected in a P2P-like way. The main idea of Diaspora is that theoretically any user can set up his/her own Pod and administrate it (he is then a so-called “Podmin”). Users can create accounts on every Pod available, where an account name is represented by a *Jabber Identifier* (JID), i.e. `<username>@<domain name of Pod>`. If users add new friends, their data is replicated between both affected Pods.

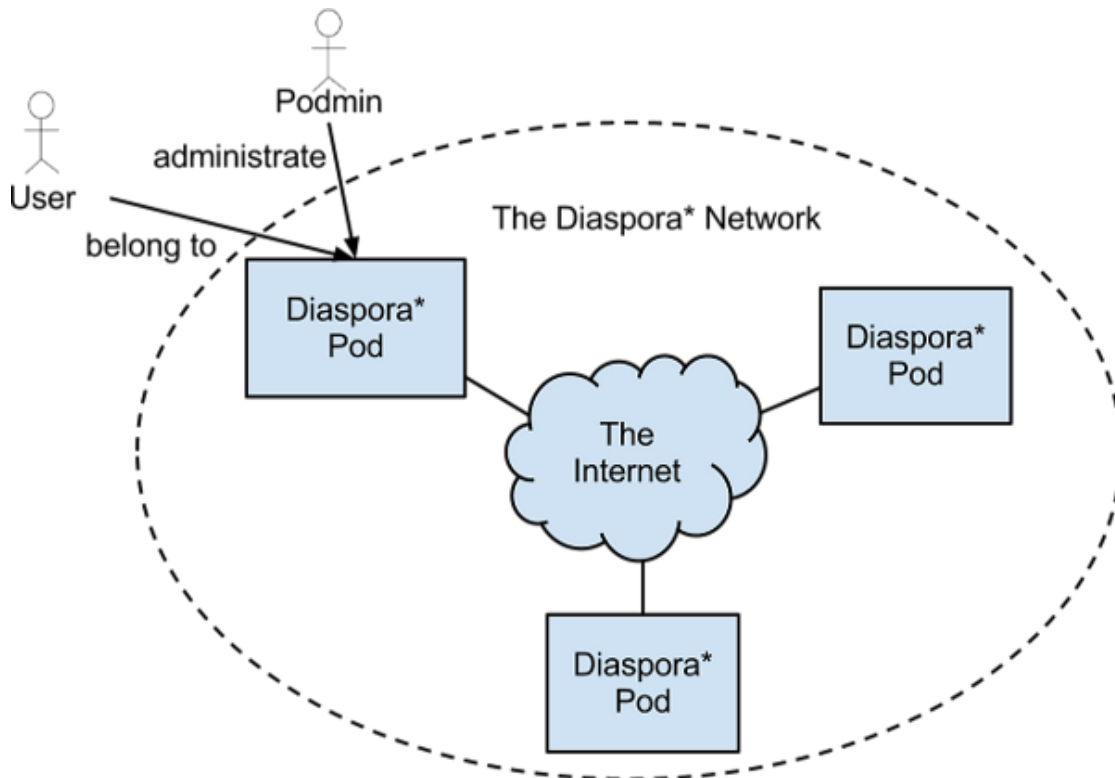


Figure 2.7: Diaspora architecture [7]

As can be seen, Diaspora only has a P2P architecture between its servers but not between its users. A user still connects to one of the servers (he can decide which one) in a client-server way and provides all of his/her data to this single instance, which doesn’t solve the privacy issues.

Theoretically, every peer could host his/her own server, but in practice this is too complicated for the common user, as Diaspora uses too many different technologies. This has already been criticized by c’t, a german IT magazine, in 2012. The author of the article stated the following:

“To install a Diaspora server one needs a Linux or Mac OS server with Ruby, RubyGems, Bundler, MySQL or PostgreSQL, SQLite 3, OpenSSL, libcurl, ImageMagick, Git and Redis. After installing these components additional manual configuration is necessary. An easy-to-install application for PC or the own web hoster? No chance!” – translated from [8]

The quotation proves true if one takes a look at the installation guides at [9]. Especially interesting is the bad support for machines running Microsoft Windows. The corresponding installation guide states:

“It is technically possible to run Diaspora on a Windows box. [...] But i don’t recommend it at this time. The configuration is not well documented and it requires advanced knowledge of both Windows and Linux environments. Future program changes could easily break it.” [10]

This means that for the majority of users (i.e. Windows users) the setup of an own Diaspora server is even more difficult than for Linux users. With that said, it is very likely that users join one of the publicly listed servers. However, there are just very few in contrast to the whole user base, which currently is about 400,000 users. By looking at the user distribution at [11], one can see that more than 200,000 users belong to Diaspora’s own hosted server. Another 100,000 users belong to two other servers, and the missing 100,000 users are distributed around 68 different servers. Only 81 users host their own server exclusively for themselves. Considering these numbers, the whole concept is reduced to absurdity.

Since the whole user base (except for 81 users) is only distributed among 71 servers, severe privacy problems are caused. Please note that the server administrators can access all data of their users, just as in client-server OSNs. This is due to the lack of end-to-end encryption – only connections between servers are encrypted. The data is decrypted at the receiving server and stored unencrypted within its database. The *Frequently Asked Questions* (FAQ) section on Diaspora’s website contains the following passage, admitting the problem:

“Communication *between* pods is always encrypted (using SSL), but the storage of data on pods is not encrypted. If they wanted to, the database administrator for your pod (usually the person running the pod) could access all your profile data and everything that you post (as is the case for most every website that stores user data). Running your own pod provides more privacy since you would control access to the database.” [12]

Nevertheless, even the suggested running of your own server would only guarantee privacy if all of your friends created their profiles on your server. As your personal data is replicated to all servers your friends belong to, it is again readable by the administrators of these servers.

“Once you are sharing with someone on another pod, any posts you share with them and a copy of your profile data are stored (cached) on their pod, and are accessible to that pod’s database administrator.” [12]

Other popular approaches like friendica¹ and buddycloud² work similarly. Both need less technologies than Diaspora (at least Apache and MySQL), but their setup is still too complicated for the average user. This is why both are not discussed in more detail in this work.

¹<http://friendica.com>

²<http://buddycloud.com>

2.2.2 Mailbook

A totally different method has been presented by Yong et al. in [YJS⁺11] and is called Mailbook. It is a P2P OSN based on free email services. The architecture can be seen in figure 2.8.

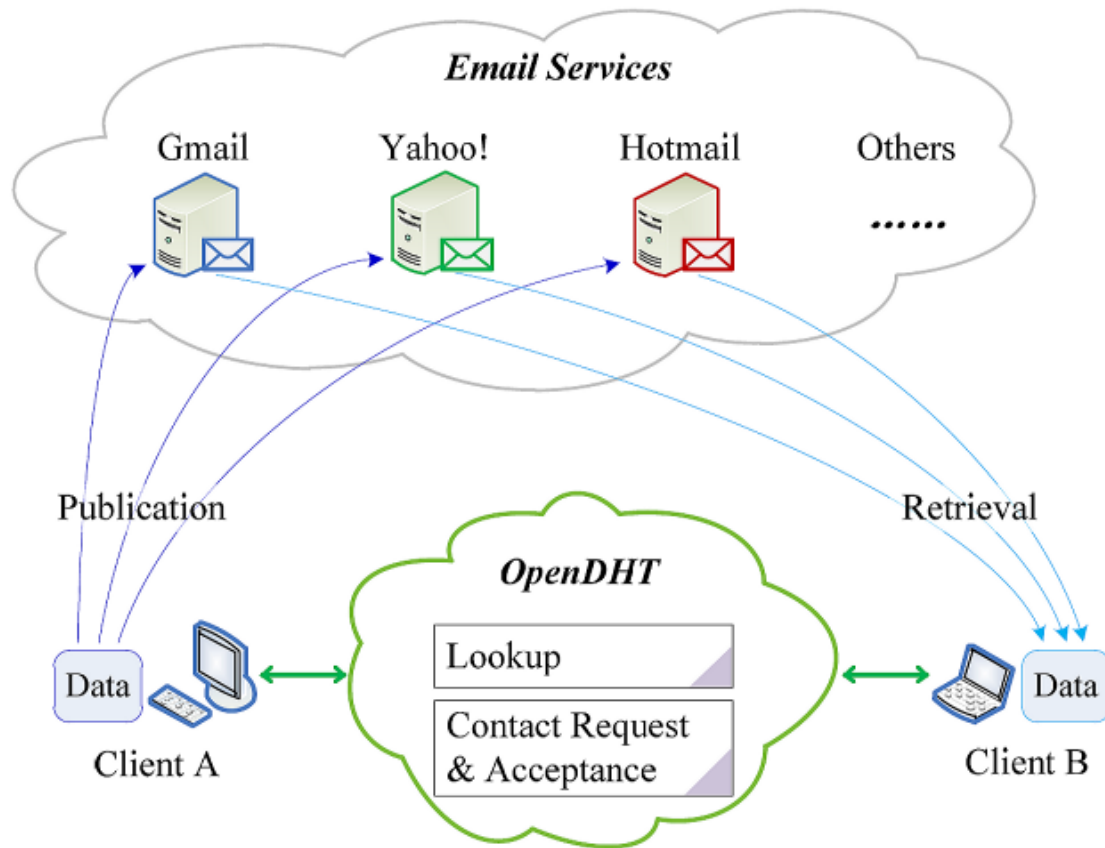


Figure 2.8: Mailbook architecture [YJS⁺11]

Clients perform their initial lookup and friendship requests via OpenDHT³, which is a public DHT service that has been hosted by Rhea et al. until July 1, 2009. [RGK⁺05]. Today, the source code of this DHT implementation is publicly available and can be used to setup an own OpenDHT service, such as done by the authors of Mailbook.

Profile or message data is encrypted and sent to other friends via email attachments. The underlying technologies used are *Simple Mail Transfer Protocol* (SMTP) for sending and *Post Office Protocol 3* (POP3) for retrieving data. Each user needs several mailboxes on different free mail providers that serve as personal stores for the data. The data is dispersed among all mailboxes of the user in order to bypass account limitations (e.g. only 500 mails per day on Gmail⁴), vendor lock-in, and to ensure data availability even if one of the providers is offline.

³<http://www.opendht.org>

⁴<http://mail.google.com>

Concerning privacy, PeerSoN relies on encryption of personal data and the control of access by appropriate key sharing. In this context, the authors assume the availability of a *Public-Key Infrastructure* (PKI) which they can access. Certainly this is a very strong assumption, as for a real OSN we would require a global PKI. However, neither such a PKI already exists, nor is yet in sight. There are certain reasons why a global PKI hasn't been created yet, like high overall complexity, poor usability, large investments needed, and questions regarding liability [LOP05]. Also, some kind of chicken-and-egg problem exists, since at the moment there are almost no applications where customers really need their own key pair. That's why most of the users simply don't have such a key pair. But if only few users have an own key pair, there is also no incentive for companies to build applications, requiring clients to authenticate with keys rather than passwords, because too many potential customers would be locked out. So, basically one could consider the current global PKI to be "*in a sorry state*" [HBKC11]. Therefore, the assumption of an available PKI simply doesn't hold in case of a global OSN. This means that either the whole encryption can't be realized, i.e. privacy isn't ensured at all, or that the keys need to be managed and shared by the users themselves, i.e. the efforts for setup are very high.

As can be seen in figure 2.9, PeerSoN stores an undeliverable message (e.g. if the target peer is offline) in the OpenDHT to ensure availability and consistency of this data. However, this violates the two-tiered architecture, since private data is now stored in the lookup service.

Another drawback of the approach is that the whole data storage on the peers is file-based which is slow, compared to other methods such as key-value stores or *Relational Database Management Systems* (RDBMS).

2.2.4 Persona

Baden et al. present another P2P OSN in [BBS⁺09] that they call Persona. As in PeerSoN, the storage is not trusted, which means that privacy of own data, distributed to other peers, needs to be ensured via encryption. Please note that the distribution to other peers is required for a decent level of data availability.

In contrast to PeerSoN, Persona uses a more complex encryption approach called *Attribute-Based Encryption* (ABE). This can be used to implement group-based encryption which is important to allow e.g. different visibilities for profile data or messages (e.g. to distinguish between family members and co-workers). It works by generating a unique ABE secret key for every friend, containing the groups that he/she should be part of. All users are able to retrieve all data from all other users, but only those belonging to the groups, defined by the data owner, will be able to decrypt it. Persona uses its own Firefox plugin that implements a secure key store to which users upload their private and public keys. Thereby, the effort required by users for setup and key management is reduced in contrast to PeerSoN (assuming that no PKI is available), but still not practicable for the common user.

Nevertheless, this solution also suffers from some drawbacks. One of them is that the encryption itself is computationally expensive due to the many keys involved. Another problem is that data once replicated to other peers isn't deleted in a confirmed way anymore. User access can be revoked via re-keying, though the data is spread across many peers and not "cleaned up" anymore. Thus, if a vulnerability in the used encryption is discovered in the future, all peers still holding these data could decrypt it even without the new keys. Other approaches force deletion

on all peers, therefore preventing future access to the data – at least if none of the peers made a private backup.

2.2.5 Safebook

Safebook [CMO11] is a P2P OSN concentrating on security and privacy. The peer software is implemented in Python and thus can be executed on all common operating systems. The front-end is created with PHP and MySQL, thereby generating a Facebook-like user experience (cf. figure 2.10).



Figure 2.10: SafeBook user interface [CMO11]

Safebook relies on so-called “matryoshkas” that consist of a set of nodes grouped in concentric rings. The rings represent the level of trust that the nodes on it have with respect to the user, where the innermost ring is the most trusted layer consisting of friends. To ensure availability, data is only replicated to these trusted nodes, also called “mirrors”. Via multi-hop routing in the matryoshka, Safebook can guarantee anonymity. More about this technology is beyond scope of this work and can be read in [CMS09b].

Problems of the approach are first of all the additional effort to prove your identity in order to get into the trusted network. Therefore, face-to-face meetings or other schemes, such as passports or id cards, are required [CMS09a]. Additionally, no fine-grained visibility control is possible – messages are always replicated to all friends.

2.2.6 LifeSocial

The last approach we will discuss in this thesis is LifeSocial [GG⁺11]. It uses Pastry [RD01], a distributed object location and routing framework, for the underlying P2P communication. Compared to the other P2P OSNs, it can be considered to be developed the furthest. The authors describe it more as a “P2P Framework for Social Networks” [13], since it is highly extensible via plugins. Some already developed plugins (including core features) are:

- Profiles
- Messaging
- Photo albums
- Groups
- File transfer
- Whiteboard
- Calendar

Hence, the solution already covers a broad set of features, including collaboration tools. Similar to Persona, it uses multiple keys to encrypt data, so it can be decrypted only by peers allowed by the owner. A screenshot of a profile in LifeSocial is shown in figure 2.11.

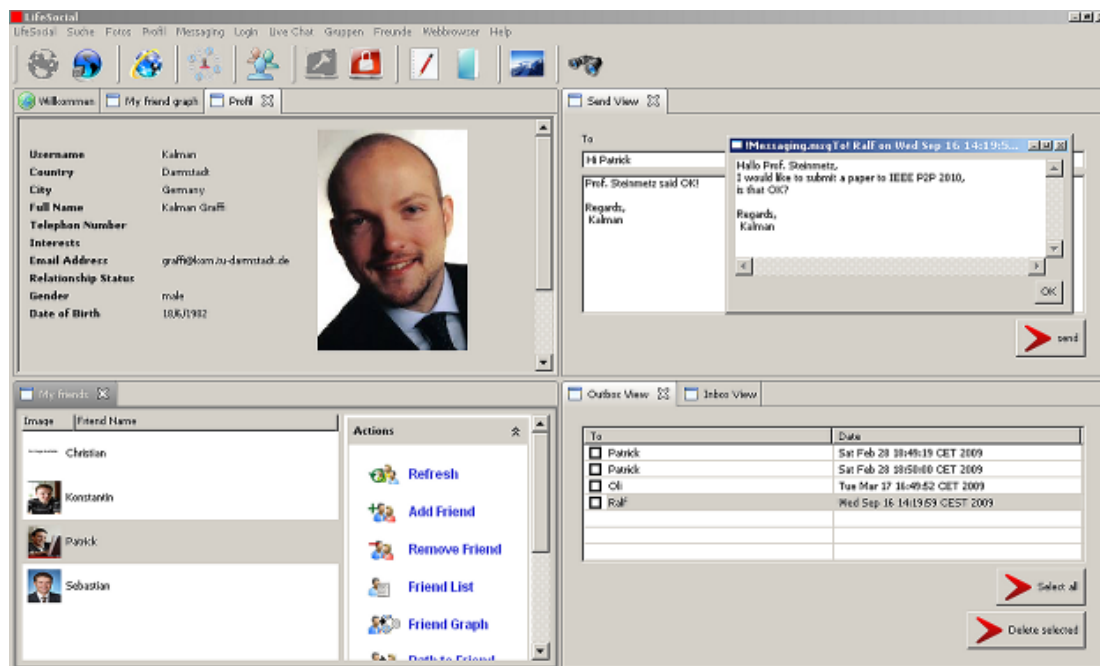


Figure 2.11: LifeSocial user interface [GG⁺11]

As can be seen, the solution's usability is not as good as those of the web-based OSNs, like Facebook, Diaspora or Safebook. Users are familiar with the concepts predetermined by Facebook, which is likely to cause problems with user acceptance of LifeSocial's *User Interface* (UI). Another drawback is that fine-grained access restrictions are not possible for several features, such as posts on your own wall or the data provided in your profile.

2.2.7 Comparison

In this section we will compare the previously presented P2P OSNs. Therefore, we introduce a list of criteria that is required for a proper P2P OSN. Some of these criteria have been presented in [BD09], e.g. availability, extensibility, and security. Others have been defined by us, based on our observations when analysing the approaches. The comparison results can be seen in figure 2.12.

		Facebook	Diaspora	Mailbook	PeerSoN	Persona	Safebook	LifeSocial
Functionality	Communication	✓	✓	✓	✓	✓	✓	✓
	Collaboration	✗	✗	⚠	✗	✓	✗	✓
	Availability	✓	✓	✓	⚠	✓	✓	✓
Simplicity	Familiar UI	✓	✓	✗	⚠	✗	✓	✗
	Easy deployment	✓	✗	⚠	✗	⚠	⚠	✓
Extensibility	Plugin/Module support	✓	✗	✗	✗	✓	✗	✓
Privacy	Protection against data abuse	✗	✗	✓	✓	✓	✓	✓
	Fine-grained access restrictions	✓	⚠	⚠	✗	✓	✗	⚠
	Assured deletion	✗	✓	✗	✗	⚠	⚠	✓
	Knowledge about physical data distribution	⚠	⚠	✗	⚠	✗	⚠	✗
Security	Protection against manipulated clients/requests	✓	⚠	⚠	✓	✓	✓	✓

Figure 2.12: Comparison of P2P OSNs

The meanings of all criteria and details about the comparison results are presented below:

Communication: A basic feature of all OSNs is the support of several ways of communication. Some of these are private chats, posting messages on other users' walls, or message exchange in closed and public groups. All of the evaluated approaches support this key feature in a sufficient way.

Collaboration: Collaboration features (e.g. file exchange, polls, calendars, ...) on the other hand, are not that widely supported. This is because Facebook itself (neglecting all of its third-party apps) does not provide these features and, therefore, the alternatives also neglect them. However, collaboration is also important for private users, e.g. to arrange appointments or to exchange the newest holiday pictures without making them available to third parties. Further examples can be seen in specialized collaboration tools, such as Microsoft Groove⁵ or Google Wave, which has unfortunately been discontinued [14].

Of all the presented OSNs, only LifeSocial and Persona provide such collaboration features in an adequate way. In LifeSocial these are a whiteboard for common writing, polls, file exchange and calendars. Persona provides a special “Doc” application which can be used as a basis for collaboration features. Mailbook at least allows file exchange, which is inherently supported by its email-based architecture.

Availability: In client-server OSNs, data availability is no problem, since the server is always online. However, in P2P-based approaches, data availability stands in direct contrast to privacy. If all private data is located only on the peer and the peer goes offline, the OSN gets unusable for the other members. E.g. the profile picture, wall, and messages from the user are not available anymore. Therefore, his/her data needs to be replicated, but this means possible threats of data abuse by the peers it’s replicated to.

Facebook and Diaspora are server-based, i.e. they have no availability problem as the servers are always running and replicated (at least in case of Facebook). All the other alternatives use data encryption or intelligent data distribution to ensure availability, while avoiding privacy issues. Only PeerSoN struggles with a problem concerning availability, since it uses the DHT to store undeliverable messages. This works and also other approaches act in a similar way. However, PeerSoN strictly breaks its architecture, as the DHT is intended to be used as a lookup service only. Therefore, the solution can be considered as a “hack”, not allowing it to be evaluated as good as the other approaches.

Familiar UI: Another important criteria is the user interface. Users are familiar with the interaction concepts of Facebook and don’t want to become acquainted with new UIs. This is a crucial point for user acceptance.

Only Diaspora and Safebook use web-based UIs that are inspired by Facebook. The application or console UIs of Mailbook, Persona and LifeSocial don’t satisfy this criteria and are highly customized solutions. Concerning PeerSoN it is unknown whether it supports web-based UIs, as the authors of [BSVD09] don’t state anything specific about their implemented prototype (e.g. program languages or technologies used).

Easy deployment: Besides the UI, it is important that the proposed solutions are also easy to deploy for the user. Considering Facebook, he/she only needs to register an account and is ready to go. In P2P-based solutions, this very low setup effort cannot be achieved due to the architecturally inherent need of a local client software. Nevertheless, the deployment effort required should be as low as possible.

⁵<http://grv.microsoft.com/>

Concerning Diaspora, one would need to install several different technologies, which is not very user-friendly (cf. section 2.2.1). In Mailbook, the user needs to create multiple accounts at several freemail providers and download the Mailbook client. This is still time-consuming, but not as much as with Diaspora. As already stated, nothing is known about the technologies used for PeerSoN. However, the authors of [BSVD09] assume an available PKI, which is simply not the case on a global basis. The effort to setup and manage such a PKI is unacceptable for common users and has to be considered when evaluating this solution. In Persona, the user has to install the Firefox plugin and download the Persona application. Also, he/she has to generate and import his/her private and public key to the Firefox plugin. Especially the key generation is very unintuitive for common users, which complicates the deployment of this approach. On the other hand, the overall key management is performed by the Firefox plugin, therefore making the handling easier in contrast to PeerSoN. Considering Safebook, one needs to download the client and verify one's identity to the trusted identification system. This could be done via face-to-face meetings or the entering of passport ids, thereby generating an additional amount of work for the user, in contrast to Facebook. Finally, LifeSocial includes everything within its client. Users only need to download it and are ready to go, as also the UI is integrated in the client itself.

Plugin/module support: As an OSN is likely to evolve with time, it is necessary to allow adding new features easily. This can be achieved by plugin/module support, howsoever one calls it. Thereby, features can be added later on by just adding a new plugin/module without changing the core implementation of the OSN.

Facebook allows this for anyone by its *Graph Application Programming Interface (API)*⁶. Persona and LifeSocial both have a modular architecture, allowing to add further functionality by additional applications (Persona) or plugins (LifeSocial). Concerning Diaspora, Mailbook, PeerSoN and Safebook, nothing implies that they support modules or plugins.

Protection against data abuse: The main target of all alternative approaches is to protect the user's data against abuse by third parties. In client-server OSNs such as Facebook this would only be possible with end-to-end encryption.

However, due to its server-like architecture and the lack of end-to-end encryption, also Diaspora's server administrators have full access to all data of their users. One could consider this to be even worse than on Facebook, as Facebook risks at least its reputation if someone gets scent of data abuse – in contrast to an anonymous Diaspora Podmin. Mailbook, Persona, PeerSoN and LifeSocial on the other hand use a pure P2P architecture. Even though data could also be replicated to untrusted peers, their full data encryption makes it impossible for unwanted users to read the data. Safebook only replicates data to trusted friends that can read the data anyhow on the user's profile, thereby avoiding the risk of data abuse.

Fine-grained access restrictions: Besides the protection against access of third parties, users actually like to share their data (e.g. birthdays, messages and pictures) with other users.

⁶<http://developers.facebook.com/docs/reference/api/>

To achieve this, the OSN needs to provide fine-grained, configurable access restrictions for each data record.

Facebook allows group-based visibility settings for all data. In Diaspora, messages on your own wall can be configured independently, but single profile data sets not. Considering Mailbook, nothing has been mentioned about fine-grained access restrictions. However, it can be assumed that the receivers of messages and profile data could be configured via specifying email addresses one by one or by email distribution lists. PeerSoN and Safebook on the other hand only distinguish between friends and non-friends, i.e. no custom groups are possible. LifeSocial basically implements group-based configurable access restrictions, but they are not supported for profile attributes and posts onto your wall. Due to its ABE support, Persona allows access restrictions for every data record.

Assured deletion: A big problem of client-server OSNs is that the user has no real control of the data. So, if a user deletes a data record, he/she can't be sure that it is actually deleted from the server.

Recent incidents [15] have shown that Facebook doesn't delete data but only marks it as deleted. Diaspora guarantees deletion by a special retraction request that is replicated to all Diaspora servers containing personal data of the user. In Mailbook, deletion is technically impossible, as emails can't be deleted from the outside anymore, once they are delivered. LifeSocial ensures deletion of data by overwriting it with an empty tombstone object that is replicated like normal data. Considering PeerSoN, the assumed PKI would allow to retract keys and thereby make it impossible for former friends to read personal data anymore. However, as already stated, this is not part of the solution itself and thus can't be taken into account for the evaluation. Persona allows re-keying to revoke read access from former friends. Nevertheless, strictly speaking, the data is not deleted from their machines. The authors of Safebook don't state anything about deletion, but it can be assumed that they replicate delete request equally to normal requests. Of course, deletion can only be assured for data managed by the peer client software itself. If a user copied data and stored it in another location, it would be technically impossible to delete it automatically.

Knowledge about physical data distribution: This is a criterion on that we will focus on. It means that users need to be informed about on which machines their data is replicated to, which is especially important for P2P-based OSNs. We assume that thereby their awareness regarding privacy can be increased, as they might get shocked about how many other persons actually possess sensitive data about them.

Since Facebook itself is client-server-based, users basically know that their data only relies on the servers of Facebook. However, due to re-occurring incidents in the past, trust has faded, thereby raising the question whether data is distributed to third parties or not. Users of Diaspora also implicitly know to which servers their data is replicated to, if they look at the JIDs of all their friends. Nevertheless, this is not very user-friendly. As Mailbook separates its messages into different emails and sends them to several different email addresses, it is not possible for the common user to get information about the actual storage locations of their personal data. The data of PeerSoN and Safebook is replicated to

all friends, thereby allowing users to implicitly know their data's locations – though this is not visualized in any form. Persona theoretically allows anybody to get the (encrypted) data and store it locally, which makes it impossible for users to track where their data is. In LifeSocial, data is replicated to other peers in a way not known by the user.

Protection against manipulated clients/requests: Finally, all the previous criteria regarding privacy are worthless if the solution is vulnerable to manipulated clients or requests. Malicious peers could modify the client software or requests to circumvent restrictions implemented in the client.

Due to their client-server nature, this is no problem for Facebook and Diaspora, as all validations and restrictions are implemented on the server side, not manipulable by the user. However, in Diaspora, peers (i.e. Podmins) could manipulate their servers and try to retrieve data from other servers. Mailbook suffers from its email-based architecture which is very vulnerable to attacks. Regarding PeerSoN, Persona, and LifeSocial, unauthorized data access is impossible, thanks to the used encryption. Safebook ensures security via its matryoshka-based architecture.

As can be seen, every approach has certain drawbacks as they all focus on different aspects. Interestingly, all of them have been built up from scratch (neglecting P2P overlay networks such as OpenDHT or Pastry), even though most of the criteria listed above could be provided by a generic P2P framework. Availability, easy deployment, plugin support, protection against data abuse, fine-grained access restrictions, assured deletion and protection against manipulated clients are relevant for most P2P applications, not only for P2P OSNs. However, the reason why all of the P2P OSNs presented have been developed from scratch, is that only very few of such frameworks exist, suffering from drawbacks themselves. In the next section we will discuss the most promising approaches.

2.3 P2P Frameworks

Besides OpenDHT and Pastry, JXTA⁷ and AntHill [BMM02] are some of the frameworks available to reduce the developers effort when developing P2P applications. What they all have in common, is that they mostly deal with abstracting the communication layer of P2P networks and provide an interface for developers to build an application on top of it. The only two approaches found that decrease the developers work load even more are presented in the following.

2.3.1 TOMSCOP

Technology of Multi-user Synchronous Collaboration Platform (TOMSCOP) [KM04] acts as a P2P collaboration platform that eases the development of shared applications. To handle the underlying P2P communication it uses JXTA as a basis. It is interesting to note that in TOMSCOP, each peer can administrate collaborative rooms which represent groups. Via these rooms, group-based communication and collaboration can be implemented.

⁷<http://jxta.kenai.com>

Figure 2.13 shows the various communication channels supported by TOMSCOP. Peers can either create a room and communicate with other peers in this room, with all other peers in the global community room, or only with one peer in a private chat.

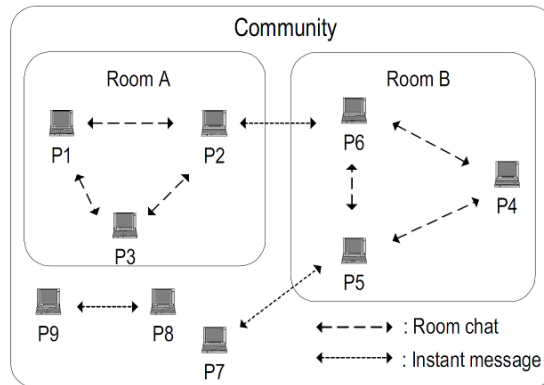


Figure 2.13: TOMSCOP communication channels

In figure 2.14 you can see an example application developed with TOMSCOP. The application is a simple shared web browser that is used by the peers `tom-labo` and `tom-note` within the room `Ma-lab`. If one peer enters a new *Uniform Resource Locator* (URL), the website is loaded at all peers within the room.

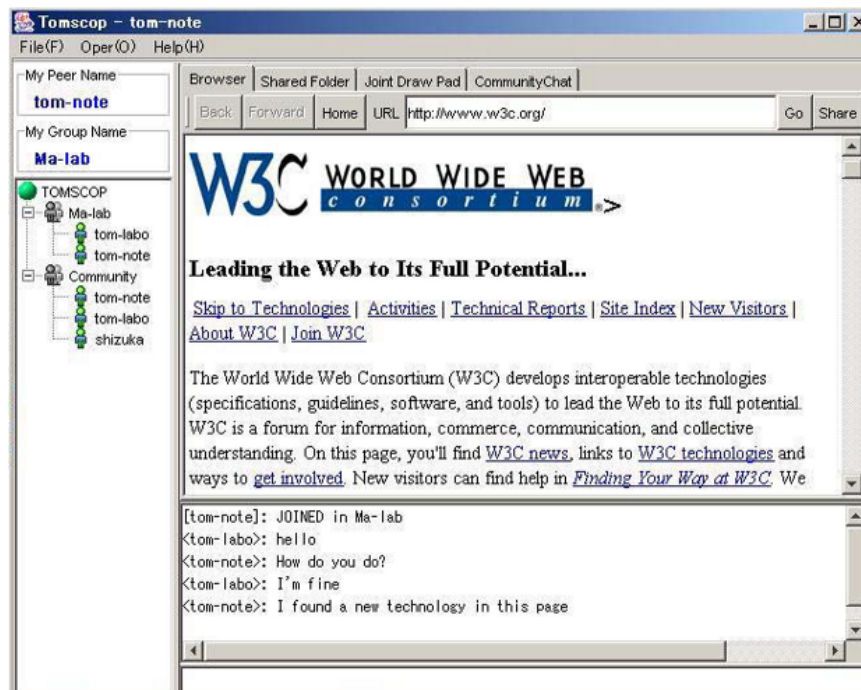


Figure 2.14: TOMSCOP shared web browser example

Even though this room-based approach is very promising, the development of TOMSCOP seems to be discontinued. Besides that, it does neither target the privacy issues we have pointed out above, nor allow creating a Facebook-like UI. Hence, it is not a perfectly suiting candidate for developing P2P OSNs.

2.3.2 The P2P Application Framework

“The P2P Application Framework” – as it is called by the authors of [WHR⁺08] – is also a further developed P2P framework. As TOMSCOP, it uses JXTA for P2P communication. The framework is written in Java and uses reflection to integrate plugins. These are stored in a special directory and loaded on framework initialization.

The initial implementation of the framework requires an index peer that runs a MySQL database and captures all details about the state of the network, e.g. available resources or CPU performance. A typical peer network used by the framework can be seen in figure 2.15.

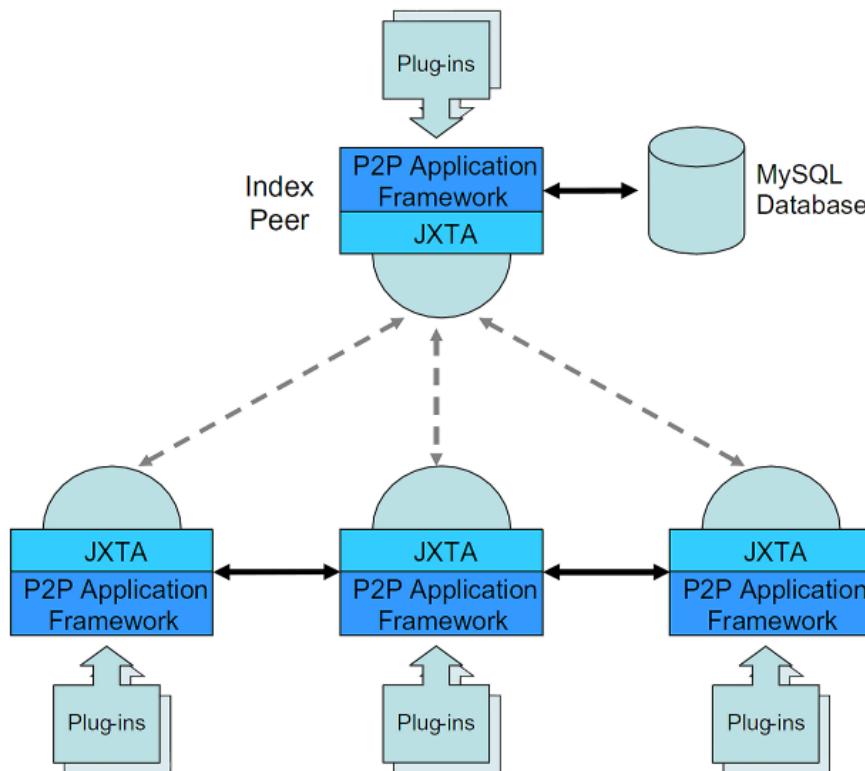


Figure 2.15: The P2P Application Framework peer network

An application based on the framework is shown in figure 2.16. It represents a digital library in which peers can add and delete papers. As can be seen, the application is added as a plugin to the P2P Application Framework and can be used immediately between all users having this plugin.

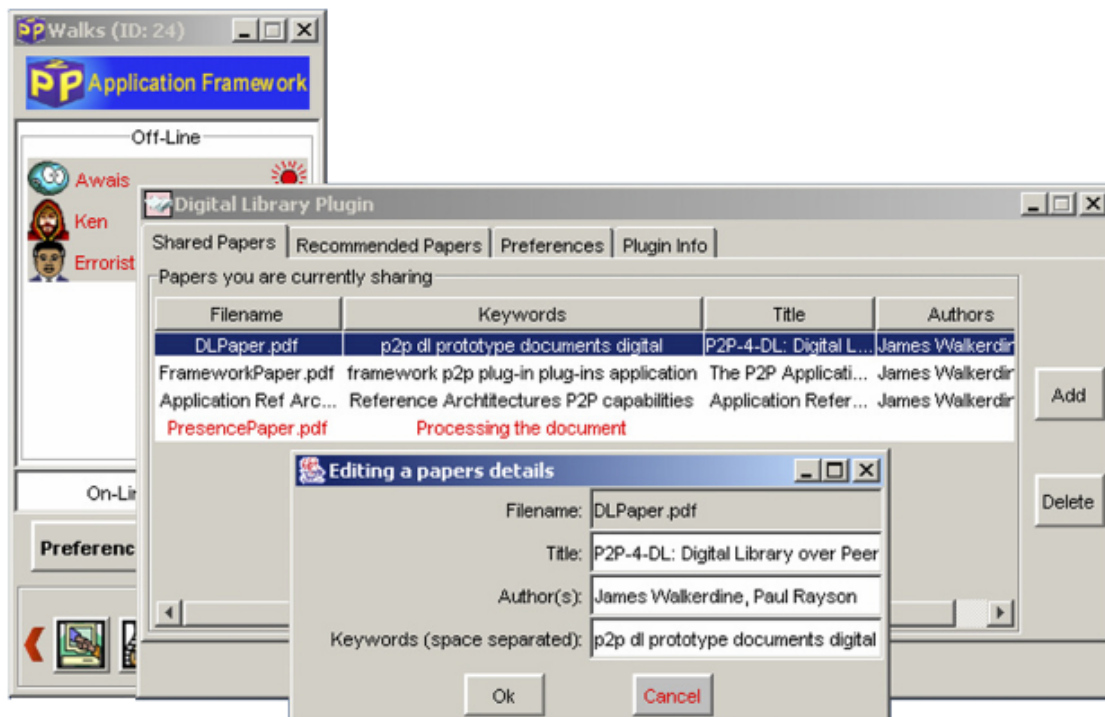


Figure 2.16: The P2P Application Framework digital library example

Unfortunately, also this approach is unsuitable for P2P OSNs. First of all, the index peer can't be realized as every user frequently goes on- and offline. Second, the P2P Application Framework doesn't consider privacy issues and provides no easy possibility to generate a Facebook-like UI due to its Java Swing⁸ UI. Additionally, it seems to be discontinued as well.

2.4 Conclusions

As we have seen, none of the current P2P OSNs satisfies all of the criteria listed in section 2.2.7. This could be different if a P2P framework existed that covers a decent part of these criteria itself. However, as has been outlined in the previous section, this is not the case. Hauswirth et al. came to a similar conclusion:

“We are convinced that P2P-based collaboration is a paradigm that meets the requirements of users, but there is still a lack of enabling technologies which have to be researched and implemented.” [HPD05]

Therefore, we will create a fitting P2P framework on our own and implement a P2P OSN as a proof-of-concept with this framework. The requirements of the framework can be derived directly from the criteria of section 2.2.7 and are presented in the following:

⁸<http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>

Functionality: The framework needs to support several ways of communication and collaboration. Also, it has to provide a decent level of data availability, which is a special problem in P2P-based applications.

Simplicity: First of all, it is important to use as few technologies as possible (in contrast to e.g. Diaspora), since this significantly reduces the overall framework complexity. Second, the framework should enable the developers to become acquainted to it easily and to deploy the generated application in a simple way. So, neither complex, nor a lot of tasks should have to be performed by the user to get started. Finally, the framework has to allow the developers to create their UI without limitations, in contrast to the other solutions presented (e.g. LifeSocial, TOMSCOP and the P2P Application Framework).

Extensibility: A P2P framework that is not extensible is useless. Therefore, the framework to be implemented needs to support plugins or modules with which additional functionality can be added. Additionally, these plugins or modules should use open formats (e.g. XML) to keep the system as easy as possible (cf. Simplicity).

Privacy: As a central point in this work, privacy also plays an important part in the new P2P framework. Hence, the framework has to protect every peer's data against abuse by third parties, allow fine-grained (group-based) access restrictions for each data record, guarantee deletion, and provide a mechanism for users to show where their data actually is located at.

Security: Finally, the framework needs to ensure a decent level of security, i.e. it has to prevent attacks by malicious users that might manipulate the peer client software or requests to other peers.

Approach

We will now present the concepts and technologies used by our approach. Afterwards, we will describe how they contribute to fulfill each of the requirements defined in the previous section.

3.1 Micro-Room Concept

The creation of an application with the XVSM Micro-Room Framework is a three-step task. First, programmers create the desired functionality via implementing modules and adding them to the framework. Then, these modules have to be configured and orchestrated to a workflow via so-called “business logic” files. Finally, the UI designer can create the UI based on the workflow. Therefore, he/she can use the framework’s *Representational State Transfer* (REST) API to interact with the orchestrated modules.

Strictly speaking, modules mean either micro-rooms or plugins in this case. Micro-rooms are a new concept of how to abstract functionality in a way that is understandable also for users without IT background. It can be compared with the room concept of TOMSCOP (cf. section 2.3.1), but is far more flexible. We will describe it with help of a practical example, i.e. the library shown in figure 3.1.

Each micro-room basically consists of a unique name, a set of actions and a set of plugins. Also, it can be configured by adjusting room-specific settings. The actions define what users (i.e. peers) can do within this room and are directly callable from the UI. In the example, a member of the `Library` could, e.g. borrow or return books. All of the actions can be implemented (without limitations) by the module developer. Thus, arbitrary functionality can be added to the framework. Actions can be connected to each other, thereby allowing to create workflows between different rooms.

Besides that, even the functionality of a micro-room itself can be extended, which is done via plugins. Each room needs to be configured at least with the `Replication` and the `Access` plugin. The `Replication` plugin allows to configure whom this room should be replicated to, whereas the `Access` plugin allows to configure who is permitted to execute which action.

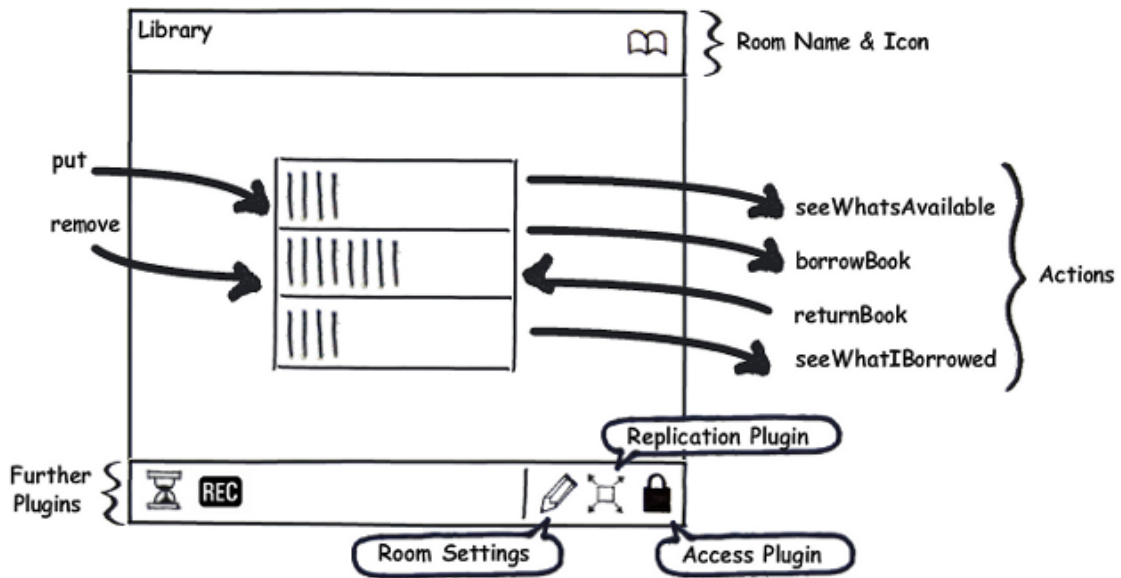


Figure 3.1: Micro-room example

Other plugins can be created by module developers without restrictions and could, e.g. allow the modification of access restrictions during runtime or the event-based execution of actions.

Our concept provides several advantages over traditional approaches. First of all, it is easy to understand and use. The structure of module and business logic files is held simple and in well-known technologies (cf. sections 3.2.2 and 3.2.3). Second, it separates the creation of complex P2P applications cleanly into three different ranges of duty. The functionality is implemented by module developers, the UI by UI designers and the overall workflow of the application can be configured e.g. by managers or affected users themselves, as they only need to understand the micro-room concept – nothing else. Third, extending the framework's and thus the P2P application's capabilities can be achieved very easily by adding new micro-rooms or plugins. Finally, the XVSM Micro-Room Framework allows to speed up development time of P2P applications due to its separation of concerns and the decreased workload for developers.

3.2 Technologies Used

In this chapter we present the technologies used by the XVSM Micro-Room Framework and the implemented P2P OSN.

3.2.1 XVSM

We use many of the capabilities of XVSM to avoid building up our framework completely from scratch. XVSM is a space-based middleware that is developed by the Space Based Computing Group of the Institute of Computer Languages at the Vienna University of Technology.

Mozartspaces¹ is the Java implementation of XVSM and – more precisely – used by the XVSM Micro-Room Framework as it is also Java-based. The current stable version 2.2 can be downloaded and used by the terms of the GNU Affero General Public License (AGPL) Version 3².

XVSM is based on the Linda model that has been presented by Gelernter in [Gel85]. The basic idea is to allow the communication between different processes without explicitly addressing them via process *identifiers* (ID). Therefore, all processes have access to a shared space, where they can put data in form of tuples (tuple space). This results in a decoupling of processes regarding both space and time, as process A can write data to the space and process B will still be able to read the data if B goes offline.

This approach for processes on the same machine has been extended by XVSM to distributed P2P environments. Figure 3.2 shows how the communication between several peers works in XVSM.

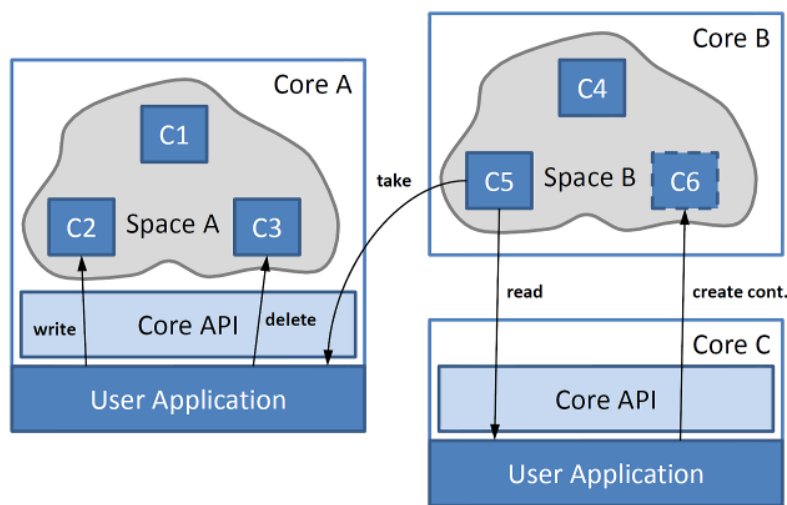


Figure 3.2: XVSM communication [Dö11]

Each core represents one instance of XVSM running on an arbitrary peer and providing a part of the shared space. The peer application can access the space via a defined API. It is important to note that the access is not restricted to the “own” part of the shared space but is possible also for spaces running on other peers, thereby creating a distributed shared space.

C1 to C6 represent containers, which are used to encapsulate data. Hence, each container holds a set of entries that can be accessed in multiple ways. Entries can be of any type that implements the `Serializable` interface in terms of Java. In figure 3.3 you can see the internal representation of such a container.

Each entry is stored only once, but on each container multiple so-called “coordinators” can be defined. Some of them are shown in the figure, i.e. the `Random`, `First In – First Out (FIFO)` and the `Key Coordinator`. These coordinators store additional meta data for the entries in order to access them in different ways. E.g. the `Random Coordinator` returns a random

¹<http://www.mozartspaces.org>

²<http://www.gnu.org/licenses/agpl.html>

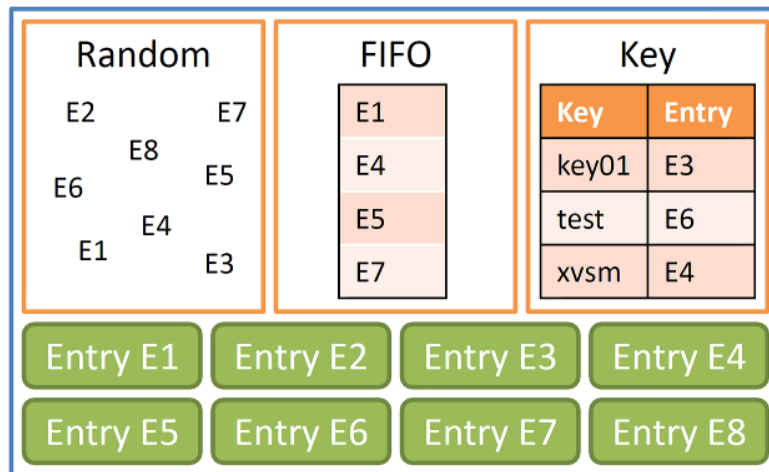


Figure 3.3: XVSM container [Dö11]

entry, while the `FIFO` Coordinator returns entries in the same order as they have been written into the container. The `Key` Coordinator on the other hand, allows to label entries with a unique key and to access them again via this key.

Already with the presented coordinators, well-known communication paradigms, such as message queues (`FIFO` Coordinator), can be realized. Not to speak about the other coordinators, including the *Last In – First Out* (`LIFO`), `Label`, `Type`, `Vector`, `Linda`, and `Query` Coordinator. Notably, the `Key` Coordinator, the `Linda` Coordinator, which allows to “search” for entries via template matching, and the `Query` Coordinator, which uses a *Structured Query Language* (`SQL`)-like query language for filtering entries, leave nothing to be desired.

The operations possible on each container are to `write`, `read`, `take`, and `delete` one or more entries. Additionally, one can define aspects that are executed before or after a container operation (cf. figure 3.4).

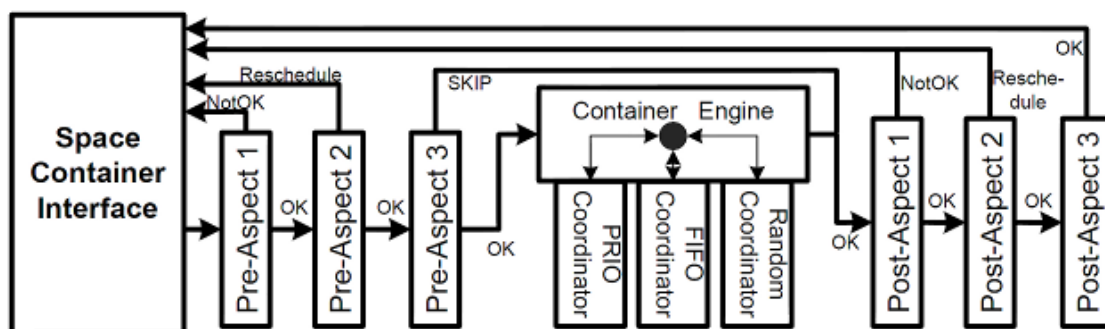


Figure 3.4: XVSM aspects [MKS10]

In `Pre-Aspects` one could e.g. perform data validation tasks, whereas logging could be

performed in `Post-Aspects`. If errors occur, aspects can be rescheduled to retry the current operation later. They can also return `NotOK` to totally abort the operation or `SKIP` to skip the actual operation, but execute the other aspects. Further details about the core features of XVSM can be read in [Bar10] and [Döl1]. The formal specification of XVSM is presented in [Cra10].

Four rather new features of XVSM will be described in more detail, as they are of great use in the XVSM Micro-Room Framework.

XVSM Security

First of all, there is XVSM Security, which is outlined in [CK12,CDJK12,CDJ⁺13] and adds access control capabilities to XVSM. These are crucial to achieve the desired privacy and security criteria (cf. sections 3.3.4 and 3.3.5). It is based on *Extensible Access Control Markup Language* (XACML), that allows to define access control policies in a declarative way. Interestingly, also other contributions such as [Nas10] try to enhance privacy in P2P OSNs via XACML.

XVSM Security allows to specify fine-grained access rules on containers. Such an access rule consists of the following fields:

Subjects: Define the set of subjects (e.g. roles or user ids) for which the rule applies.

Resources: Specify the set of containers for which this rule applies.

Actions: The set of container operations (i.e. `read`, `write` or `take`) for which the rule applies.

Condition: Conditions allow to perform queries with arbitrary coordinators on other containers, to come to a dynamic result during runtime. A condition will evaluate to `true` if the query returns at least one entry. The rule will be applied if the condition is fulfilled.

Scope: The scope is also defined via a query, but this time it is performed on the target container of the operation itself. Thereby, it selects a subset of entries on that the rule is applied.

Effect: Defines whether access shall be permitted or denied if this rule matches.

An example rule could be as follows:

Subjects: `role: admin`

Resources: `logContainer`

Actions: `take`

Condition: `confContainer | key('allowDeleteLogs')`

Scope: `query(logDate < now - 30 days)`

Effect: `PERMIT`

The rule allows admins to remove log entries from a log container, but only if there is a special entry with the key “allowDeleteLogs” present in the configuration container (cf. condition). Hence, dynamic changes of access restrictions during runtime are possible without modifying access rules themselves. Additionally, the scope ensures that only entries which are older than 30 days can be deleted.

As can be seen, especially with the condition and scope fields, arbitrary complex restrictions can be implemented. More details about the access rules used in the XVSM Micro-Room Framework can be read in section 5.3.1.

REST API

As has already been stated, our solution provides a REST interface to allow access from the UI. Therefore, the implementation of the UI is completely independent from the framework, thus allowing UI designers to develop it with arbitrary technologies. Normally, this would result in a lot of additional work for the framework implementation as a REST service would have to be added. Fortunately, XVSM already provides full REST support for all of its operations (e.g. `read`, `write`, `take`, or `delete` on a container), as described in [Pro11]. This allows us to focus on implementing the core features of the framework.

XVSM Persistence

XVSM Persistence [Zar12] allows to persist the containers and entries by using Berkeley DB³. It can be configured to write the database log synchronously with committing the transaction, thus ensuring data recoverability in case of a crash, which is important for consistency.

The authors of ePOST [MPHD06], a P2P-based email system, describe some of the pitfalls of developing reliable P2P systems. One of them is to use files to maintain persistent data as the number of files is likely to increase drastically with each joining peer. Thereby, one would be better off by using a RDBMS from the beginning, which underlines the feasibility of XVSM Persistence.

Distributed Transactions

In contrast to the features above, distributed transactions are included neither in the current stable version (2.2), nor in the developer version (2.3) of Mozartspaces. Only local transactions on the same space instance (i.e. the same machine) are supported. They allow to perform multiple container operations on the local space in an atomic way, thus ensuring the well-known ACID properties as proposed by Haerder and Reuter in [HR83]. However, in a P2P environment, distributed transactions that range over multiple peers and thus across the whole shared space are required. Hence, an approach for XVSM has been developed and introduced in [Brü13]. For our implementation, we have integrated this solution and thereby created a modified version of XVSM.

The described capabilities of XVSM allow us to rely on one library only and thus to avoid the use of many different technologies, such as e.g. Diaspora does. Therefore, we keep the framework itself as simple as possible.

3.2.2 Java

As already stated, the framework can be extended easily via modules (i.e. micro-rooms and plugins). Hence, it is important with respect to extensibility but also simplicity to use a proper programming language for developing these modules. This language has to be well-known by most of the developers, platform independent, and capable of arbitrary features that developers might need.

³<http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html>

Therefore, we decided to use Java, as it fulfills all of these criteria. Considering popularity, it reaches rank 2 in TIOBE's Programming Community Index for July 2013, directly after C [16]. This index measures the popularity of all programming languages world-wide, based on the number of skilled engineers, courses, and third party vendors. An additional benefit of Java is that it allows us to use Mozartspace (cf. section 3.2.1) directly. Therefore, the whole framework is implemented only in Java and thus kept simple regarding technologies used.

Micro-rooms and plugins can be implemented by single Java classes, extending well-defined base classes. These classes only need to be compiled into a *Java Archive* (JAR) file and put into a specified folder of the XVSM Micro-Room Framework, which will load them automatically at startup.

3.2.3 XML

The XVSM Micro-Room Framework also allows to configure the modules described in the previous section and to create workflows by orchestrating them in business logic files. They can be provided either by application developers or by end-users themselves. Therefore, a popular and simple descriptive language has to be used for specifying these files.

XML seems to be the most suitable candidate, as it is both well-known and easy-to-learn. Additionally, the structure of business logic files is kept as simple as possible, i.e. it restricts itself to defining rooms, plugins and connections between actions. The files need to be stored into a predefined folder from which the XVSM Micro-Room Framework will read them at startup.

3.2.4 HTML5

To also keep the UI as simple as possible, we will use *Hypertext Markup Language* (HTML). HTML has been well-known since the beginning of the Internet, but has become even more popular in the last years. This is mainly because of two factors: First, the emergence of smart phones allowed HTML developers to not only create web-based UIs but also fully working apps for Android and iOS. Second, the actual version of HTML – HTML5 – allows developers to use many new standardized features. Some of them are the direct embedding of audio and video without any plugins, an accessible local storage for saving data, thread-like web workers, and WebSockets for direct P2P communication [17].

Besides the increasing popularity and thus spread knowledge of HTML5, it is also relatively easy to learn and use, compared to other technologies such as Java or C++. A final – and very big – advantage of HTML-based UIs is their portability, as they can be used on basically any machine with a browser. This includes Windows, Mac, and Linux PCs as well as smart phones, regardless if they are using iOS, Android or Windows Phone. So, also considering the users' familiarity with Facebook's web-based UI, HTML5 is probably the best choice for developing the UI of an P2P OSN nowadays.

3.3 Fulfillment of Requirements

In the following, we will describe how the presented concepts and technologies are used by the XVSM Micro-Room Framework to achieve all of the requirements defined in section 2.4.

3.3.1 Functionality

In order to allow developers to implement the desired functionality of their P2P application without limitations, the XVSM Micro-Room Framework uses a module-based architecture. Hence, developers are able to add new features (including arbitrary ways of communication and collaboration) by implementing their own micro-rooms and plugins (cf. section 3.1).

3.3.2 Simplicity

To keep the complexity of the framework itself as low as possible, we only use Java and XVSM (which is itself written in Java) to implement it. This space-based middleware provides us with all necessary features to create a proper P2P framework.

P2P application developers can develop their own modules in pure Java and configure them via business logic files in XML format. As Java and XML are well-known technologies, it is easy for developers to get acquainted with the framework.

But also on the user side, the effort for getting everything set up and using the P2P application is kept low. The XVSM Micro-Room Framework itself (i.e. a single JAR file) and the additional module and business logic files can be packed into a deployment package by the application developer. The user then only needs to download, unpack, and start it. Also, no encryption is used to maintain privacy, and thus no additional PKI needs to be set up. Instead, intelligent replication is used, as described in section 3.3.4.

Concerning the creation of the UI, P2P application developers face no limitations. The XVSM Micro-Room Framework provides a REST interface via which actions of the modules can be called. Thereby, application (i.e. module) and UI development can be strictly separated, thus reducing the complexity of the development itself. Also, it allows UI developers to implement their UI in any technology that supports communication with REST services, e.g. Java or C++. For the P2P OSN proof-of-concept, we use HTML5 to develop the UI. Besides the benefits presented in section 3.2.4, the choice of this technology is also important to create a Facebook-like user experience and thereby increase user acceptance. To make UI development with HTML5 even more convenient, we implement a *JavaScript* (JS) wrapper library that encapsulates calls to the framework's REST interface.

3.3.3 Extensibility

The XVSM Micro-Room Framework provides full module support, i.e. additional functionality is addable without modifying the framework itself. This is achieved by the micro-room concept as presented in section 3.1.

3.3.4 Privacy

Concerning privacy, our solution needs to provide four different aspects: First, it has to protect every peer's data against abuse by third parties. Second, it needs to allow fine-grained (group-based) access restrictions for each data record. Third, it has to guarantee deletion of data over all involved peers, and finally, it has to provide a mechanism for users to show where their data actually is located at.

The presented P2P OSNs satisfied the first two requirements via encryption, the third partly with unique approaches, and the fourth not at all. Considering the first two requirements, encryption seems to be the state-of-the-art. This is confirmed by other similar solutions like Freenet [CSWH01], which aims to create a secure and uncensorable global storage system based on P2P. Vegas [DW11] is another privacy-preserving P2P OSN that ensures privacy by using public key cryptography. Also in [ADBS09], Anderson et al. propose a client-server OSN, where the server is untrusted, and privacy is ensured by end-to-end encryption.

However, encryption also comes with a set of problems. In [BB11] Bodriagov and Buchegger compare the different encryption systems of Diaspora, Persona, PeerSoN and Safebook. Their findings underline the importance of encryption systems to be efficient in order to keep the P2P OSNs scalable. Greschbach et al. [GKB12] point out another problem that can't be countered with encryption, namely untrusted relay nodes. By analyzing meta data such as communication flows or the IP addresses, intermediary nodes can draw conclusions about the interacting users, e.g. their relationships and locations. As data in encryption-based P2P OSNs is likely to be spread across many untrusted peers to ensure availability, this threat is even bigger in such systems than in those without encryption at all. Finally, encryption-based P2P OSNs need a PKI, which requires a lot of effort to set up and maintain. As a global PKI is unlikely to become reality anytime soon [LOP05, HBKC11], such solutions are often infeasible for the common user.

Therefore, we propose a solution that doesn't simply rely on encryption, but uses other mechanisms to ensure privacy. The first requirement (i.e. data protection against abuse by third parties) is achieved by the concept of micro-rooms in combination with editable replication settings. Thereby, data within a room can be replicated, e.g. only to all other room members that can read the data anyways. This intelligent data replication is especially suitable for OSNs, as friends that can read personal data or messages on the user's wall can act as replicas in order to provide a decent level of data availability. Please note that a similar approach is also used in Safebook (cf. section 2.2.5).

To fulfill the second requirement (i.e. fine-grained access restrictions), we use the capabilities of XVSM Security (cf. section 3.2.1). With it, actions of micro-rooms can be restricted based on user ID, role, room membership, room ownership, or item ownership. Additionally, arbitrarily detailed access restrictions can be implemented within the actions themselves if required. Thus, data access can be defined on a very fine-grained level.

The third requirement (i.e. guaranteed deletion) is ensured via the framework's internal replication mechanism. As all actions, also delete actions will be replicated to all other peers currently online. Those that are offline will synchronize at their startup and thereby catch up the delete action as soon as possible.

In contrast to all proposed solutions, the XVSM Micro-Room Framework provides a possibility for users to see where their data is currently replicated to. Therefore, each item (i.e. the internal representation of a data record) contains additional meta data about the current replication status. As this meta data is inseparably connected to the actual data itself, the UI designer can easily show it to the user without any additional effort. Thus, users will be aware of who actually possesses and thereby can access their data. Hence, they are more likely to reconfigure their access restrictions (cf. privacy requirement two).

3.3.5 Security

Finally, the framework needs to ensure a decent level of security, i.e. it has to prevent attacks by malicious users that might manipulate the peer client software or requests to other peers. The XVSM Micro-Room Framework ensures this by securing the micro-room actions with XVSM Security. As the replication of these actions is the only way that peers communicate with each other, other peers can't access data directly. Of course, attackers could manipulate the framework locally to send forged action requests to other peers. Nevertheless, they won't be executed on remote peers as their framework instances are still running with all access restrictions in place.

Now that we have described the concepts of the XVSM Micro-Room Framework to fulfill all requirements, we will present its architecture in more detail in the next chapter.

CHAPTER 4

Design

In this chapter, the structure of the XVSM Micro-Room Framework will be presented, thus allowing the reader to get a basic overview of how the framework actually works.

4.1 Architecture Overview

A P2P application that is developed with our approach consists of three major parts:

1. XVSM Micro-Room Framework
2. Files
3. User interface

The framework instance itself generates the full back-end of the application, including business logic and data layer. To achieve this, it acts as a runtime interpreter that performs operations according to its configuration, which is done via different files.

Since the UI is highly depending on the application itself, it cannot be generated automatically. However, arbitrary technologies for UI generation are supported. For our P2P OSN we will use HTML5 due to its benefits (cf. section 3.2.4). To interact with the framework, the UI designer can use the JS wrapper library developed by us, which encapsulates the calls to the REST API of the framework. The JS library will be discussed in detail in section 6.3.2.

It is also important to note that the UI doesn't need to run on the same machine as the framework instance. Thereby, we also allow thin-client-like deployment scenarios where users could run their UI, e.g. on their smart phone and connect to a framework instance running on their *Personal Computer* (PC). Further deployment possibilities will be discussed in section 6.4.

Thus, the communication between user and framework takes place only via the UI. The P2P communication on the other hand is performed just between the framework instances. Therefore, the XVSM Micro-Room Framework provides two interfaces: the REST interface and the

XVSMP interface. The REST interface is based on the XVSM REST API implementation of Proinger [Pro11] (cf. section 3.2.1). XVSM is an XML-based protocol for XVSM that suits the needs of P2P environments, as it provides asynchronous event-driven communication. It is described in detail in [Dö11] and won't be focused in our contribution. Theoretically, both the UI and other framework instances could communicate via both interfaces with a framework instance. However, our design intends the REST interface for UI communication, whereas inter-peer communication takes place only via the XVSM interface.

4.2 Components

The components of the framework itself and the interactions with the two other parts can be seen in figure 4.1 and will be described in detail in the following sections.

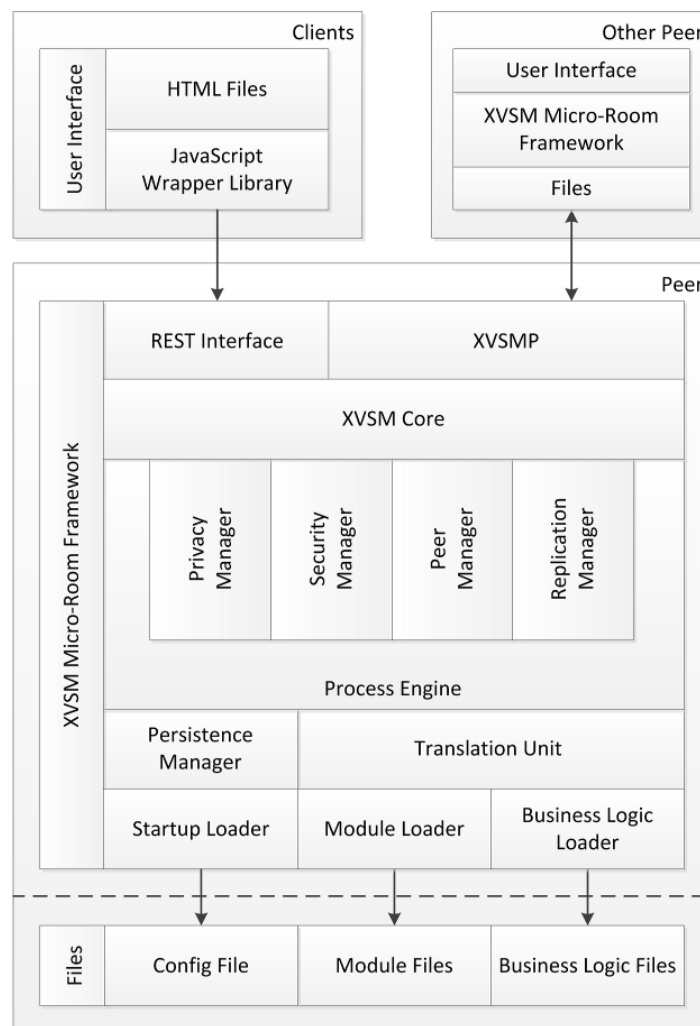


Figure 4.1: XVSM Micro-Room Framework component view

4.2.1 Startup Loader

The startup loader is the first component that will run after starting the framework. It reads the content of the `config.properties` file, which has to be located in the same directory as the framework executable. This file has to be configured and provided by the application developer for configuring the framework itself. It defines which module loader and business logic loader should be used as well as other important parameters such as the user ID and password of the peer. The detailed structure of this configuration file will be described in section 6.1.1.

4.2.2 Module Loader

By providing module files, developers can add the desired functionality to the framework. Thereby, they need to implement their micro-room and plugin Java classes, compile them and put the created `.class` files into a JAR file. All such JAR files located in the `modules` directory will be loaded by the module loader during the framework's initialization phase. One file, namely `default.jar`, is provided with the framework itself. This archive includes all of the micro-rooms that we develop in this work. Other JAR archives can be added as described if the provided modules do not offer sufficient functionality.

The module loader is available in a local and a remote version. In the local variant it loads files from the local `modules` directory as described. The remote version retrieves module files from a central web server specified via `config.properties`. Thereby, deployment and maintenance of modules across all peers can be eased even more.

4.2.3 Business Logic Loader

After loading all available module archives, the business logic loader will read all XML files located in the `business-logic` directory. These business logic files can be provided by the application developer or the user itself. They are used to configure micro-room and plugin instances and to connect actions of micro-rooms with each other, thereby specifying workflows. It is intended that each business logic file contains the logic for exactly one scenario, in order to keep the files simple. Such a scenario could be, in case of an OSN, everything concerning the personal profile.

Just as the module loader, also the business logic loader is available in a local and a remote variant. The local business logic loader works as described, whereas the remote version downloads the XML files from a central server configurable in `config.properties`. The benefit of this variant is again that the business logic files can be updated on one location (e.g. to fix bugs) and thus are available immediately for all peers at their next startup.

4.2.4 Translation Unit

The already loaded modules and the XML content of the business logic files are then passed to the translation unit. There, the XML content will be parsed and used to create and configure instances of the corresponding micro-room and plugin classes. These internal data structures will then be handed over to the process engine.

4.2.5 Persistence Manager

Besides creating the data structures from scratch, such as done by the translation unit, the persistence manager restores them at startup. Therefore, it uses XVSM Persistence (cf. section 3.2.1) and a special `Persistence Container` for storing meta data. On startup, it re-creates all micro-rooms and plugins possible from the internally used Berkeley DB and the `Persistence Container`. These data structures are again passed to the process engine. Summarizing, the aim of this component is to save in-memory data at shutdown on the hard disk, and to restore this data again when the framework instance starts up the next time.

4.2.6 Process Engine

The process engine is the heart of the whole framework. On startup, it initializes the data structures obtained by the translation unit or persistence manager. Additionally, it performs various synchronization tasks (cf. upcoming sections). During runtime, it handles incoming requests that have been sent by the user, i.e. by the UI via the REST interface. After processing the corresponding action, it supplies the response for the request if necessary, so that the UI can read it. More details about request handling will be presented in section 4.3. Finally, the process engine has to ensure a clean shutdown, again requiring several synchronization tasks.

4.2.7 Privacy Manager

Since a central part of this work is privacy, it is handled by an own component. The privacy manager monitors where which data should be replicated to and to which other peers it already has been replicated to. This data is added as meta data to each item (i.e. data record) and thus can be visualized in the UI easily. Hence, users are able to see who is currently in physical possession of which of their data.

4.2.8 Security Manager

This component checks whether external calls of the UI or other peers (i.e. due to replication requests of their framework instances) are allowed. In case of the UI, the user first needs to provide his/her user ID and password, e.g. in a login mask. These credentials are then forwarded to a central authentication service (e.g. OpenAM¹) that verifies them and returns a *Single Sign-On* (SSO) token. The framework instances of other peers know their respective credentials, as they are specified within their `config.properties` files. But just as the UI, they have to send them to the central authentication service first, in order to obtain a valid SSO token.

In both cases, the obtained SSO token needs to be attached to all requests that are sent to a framework instance. By contacting the authentication service, the security manager is then able to derive the requesting user's ID and his/her roles without doubt. This information is used within the access rules of XVSM Security (cf. section 3.2.1) to decide whether access to the requested action is allowed or not. The basic set of access rules is also defined by the security manager and remains static (cf. section 5.3.1). However, they are influenced directly by the

¹<http://forgerock.com/what-we-offer/open-identity-stack/openam/>

settings made in the business logic files. More about the security concept will be described in section 4.4.

4.2.9 Peer Manager

At startup, not only the described configuration files need to be loaded, but also actions need to be taken to integrate the peer into the P2P network. During runtime, the peer has to maintain a list of all other currently online peers. At shutdown, it needs to tell all other peers that it is now offline. All of these tasks are performed by the peer manager. More details about peer management will be presented in section 4.5.

4.2.10 Replication Manager

An important aspect of P2P applications is the replication of data between its peers. This is necessary to ensure a certain level of data availability if peers go offline. Besides the replication itself, the replication manager is also responsible for data synchronization with other peers if the peer goes online. Further details about the replication will be discussed in section 4.6.

4.3 Request Handling

Now that we have described all components of the framework, we will outline the way requests from the UI are handled. Figure 4.2 shows the typical way of interaction with the framework, which is based on the secure service space concept presented in [CDJ⁺13].

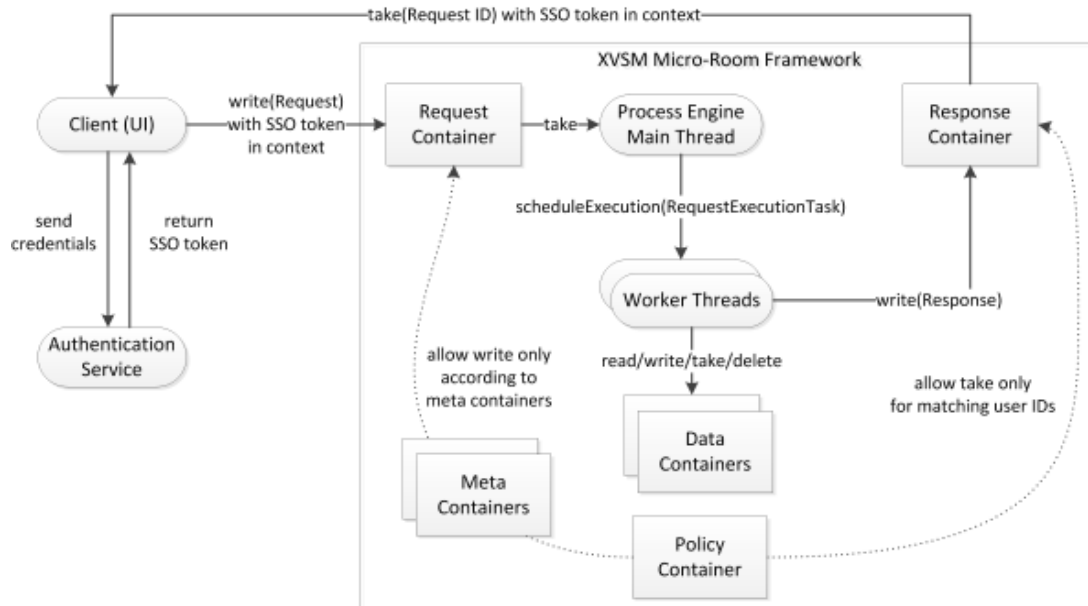


Figure 4.2: XVSM Micro-Room Framework request handling

It can be seen that first of all, the UI needs to send the credentials entered by the user to a central authentication service, as already described. In turn it retrieves a SSO token, which is necessary for interacting with the XVSM Micro-Room Framework. Each interaction with the framework that doesn't include a valid SSO token will be denied (cf. section 4.4). E.g. if Anna clicks on the button to confirm a friendship request of John, some kind of event is triggered within the UI. In this event, the UI needs to create a `Request` object, containing request ID, scenario name, room name, action name and parameters of the action. In the example this could be "Anna1234" "Friendship scenario", "My Friends", "confirmNewFriend" and "John". The `Request` object then needs to be written into the `Request Container` of the XVSM Micro-Room Framework via its REST API. In the framework the `Request` object will be taken out of the container, transformed into an executable `RequestExecutionTask`, and handed over to a pool of worker threads of the process engine. These threads will then execute the request and write the outcome as a `Response` object into the `Response Container`. In the example, this could be a simple "Ok" or "true", if the friendship confirmation was processed without errors. From this container, the UI can read the `Response` object again via the framework's REST API and finally show a corresponding message to the user, e.g. "New friend John has been confirmed successfully!".

The overall sequence of operations concerning request handling will be discussed in more detail in section 5.2. The exact specifications of the REST API for UIs, in order to send requests and read responses, is presented in section 6.3.1.

4.4 Security

The request handling process and the overall framework are tightly secured. Direct access to data containers (defined by the modules) or meta containers (defined by the framework itself) is completely denied from outside the framework. All of them are secured via request-based authentication, i.e. requests targeting those containers can't even be written into XVSM's internal `__Request Container` and thus won't be processed (cf. [CDJK12]).

Only the `Request` and the `Response Container` are accessible from the UI and additionally restricted via access rules of XVSM Security. All of these access rules are stored within the `Policy Container` (cf. figure 4.2). For the `Request Container`, the rules are based on several meta containers (i.e. `Permission`, `Denied Users`, `Room Member`, `Room Owner` and `Item Owner Container`). The meta containers are described in detail in section 5.1.3, the access rules are defined in section 5.3.1.

The information stored in `Permission Container` is derived from the business logic files. There, access can be specified on the basis of room actions. So, the UI can only write requests into the `Request Container` that contain actions allowed in the business logic files. Reading from the `Response Container` is restricted to only those `Response` objects that have been created by a `Request` object sent by the same user ID. In the process engine, this user ID is first retrieved by verifying the SSO token of the request and later on passed to the response. Thus, it cannot be influenced or manipulated from the outside.

Considering the `Request Container`, access can be configured per micro-room action and based on a wide range of information:

1. Allow only several users, identified via their IDs.
2. Allow only certain roles, identified by their role names.
3. Allow only users with special permissions, i.e. either
 - a) all members of the micro-room on that the action is called.
 - b) the owner of the micro-room on that the action is called.
 - c) the owner of an item that is passed as an argument to the called action.

All of these restrictions can be defined in the business logic configuration files via the `Access` plugin (cf. section 6.1.2 for more details). Additionally, the different ways can be combined in arbitrary combinations, e.g. an action `removeBook()` in a book sharing room could be restricted to the owner of the book (i.e. the item owner) and the head of the book sharing group (i.e. the room owner) only. Please note that such a combination is always performed by a logical OR.

4.5 Peer Management

Unfortunately, XVSM doesn't support native peer discovery mechanisms in its current version. Thus we have to implement this essential feature of the XVSM Micro-Room Framework ourselves.

For deciding which approach suits best, we first need to describe all of the possibilities. According to [LCC⁺02], P2P networks can be divided into the following categories:

Centralized: Systems with this architecture have a central directory that contains the addresses of all peers. Therefore, peers first communicate with this lookup service to get information about the other peers, and then connect directly to them for exchanging data. An example for such a system is Napster, which also shows the disadvantages of the approach: In July 2001 it was shut down as it has been used to illegally share music [18]. That was only possible by taking down its central directory, which represented a single point of failure.

Decentralized and Structured: In these P2P systems no central lookup service exists. However, peers have a certain level of "structure", i.e. some determined algorithm where to store and search data again. Thus, they are able to find each other again, even without a central directory. An example for such a system is OpenDHT, which is used by several of the P2P OSNs discussed in section 2.2. Other approaches of this category, such as Pastry, Chord or CAN, are described in [WL03].

Decentralized and Unstructured: Finally, there are systems which neither have a centralized directory, nor a structure. These are highly focused on ad hoc communication, i.e. peers randomly joining and leaving the system. The storage locations of data are not determined, thus peers need to query their neighbors to find data again. A typical method to do so is flooding, which means that all neighbors within a certain range are asked for the data.

The thereby generated, huge network load can be considered the most important disadvantage of this approach. An example for a decentralized and unstructured P2P network is Gnutella, which is described in detail in [Rip01].

For the XVSM Micro-Room Framework we use a centralized peer discovery approach due to its simplicity. Peer management is not the main focus of our contribution and could be enhanced in the future, e.g. if XVSM provides native peer management capabilities. This is easily possible via exchanging the peer manager component.

Our implementation therefore includes a simple lookup server (i.e. the so-called “peer coordination server”) containing the addresses of all peers that are currently online. At startup each peer registers itself at this server by providing its user ID, IP address and port. At shutdown each peer removes the corresponding entry again from the peer coordination server. During runtime the server keeps all peers up-to-date by performing `write` and `delete` operations on special callback containers that are located on each peer. The application specific location of the server can be configured via the `config.properties` file.

As the framework itself, also the peer coordination server is secured via XVSM security. Thus, for registering and unregistering, the corresponding peer needs to provide a valid SSO token. The peer coordination server only allows the action if it fits to the user ID correlated with the token. Therefore, peers can only register and unregister themselves, preventing malicious peers from masquerading themselves with another user ID. The concrete access rules can be found in section 5.3.1.

4.6 Replication

Currently, a solid basis for integrating different replication mechanisms in XVSM is developed [Hir12, CHKSC13]. However, replication support, as we need it for the XVSM Micro-Room Framework, is not available yet. Therefore, we need to come up with a proper replication mechanism ourselves. In the future, our mechanism can easily be exchanged by XVSM’s capabilities via just implementing a new replication manager (cf. section 8.3).

The desired replication approach has to both enable data availability and guarantee privacy, as these are main requirements of the XVSM Micro-Room Framework. To decide which replication approach fits best, we first have to discuss all the possibilities. According to [MPV06], replication can be classified by three different aspects, which are presented in the following.

4.6.1 Single-Master vs. Multi-Master

First of all, it has to be decided how many primary copies of an object should exist. In a `Single-master` approach, only one peer has a primary copy and all the other replicas are secondary copies. Read access is allowed on any of the replicas, but updates can only be performed on the primary copy. Thus, if the master is currently offline no updates can occur. This is a potential bottleneck and could block actions in the workflow.

The `Multi-master` approach on the other hand allows multiple primary copies of an object. Hence, no blocking update actions are possible as long as any of the masters is online.

But on the other hand, as all of the primary copies are updatable concurrently, this could result in replication conflicts. Therefore, special coordination or reconciliation strategies are required. In table 4.1, the advantages and disadvantages of both approaches are summarized.

Compared aspect	Single-master	Multi-master
<i>Distinguishing feature</i>	One primary copy	Multiple primary copies
<i>Synonymous</i>	Master/slave	Update anywhere
<i>Distributed concurrency control</i>	Not applied	Coordination Reconciliation
<i>Up-to-date values at</i>	Primary copy	Unknown copy
<i>Update approach</i>	Centralized	Distributed
<i>Update blocking</i>	Master site down	All master sites down (if using reconciliation)
<i>Possible bottleneck</i>	Yes	No

Table 4.1: Single-master vs. multi-master replication [MPV06]

4.6.2 Full Replication vs. Partial Replication

Second, one has to decide where the replicas should be stored. Full replication means that a copy of each object will be replicated to all peers, whereas in Partial replication each peer only holds a subset of all objects. This has the benefit that it requires less space on the peer and reduces the overall network traffic. Full replication on the other hand ensures maximum data availability and keeps the replication protocol simple. Table 4.2 shows all pros and cons of both replication modes.

Compared aspect	Full replication	Partial replication
<i>Distinguishing feature</i>	All sites hold copies of all shared objects	Each site holds a copy of a subset of shared objects
<i>Load balancing</i>	Simple	Complex
<i>Availability</i>	Maximal	Less
<i>Storage space</i>	May be expensive	Reduced
<i>Communication costs</i>	May be expensive	Reduced

Table 4.2: Full vs. partial replication [MPV06]

4.6.3 Synchronous Propagation vs. Asynchronous Propagation

Finally, it has to be decided whether the replication propagation should occur synchronously or asynchronously. Synchronous propagation ensures atomic transactions across all peers. Thus, it fulfils the so-called “one-copy-serializability” criterion, as defined by Bernstein et al. in [BHG86]. This means that all replicas always have the same state and hence appear to the outside as one logical copy to which access is always performed serially, just as if the object was located only on one peer. To realize this, special protocols such as the *2-Phase Commit* (2PC) protocol have to be used. This protocol is divided into two phases, i.e. the voting phase and the decision phase. In the voting phase the coordinating peer sends a commit request to all other peers. Now the decision phase starts, where the coordinator waits for all peers to answer and then

decides whether the transaction should be committed or not. Therefore, each peer executes the transaction up to the point of commit, to see whether it could be executed. If so, it sends back a `Yes` to the coordinator, if not, it sends a `No`. If all peers answered `Yes`, the coordinator confirms the transaction by sending a `Commit` message to all peers that then commit their pending local transactions. If at least one peer answered `No` in the voting phase, the coordinator will decide to abort the transaction and thus sends an `Abort` message to all peers, which will then rollback their local transactions. Transactional consistency is often a desirable feature, but it comes at the cost of bad scalability. The negotiation causes some overhead, and if one peer is offline, the whole system blocks. Also, with every additional peer the overall commit time as well as the failure probability increases.

Asynchronous propagation on the other hand doesn't ensure transactional consistency among all peers. The transaction is committed locally immediately and propagated to the other peers as soon as possible. Therefore, this approach is often referred to as "lazy propagation", ensuring only eventual consistency.

It can be divided into two subtypes, namely `Non-optimistic` and `Optimistic asynchronous propagation`. `Non-optimistic` approaches assume that many concurrent updates and thus replication conflicts will occur and therefore try to guarantee one-copy-serializability. However, this is not easy, as the transactions are not atomic. Hence, complex graph-based replication strategies are necessary to at least come close to the goal of one-copy-serializability. To measure how likely it is that the data used in the transaction is indeed up-to-date, a new indicator has been introduced, the so-called "freshness" of the data.

`Optimistic` methods on the other hand assume that there won't be many replication conflicts or none at all. The update propagation occurs in the background, and if an update conflict occurs, it will be reconciled by predefined rules. A lot of known systems are based on this propagation method, e.g. both *Domain Name System* (DNS) and *Concurrent Versions System* (CVS). A big benefit of the asynchronous approaches is that such systems scale much better, since single offline peers don't block transactions initiated by another peer. In table 4.3, the three different approaches are compared.

Compared aspect	Synchronous	Asynchronous	
		Non-optimistic	Optimistic
<i>Distinguishing feature</i>	All replicas change in the same update transaction	Commit as soon as possible, then propagation	Commit, then background propagation
<i>Synonymous</i>	Eager propagation	Lazy propagation	
<i>Consistency criterion</i>	One-copy-serializability	Freshness	Eventual consistency
<i>Local reads</i>	Return up-to-date values	Return up-to-date values with high probability	No guarantees
<i>Distributed Concurrency control</i>	Yes	No	No
<i>Scalability</i>	A few tens of sites	A few hundreds of sites	Larger number of sites
<i>Environment</i>	LAN and cluster	LAN, cluster, and WAN	Anywhere

Table 4.3: Synchronous vs. asynchronous replication [MPV06]

4.6.4 Our Replication Approach

Based on the theoretical background presented, we will now describe our replication approach.

General Approach

Due to several reasons, a `Multi-master Partial Synchronous and Asynchronous Optimistic` replication approach seems to suit best for the `XVSM Micro-Room Framework`.

The main cause for the `Multi-master` decision is that in `Single-master` mode, updates and thus overall workflows can block if the peer containing the primary copy is offline. Now, in P2P OSNs, it is very likely that a friend is offline when you are online and, therefore, it would be impossible to, e.g. write onto his/her wall, as he/she certainly would be the owner of the primary copy of the underlying data structures. With the `Multi-master` approach we can counter this problem.

As we don't have any data encryption in the `XVSM Micro-Room Framework`, we need to ensure privacy by intelligent replication of data. Hence, `Full replication` would break privacy, as all peers would have access to all data of all other peers. Therefore, we require `Partial replication`, so that data is replicated, e.g. only between friends.

Finally, concerning the propagation mode, we use a hybrid approach. In [Brü13], Brückl describes a way of how to add Paxos Commit support to `XVSM`. Paxos Commit [GL06] basically is a distributed commit protocol similar to the 2PC protocol. It is based on Paxos [Lam01], which is a consensus algorithm for distributed systems. In contrast to the 2PC protocol it allows more flexibility, e.g. it is possible to define more than one coordinator. The implementation of Brückl additionally supports the possibility to set the percentage of peers that are allowed to send a `No` answer, still allowing the coordinator to commit the transaction.

With this new `XVSM` capability we can ensure transactional consistency among all peers that are currently online, i.e. `Synchronous propagation`. Peers that are offline will retrieve missed messages when they come back online. Therefore, they will ask other peers if they missed something, which confers to `Asynchronous Optimistic propagation`. Via this hybrid approach, we can ensure a maximum level of consistency, but at the same time avoid blocking transactions if peers are offline. Replication errors still are possible if, e.g. peer A updates a data set while no other peers are online and goes offline before peer B gets online and updates the same data set. This requires special reconciliation strategies that are out of scope of this work (cf. section 8.3).

Concrete Architecture

Considering the architecture of the `XVSM Micro-Room Framework`, there are basically two different possibilities of how to actually integrate the described replication approach.

The first alternative is to copy each request coming from the UI and write it into the `Request Container` of every other online peer. Offline peers need to fetch missed requests when they get online. Therefore, all peers will execute the same requests and thus be synchronized. A benefit of this approach is that no further security constraints need to be defined. Only

the `Request Container` is affected by the replication, which needs to be secured against malicious user requests anyways. To additionally secure the container against forged replication requests, exactly the same access rules would be sufficient. However, the big disadvantage of this solution is that transactional consistency is not ensured among all online peers. The order of execution of the requests is not guaranteed and can be different on each peer due to network latency. Therefore, it is very likely that replication conflicts and thus data inconsistencies will occur.

The second approach is to use a distributed transaction for handling a request from the UI. In this distributed transaction, all operations (i.e. `read`, `write`, `take` or `delete`) on the data containers will be executed on all online peers, thereby ensuring the same order on each peer. Offline peers again need to synchronize missed requests when they get online. The big advantage is that consistency among all online peers is ensured. However, this approach would imply large security problems as access to each data container would be required for each peer. Hence, every peer could read all the private data of other peers or even worse, modify or delete it.

As can be seen, both approaches have severe drawbacks, making them useless for the XVSM Micro-Room Framework. The best solution, that thus has been chosen, is a combination of both alternatives. If a request from the UI is handled, a distributed transaction is created across all online peers. This transaction is used for all operations on the data containers of the initial peer. If all operations succeeded on this peer, it writes a copy of the request to all other online peers. Now, the big difference to the first approach is that this copy is not written into the `Request Container`, but into a special `Replication Request Container` on the other peers. Also, the write operation is executed within the same distributed transaction. A special post-write aspect on the `Replication Request Container` ensures that each incoming replication request is handled immediately within the same transaction as the write request itself.

Therefore, the originating peer can be sure that the whole request can be executed on the other peer, if writing the request into the `Replication Request Container` succeeded. So, after the commit of the distributed transaction, all online peers are in the same state. If a peer fails while handling the request, the whole distributed transaction is rolled back and the initiating UI will receive an error message that can be presented to the user. The `Replication Request Container` can be seen as an exact copy of the common `Request Container` and thus can be protected by the same security rules. Therefore, this approach combines all benefits: Consistency among all online peers without digging security holes into the whole framework.

The replication targets of each micro-room can be defined within the business logic files via specifying roles or user IDs of the target peers. Details about the configuration are given in section 6.1.2.

Fulfilling the CAP Properties

As is known by Brewer's famous CAP theorem [Bre00], a distributed system cannot guarantee all of the three desired characteristics, consistency (all peers see the same data at the same time),

availability (all requests are always handled) and partition tolerance (the system works even if the network is divided into parts or messages are lost) simultaneously.

If we look at our approach, we can guarantee availability to the full extent, i.e. the system is available even if you are the only peer that is currently online. Also, if other peers crash during a distributed transaction, the transaction will time out and the system will remain available. The trade-off in our approach is between consistency and partition tolerance. We ensure consistency between all online peers, but inconsistencies could occur in special cases due to offline peers, as described above. This decision has been made to ensure availability. Considering partition tolerance, the system has theoretically no problem if the network is divided into several parts, as unreachable peers are treated as offline and the remaining peers can interact normally. Of course, different partitions will develop different data sets over time, but this is no criteria for the partition tolerance meant by Brewer, as long as the system remains functional.

Nevertheless, practically the approach has a problem with partition tolerance, because there are two central servers involved in the current solution, i.e. the authentication service and the peer coordination server. So, a network partition containing both servers will work, while all other partitions will fail. However, if both servers are replaced by pure P2P-based solutions in the future (cf. sections 8.1 and 8.2), partition tolerance will be supported to the full extent.

4.7 Workflow Support

Finally, the XVSM Micro-Room Framework also needs to enable a decent level of workflow support. Therefore, it is possible to connect two actions with each other in the business logic files. However, it is the application developer's responsibility to ensure that the return type of the first action fits to the parameters required by the second action. Considering complex workflows, also the correlation between different connected actions plays an important role. A simple example scenario to outline the problem is shown in figure 4.3.

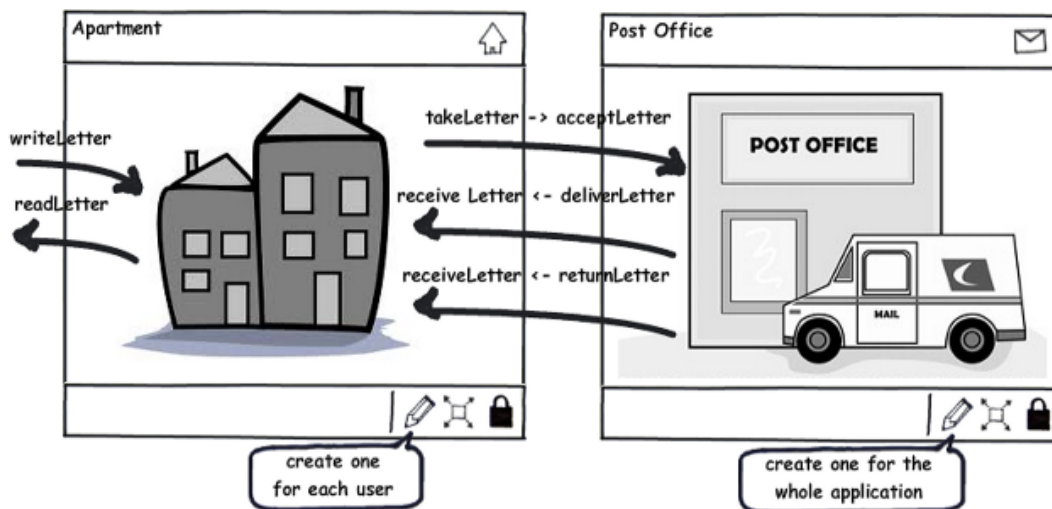


Figure 4.3: XVSM Micro-Room Framework correlation example

The scenario shown contains two different kinds of rooms, an apartment and a post office. It is important to note that due to the configuration, an own apartment is created for each user by the framework automatically. Therefore, the user ID of the corresponding user is added to the room name. This will be referred to as “room extension” from now on. So, the room IDs would be of the following form if we assume that the scenario name is “Letter Delivery” and that two users, “John” and “Mary”, exist:

- Letter Delivery->Apartment->John
- Letter Delivery->Apartment->Mary
- Letter Delivery->Post Office

The actions available in the apartments have the following signatures:

String writeLetter(Item letter): Creates a new letter, stores it in the apartment, and returns its ID (which is generated automatically by the framework).

List<Item> readAllLetters(): Reads all letters stored in the apartment and returns them.

Item takeLetter(Item letter): Removes the letter with the given ID from the apartment and returns it. The `letter` argument only contains the letter ID and no content.

void receiveLetter(Item letter): Receives the passed letter and stores it in the apartment.

The post office allows the following three actions:

void acceptLetter(Item letter): Accepts the passed letter for mail delivery.

Item deliverLetter(Item letter): This action will be called by the micro-room itself after accepting a letter. It delivers the letter with the given ID to the target apartment.

Item returnLetter(Item letter): If the delivery of the passed letter fails, this action will return it to the originating apartment. Therefore, this action is also called automatically by the post office itself.

As can be seen in figure 4.3, three different connections exist in this scenario. The first connection passes the return value of `takeLetter()` (i.e. the letter to send) to `acceptLetter()`. For this connection itself no correlation is required, as the target micro-room (i.e. the post office) is always the same and derived automatically by the framework’s standard behavior.

The second connection passes the return value of `deliverLetter()` (i.e. the letter to deliver) to `receiveLetter()`. In this case a special kind of correlation is required, as in the framework such a connection exists for every post office/apartment pair. Thus, the framework doesn’t know which apartment the letter should be forwarded to (i.e. either John’s or Mary’s apartment). The required room extension (i.e. John or Mary) could either be specified within the logic of the post office micro-room itself, or by defining a proper correlation in the business logic file of the `Letter Delivery` scenario. Such a correlation would need to be defined in the business logic file on the target action (i.e. `receiveLetter()`) and could look as follows:

```
correlationVal="$params[0].content.recipient"
```

This line tells the framework which room extension it should use for resolving the target room (i.e. the room on which the target action should be called). In the example above, the first parameter of the target action (i.e. the letter) will be examined. It is assumed that it contains a `recipient` key within its `content` Map. The `content` field of an `Item` object contains the actual data, whereas the other fields contain various meta data. More details about the structure of an `Item` can be seen later in figure 5.3. The value of the `recipient` key is exactly John or Mary. It is important to note that in this case the correlation is calculated newly every time the connection is used by the framework.

The last connection is used to forward the return value of `returnLetter()` to `receiveLetter()`. However, this time not the recipient should be the target, but the transmitter. Therefore, the transmitter could be stored in the letter, just as the recipient. In this case, a correlation, just as defined for the second connection, would be sufficient. In order to show the full capabilities of the framework, we will assume that this is not possible. Therefore, we need to add a correlation to the `takeLetter()` action of the first connection, which saves the transmitter of the letter:

```
correlationVar=X
```

This line stores the room extension of the source room (e.g. John if the letter was initiated by John's apartment). It is important to note that this variable will be newly assigned only if it hasn't been yet and that it is unique for each item ID. Thus, for every taken letter a single X variable exists, storing from which apartment it originated. Now, in the third connection this variable can be used on the `receiveLetter()` action to restore this information and hence return the letter to the correct apartment. This works with exactly the same line:

```
correlationVar=X
```

Even though the syntax is the same, the semantic is different this time. Previously, the X variable was empty and thus has been assigned a value defined by standard rules. But now, the variable already contains a value and, therefore, this value will be used for correlation. As the passed letter to the `receiveLetter()` action has the same item ID as the letter originally received from the `takeLetter()` action, the X variable can be resolved by the framework. If the variable is unresolvable or no correlation rules are defined at all, the framework will apply standard rules to resolve the corresponding micro-rooms as well as possible. The way they are applied, is visualized in figure 4.4.

As can be seen, it is mandatory to pass an `Item` object within the connection for the correlation mechanism to work. This is required, as the values of correlation variables (i.e. the room extensions) can be different for each action call. Therefore, some kind of anchor is required to store and restore the relations, which in the case of the XVSM Micro-Room Framework is the item ID.

If an item is passed along the connection, first, the correlation variable will be tried to resolve, if one is defined in the business logic file. If a value can be restored from the variable, it

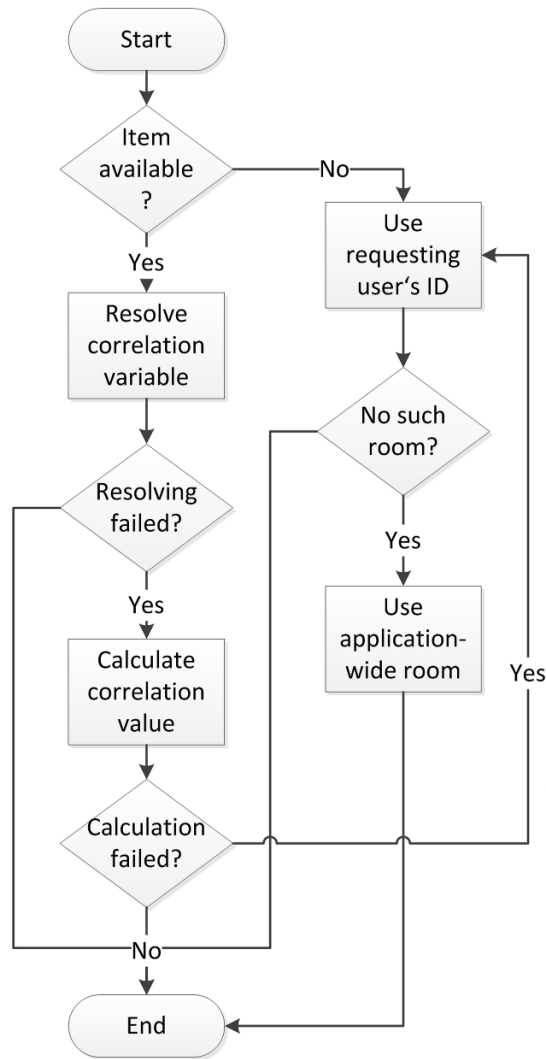


Figure 4.4: XVSM Micro-Room Framework standard correlation rules

will be used. If no value can be restored, the correlation value will be calculated newly, as specified in the business logic file. If this also fails (e.g. because nothing is defined in the business logic file), standard behavior is applied. This is exactly the same as if no item is available at all. The standard mechanism tries to first find a room that has been generated for the requesting user. If no such room exists, the application-wide room will be used.

So, in the case of our example, the post office is determined automatically by the framework's standard behavior (i.e. it is the application-wide post office). For the second connection only a correlation value has been defined that is thus calculated each time the connection is processed. Finally, connection three uses a correlation variable that is assigned in connection one. The value assigned is not specified explicitly with a correlation value and therefore also derived by the framework automatically (i.e. it is the user ID of the requesting user). When connection

three is processed, the variable has already been assigned, and thus the stored value will be used to determine the target room.

Currently, only 1:1 connections are allowed due to simplicity. In the future, 1:m or even n:m connections can be added (cf. section 8.4). Apparently, the whole topic of correlation and therefore also the approach presented is rather complex. Nevertheless, correlation is mandatory for creating complex workflows that thereby are supported by the XVSM Micro-Room Framework.

Implementation

In contrast to the last chapter, we will now go more into depth and describe the implementation of the XVSM Micro-Room Framework in detail, concerning both its structure and its sequence of operations. Additionally, we will describe the internally used containers, security rules, as well as implementation-related challenges and how they have been countered. Thus, the reader of this chapter will get detailed insights about the framework's internals, allowing him/her to modify and extend it if necessary.

5.1 Structural View

First, we will give an overview about the structure of the implementation of the XVSM Micro-Room Framework, followed by explanations of the most important classes, interfaces and containers.

5.1.1 Package Structure

The package structure is tightly related to the component view presented in section 4.2. In table 5.1, all packages, including their content, are described. Please note that all packages are prefixed with `org.xvsm.microroom.framework`.

Package	Description
<code>default</code>	Contains the main entry point of the application and the definition of application-wide constants.
<code>api</code>	All classes and interfaces needed for module developers are located in this package. They will be described in detail in section 5.1.2.
<code>exceptions</code>	Includes all application-relevant custom exceptions.

<code>helpers</code>	Helper classes for file, web, and XML access are contained in this package.
<code>helpers.responsehandlers</code>	Defines custom response handlers for data retrieved by the file and web helper.
<code>loaders.businesslogic</code>	Contains all necessary classes for the local and remote business logic loader as described in section 4.2.3.
<code>loaders.module</code>	The implementation of the local and remote module loader (cf. section 4.2.2) is included in this package.
<code>loaders.startup</code>	Includes the implementation of the startup loader as presented in section 4.2.1.
<code>objects</code>	All relevant transfer objects (e.g. <code>Item</code> , <code>Request</code> and <code>Response</code>) are located in this package.
<code>objects.configurations</code>	Contains various types of <code>Configuration</code> objects which are used for internal configuration of micro-rooms, plugins, and the framework itself.
<code>persistence</code>	The persistence manager (cf. section 4.2.5) is implemented in this package.
<code>processengine</code>	Everything concerning the request handling loop (cf. section 4.3) and the process engine (cf. section 4.2.6) can be found here.
<code>processengine.peermanagement</code>	Includes all classes for the peer manager as described in section 4.2.9.
<code>processengine.privacy</code>	The privacy manager (cf. section 4.2.7) is implemented in this package.
<code>processengine.replication</code>	Contains the replication manager as presented in section 4.2.10.
<code>processengine.security</code>	All classes relevant for the security manager (cf. section 4.2.8), including communication with the authentication server, are located in this package.
<code>translationunit</code>	The implementation of the translation unit as described in section 4.2.4 can be found here.

Table 5.1: XVSM Micro-Room Framework package structure

5.1.2 Classes & Interfaces

Now, that the basic locations of all classes have been clarified, we will present the most important classes and interfaces in detail.

Components

The first class diagram (cf. figure 5.1) shows the components of the framework, as already described in section 4.2

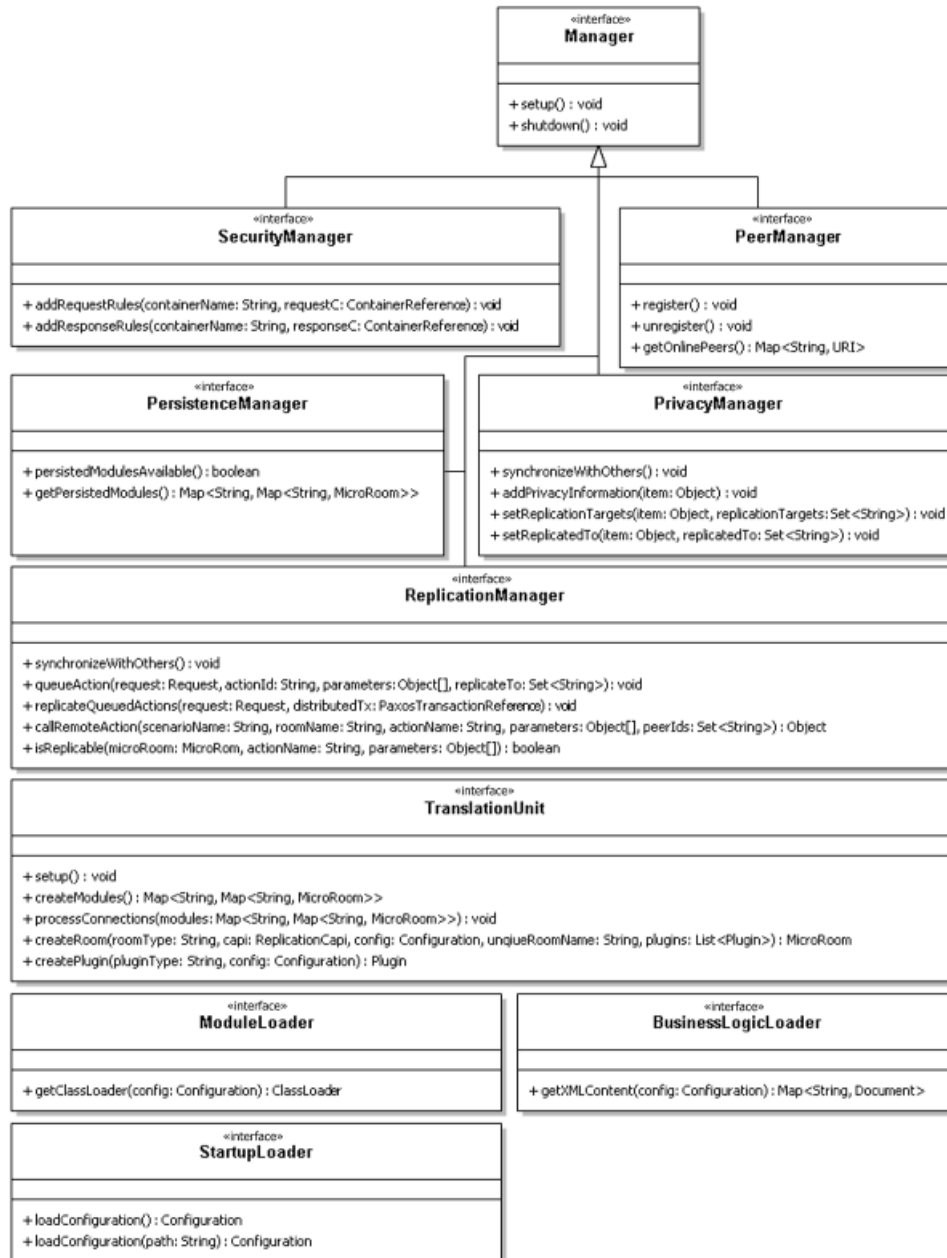


Figure 5.1: Class diagram - Components

As can be seen, each of the components has a corresponding interface, defining the needed methods. In the XVSM Micro-Room Framework implementation, each of those interfaces is implemented by a corresponding Default implementation class, e.g. `DefaultTranslationUnit`. Thus, in the future each component can be exchanged easily by another implementation of the corresponding interface.

The `StartupLoader` returns a `Configuration` object (i.e. the content of `config.properties`) via its `loadConfiguration()` method. This object will be passed to several other components, such as the `ModuleLoader`. This component returns a `ClassLoader` in its `getClassLoader()` method, based on the `Configuration` object and the module files located in the `modules` directory. The `BusinessLogicLoader` on the other hand uses the `Configuration` object as well as the business logic files located in the `business-logic` directory to create a `Map`. This `Map` contains entries in the form of "scenario name -> Document", i.e. "XML file name -> XML file content".

Both the `ClassLoader` and the `Map` are handed over to the `TranslationUnit` that instantiates all modules (`createModules()`) and the connections between them (`createConnections()`). The returned `Map` has a "scenario name -> room name -> `MicroRoom`" mapping. In case persisted modules are available on startup (`persistedModulesAvailable()`), these are restored by the `PersistenceManager`'s `getPersistedModules()` method. Please note, that the returned `Map` has exactly the same structure as described for the `TranslationUnit` and thus can easily be combined. The created `Map` is passed to the process engine, which is shown in figure 5.2.

The process engine creates one `Manager` instance of each `SecurityManager`, `PrivacyManager`, `PeerManager` and `ReplicationManager`. First of all, the `SecurityManager` creates basic access rules for `Request` (`addRequestRules()`) and `Response` (`addResponseRules()`) and adds them to the `Policy Container`.

The `PrivacyManager` is used to store new privacy information immediately to the `Privacy Container` (`setReplicationTargets()` and `setReplicatedTo()`). During request handling, it adds this information automatically to the corresponding items (`addPrivacyInformation()`).

A component that is likely to be changed in future, is the `PeerManager`. It allows to `register()` and `unregister()` the current peer, as well as to retrieve a `Map` of all currently online peers in the format "user ID -> space *Uniform Resource Identifier* (URI)". Thereby, the space URI contains both the IP address and the port of the target peer.

Finally, there is the `ReplicationManager` that is responsible for all inter-peer communication. It needs to synchronize the peer's state at startup (`synchronizeWithOthers()`) and allows to call remote actions on other peers (`callRemoteAction()`). During the execution of an action within the process engine, it is checked whether the action has to be replicated (i.e. it modifies data) via the `isReplicable()` method. If so, the action is queued (`queueAction()`) and executed locally. After all actions of one workflow have been executed locally, they are replicated to all configured peers that are currently online within the same distributed transaction (`replicateQueuedActions()`). If the transaction fails on one of the peers, all operations will be rolled back, including the locally performed ones. Further details about the replication have already been discussed in section 4.6.

Modules

A class diagram, covering all relevant classes and interfaces required by application developers, is shown in figure 5.2.

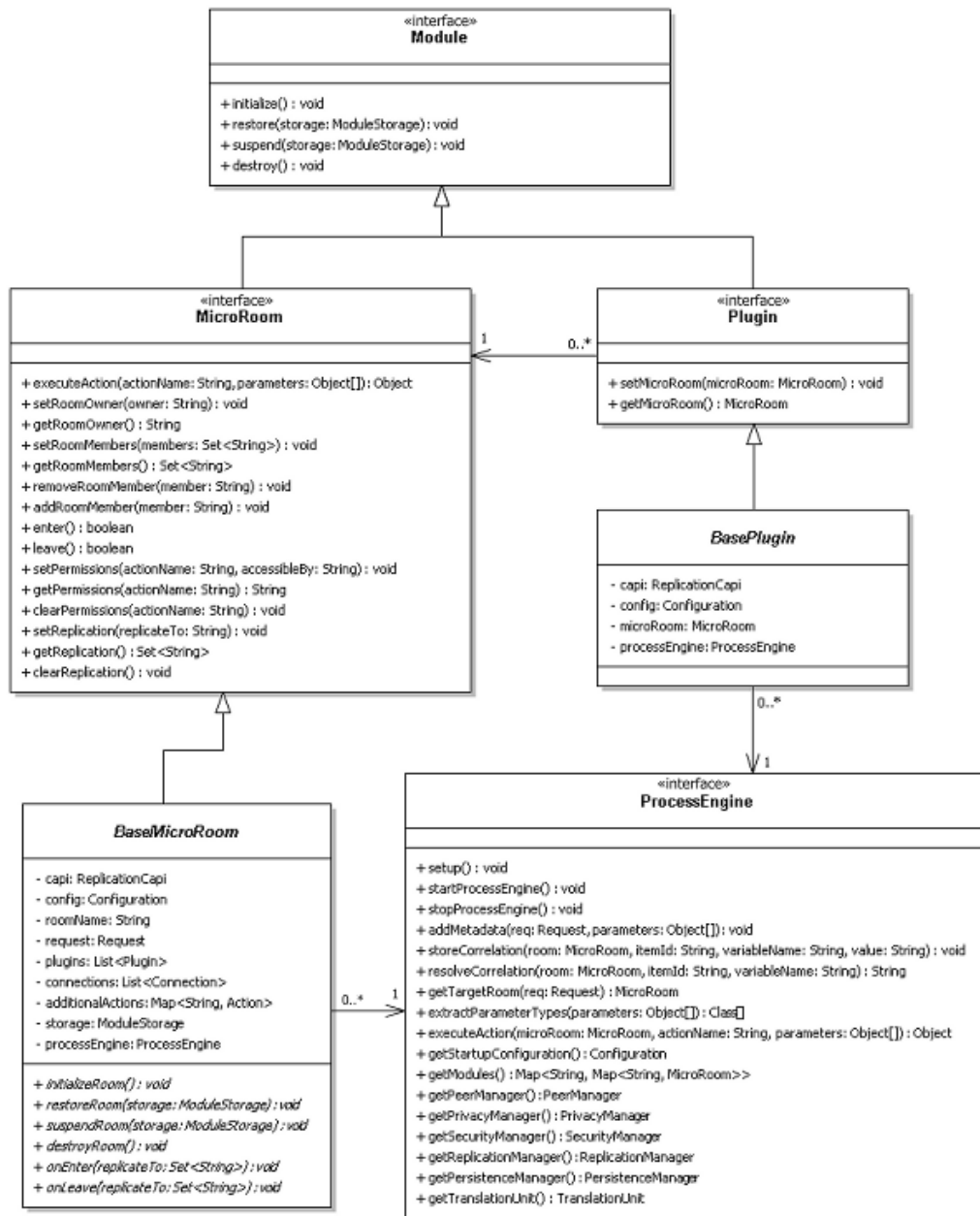


Figure 5.2: Class diagram - Modules

All modules are based on the `Module` interface that defines four important methods each module needs to provide. `Initialize()` is called only once, namely on the first framework startup when the module is created by the translation unit. `Restore()` will be called on each following framework startup by the persistence manager to restore the state of the module. This state will be saved on each shutdown via the `suspend()` method. The `ModuleStorage` object passed to both methods is unique for each micro-room and plugin. In the `restore()` method this object contains exactly the same data it has been assigned in the last `suspend()` call. Hence, it can be used to store volatile information, not stored within any container. In the background, these objects are stored by the persistence manager in the `Persistence Container` and thus persisted by XVSM Persistence. The `destroy()` method will be called if a module is explicitly destroyed (e.g. if a user deletes a chat room that has previously been created by him/herself). Please note that a module is never destroyed by the framework itself.

From the base interface the `MicroRoom` and the `Plugin` interface are derived. The `Plugin` interface only extends the `Module` interface by methods to access its parent `MicroRoom`, as each `Plugin` can only be contained within one `MicroRoom`. The most important method a `MicroRoom` needs to provide, is `executeAction()`. It will be called by the process engine to execute an action on the micro-room, as requested by the user. The other methods are used to change important meta data of the room, e.g. the room owner, room members, permissions of actions, or replication targets. Thereby, replication targets are identified via the unique user IDs and indicate to whom the whole room should be replicated. `enter()` and `leave()` are special methods which can be called by any allowed user to enter or leave the micro-room.

Based on these interfaces two base classes, i.e. `BaseMicroRoom` and `BasePlugin` are derived that implement common basic features. The application developers are urged to use these abstract classes rather than the interfaces, as they reduce the workload significantly. The `BaseMicroRoom` class implements all methods of `MicroRoom` and defines only few new abstract methods itself. All of them are called at the same point as their pendants (e.g. `initialize()` or `leave()`), but they already perform common tasks before or after the respective call (e.g. to also ensure the initialization of all plugins correlated to the room). More details about the structure of both `MicroRoom` and `Plugin` classes are described in section 6.2.

All of the created or restored modules are passed via a module `Map` to the `ProcessEngine`. On the other hand, each module has a reference of the currently running `ProcessEngine` instance. This allows access to many core features of the framework within the modules, e.g. to all managers, the startup configuration and other modules. Some of the core features of the `ProcessEngine` are its ability to execute actions (`executeAction()`) on arbitrary rooms and the correlation support. This is mandatory for complex workflows in order to know between which room instances actions should be performed (cf. section 4.7).

Objects

Finally, there are several objects which are important for the UI to communicate with the XVSM Micro-Room Framework (cf. figure 5.3).

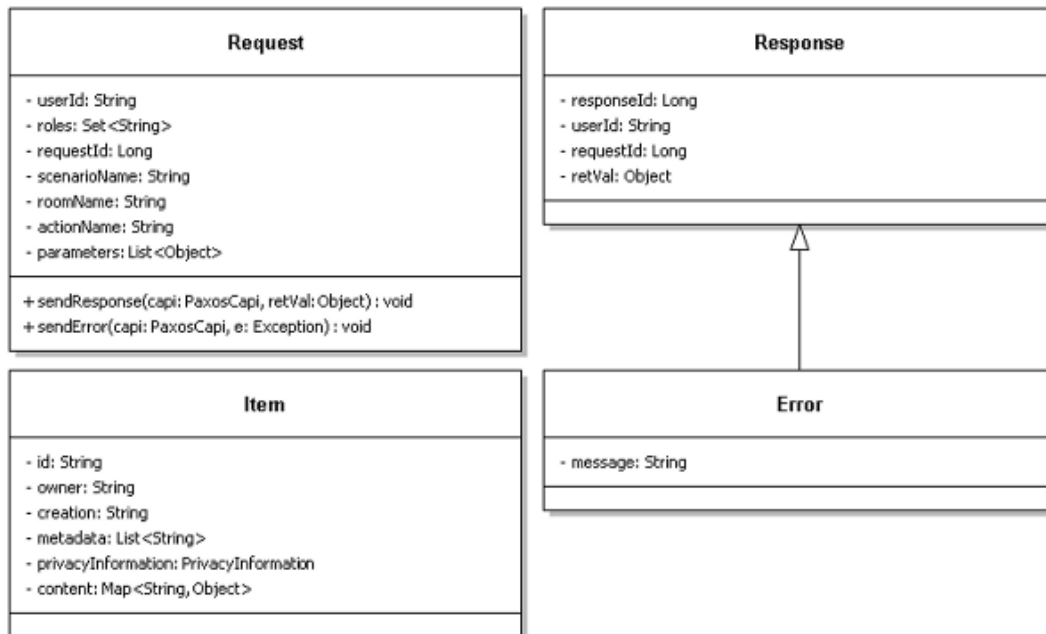


Figure 5.3: Class diagram - Objects

To execute an action, the UI needs to write a `Request` object via the REST interface into the Request Container (cf. figure 4.2). This object contains the `userId` and all roles of the user, which will be set automatically by the framework according to the SSO token embedded in the REST call. The target action needs to be specified via the `scenarioName` (i.e. the file name of the business logic file that contains the micro-room's configuration), the `roomName` on which the action should be called, as well as the `actionName` of the target action and the parameters needed. In order to retrieve an optional `Response` after the action has been completed, a `requestId` has to be provided.

If a response is expected, the UI will need to read a `Response` object from the Response Container. To obtain the right `Response` object, the UI has to provide the same `requestId` as with the initial `Request`. The `Response` object retrieved by the UI will then contain the return value (`retVal`) of the action called. If an error occurs, an `Error` object with the same `requestId` will be returned instead of a `Response` object.

Data itself (e.g. for parameters or return values) can either be specified via all common Java types, such as `Integer`, `Long`, `String`, `List`, and `Map`, or with a special data type, i.e. `Item`. Data encapsulated within an `Item` object will be supplemented with additional meta data from which it can benefit. These meta data covers an `id` unique among all peers, the `owner` and the `creation` date of the `Item`, as well as user-definable `metadata`. Additionally, `privacyInformation`, like the overall replication targets or all peers to which this item has already been replicated to, will be added by the privacy manager. The data itself is encapsulated within the `content` field.

5.1.3 Containers

Now that we have described the package structure as well as the most important classes and interfaces, we will give an overview about all XVSM containers used within the framework. Thus, we will outline the meaning of each container in the XVSM Micro-Room Framework, as well as the coordinators used and the entries stored during runtime (cf. table 5.2).

Container Name	Description	Coordinators	Entry
RequestC	Used for receiving Request objects from the UI in order to execute actions.	Fifo	Request object
ResponseC	Used for sending Response (including an action's return value) or Error objects (including an error message) to the UI.	Label reqId Label userId	Response object
CorrelationC	Used for storing and retrieving correlation information between actions.	Label itemId Label variable	value string
PermissionC	Contains all access-related information configured externally (e.g. via the Access plugin within the business logic files).	Label actionId Label userId Label role Label special-Permission	access string
DeniedUsersC	Contains a list of all users that are explicitly prohibited from accessing certain actions.	Label actionId Label userId	userId
RoomMemberC	Stores information about which users are members of which micro-rooms.	Label userId Label roomId	userId
RoomOwnerC	Stores information about which users are owners of which micro-rooms.	Label userId Label roomId	userId
ItemOwnerC	Stores information about which users are owners of which items.	Label userId Label itemId	userId
ReplicationC	Contains all replication targets configured externally (e.g. via the Replication plugin within the business logic files).	Label roomId	replication string
Replication-RequestC	Is used by the current ReplicationManager implementation to retrieve replication requests by other peers.	Key replReqId Query	Replication-Request object

Peer-CallbackC	Is used by the current PeerManager implementation to communicate with the peer coordination server. The peer coordination server writes and takes entries into/out of this container to inform this peer if other peers have gone on-/offline.	Key userId Label reqUser Fifo	userId -> space URI mapping
PrivacyC	Is used by the current PrivacyManager implementation to store privacy meta data for each item.	Key itemId Fifo	Privacy- Information object
PersistenceC	Is used by the current PersistenceManager implementation to store data from modules, that isn't stored within any of the above containers. This data is encapsulated within the ModuleStorage objects passed to each module's suspend() and restore() methods.	Fifo	ModuleStorage object

Table 5.2: XVSM Micro-Room Framework containers

5.2 Sequential View

After describing the structure, we will now give an alternative view onto the implementation, i.e. a sequential view. Therefore, we present sequence diagrams of the most common tasks in the following sections. These include framework startup, login, request handling, logout, and framework shutdown.

5.2.1 Startup

First of all, the operations performed at framework startup are important to understand the framework's characteristics in more detail. The corresponding sequence diagram is shown in figure 5.4.

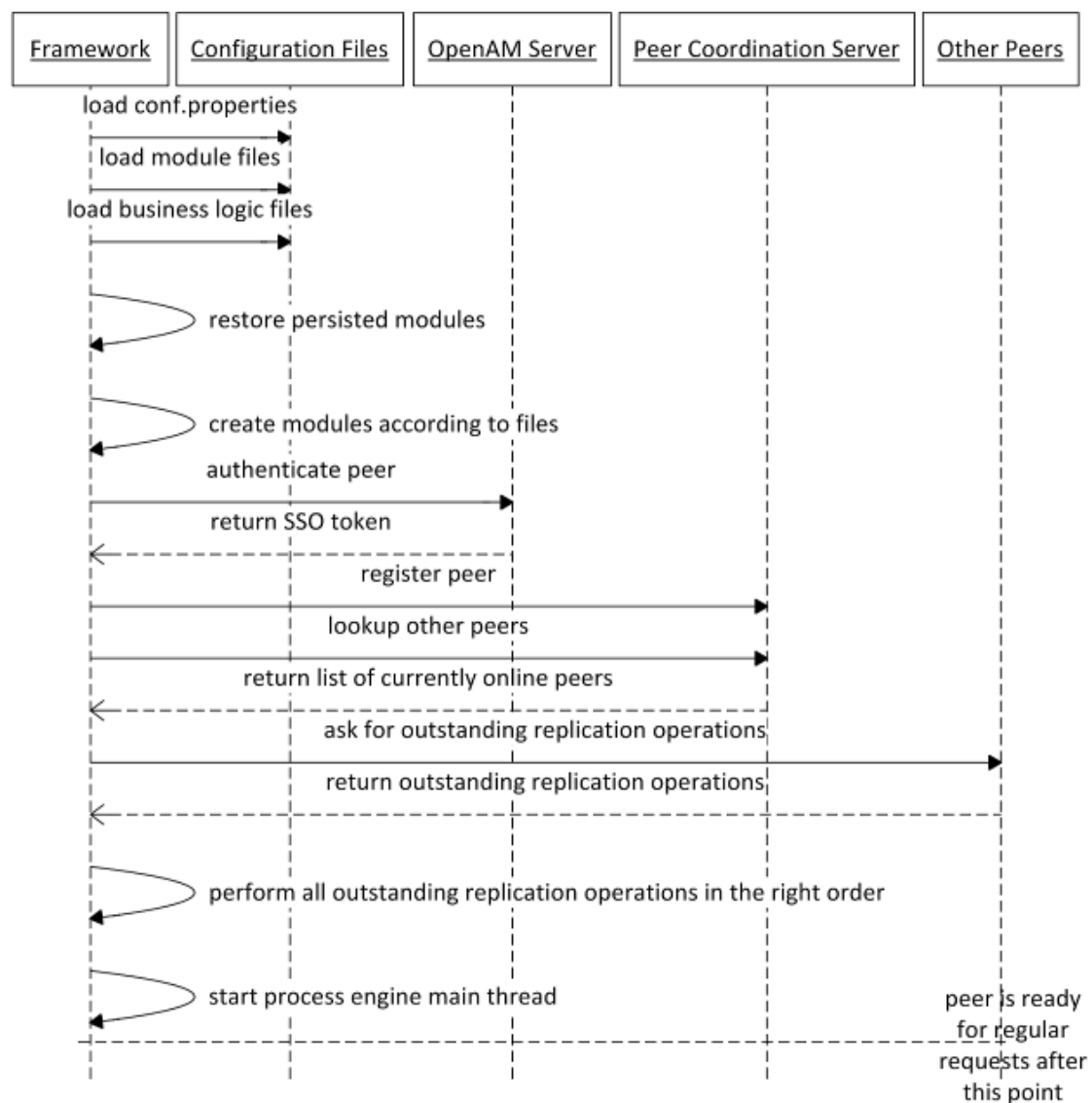


Figure 5.4: Sequence diagram - Startup

As can be seen, the framework first loads all relevant configuration files. It then restores all previously suspended modules by restoring all persisted containers and calling the `restore()` method on each module. If modules can't be restored, they will be created via the `TranslationUnit` according to the already loaded configuration files.

After that, the framework authenticates itself to the central authentication service (i.e. an OpenAM server in this case) with the credentials stored in the `config.properties`. The SSO token obtained is necessary for all following interactions with other peers, but also with the peer coordination server. In the next step, this server is informed that we are online now (cf. "register peer"). Afterwards, the framework retrieves a list of all other currently online peers,

which is kept up-to-date by the peer coordination server automatically from now on.

Then, all of the other online peers are asked whether we missed anything while we were offline. Thus, we send them information about which replication requests we have already handled and they will send us back all requests that we won't have already known. After receiving all answers, the framework removes duplicate replication requests, sorts, and executes them.

Afterwards, the process engine main thread is started, which is responsible for taking out all `Request` objects received by the UI and forwarding them to the executor threads. Thus, after this point the framework is up-to-date and fully operable, accepting requests from the UI and interacting with other peers.

5.2.2 Login

After the XVSM Micro-Room Framework has completed the startup sequence, the user will first need to log in to the UI, in order to prove that he/she is allowed to actually access the application implemented with the framework. In figure 5.5 and all upcoming sequence diagrams, we assume that the UI is implemented in HTML5 and located on a central web server.

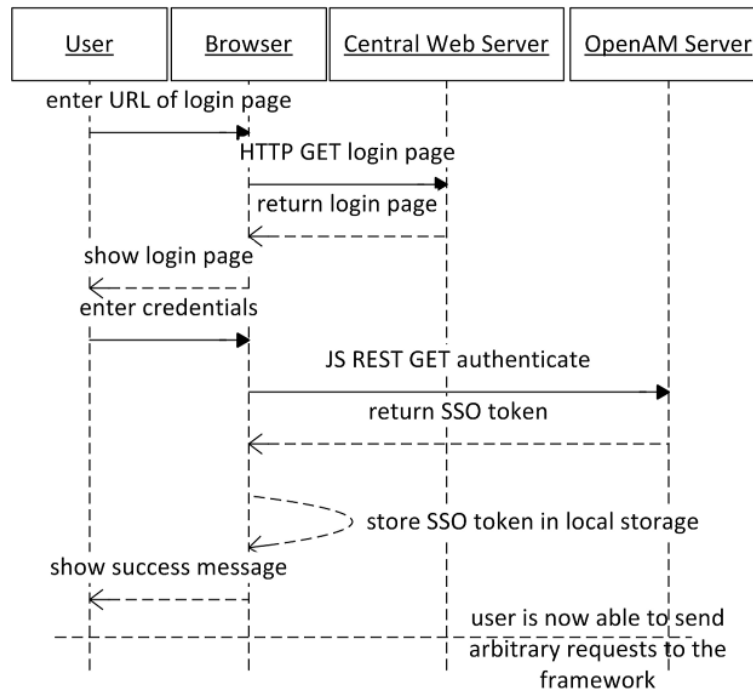


Figure 5.5: Sequence diagram - Login

The credentials entered into the UI are sent directly to the REST interface of the central authentication service to obtain a SSO token. This token should be saved locally by the UI. In the HTML5 example, the local storage of the browser is used to store it. As a remark, the local storage has been introduced with HTML5 and can be seen as an improved version of cookies. Now, the UI and thus the user is able to send arbitrary requests to the framework.

5.2.3 Requests

In figure 5.6, the typical flow of requests in a chat scenario is shown.

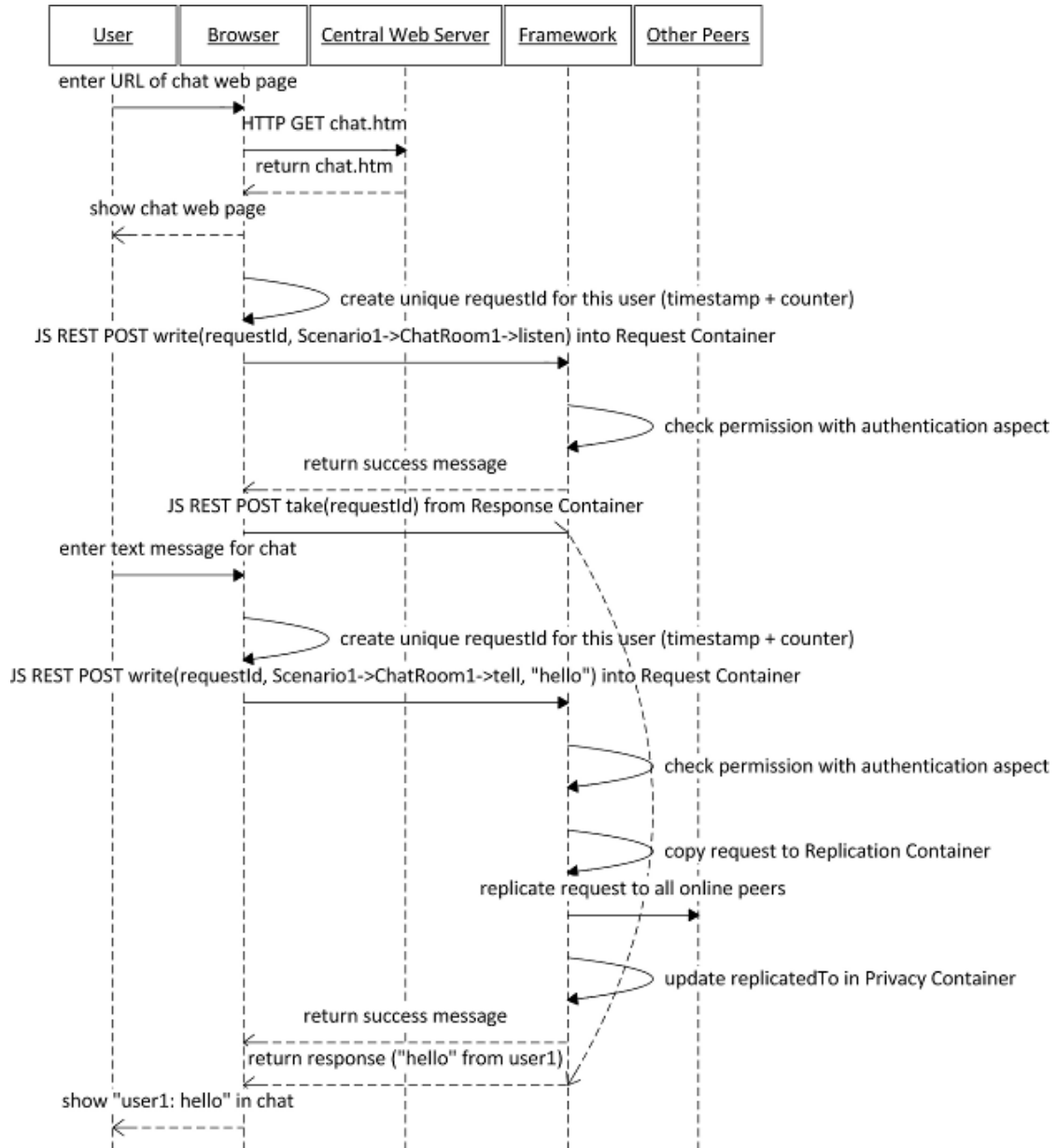


Figure 5.6: Sequence diagram - Request handling

Before the UI will be able to send a request to the framework, it first needs to create a request ID that is unique for each request of the peer. This is important to obtain responses for requests, as they are correlated with this ID. A good approach for generating the ID without any further

knowledge is to combine a timestamp with a local counter that is incremented after each request. Therefore, two equal request IDs are impossible, except if the local time is wrong or the same user sends two requests over two different UIs at the same time, which is highly unlikely.

In the example, the user first calls the `listen` action on the room `ChatRoom1` located in the scenario `Scenario1`. This action registers the user to receive all upcoming messages exchanged within the chat room. Those messages will be sent to the UI correlated with the request ID of the `listen` request. Hence, the UI will from now on be able to receive all messages by reading responses with this request ID. This is done asynchronously and thus doesn't block the UI. It is important to note that the UI needs to attach the SSO token retrieved during the login phase to each request. Otherwise the request would be rejected by the framework.

After that, the user enters a message into the UI and sends it to all other room members. This is done by calling the `tell` action of the chat room and passing the message text as a parameter. In contrast to the `listen` action, the `tell` action is annotated with the `Replicate` annotation, which means that each call of this action has to be replicated to all replication targets. This makes no sense for the `listen` action, as it doesn't influence other peers at all, but it is crucial for the `tell` action. Otherwise, the other peers wouldn't be aware of the new message, and thus the UIs of the other users couldn't show it. All replication targets that are currently online will receive a replication request immediately. Those that are offline will receive it during their synchronization phase at startup (cf. figure 5.4).

When the message has been sent, the logic of the chat room will eventually inform all currently registered listeners (i.e. users that called the `listen` action) about it. Thus, it writes a `Response` object into the `Response Container`, encapsulating the message text and the request ID of the initial `listen` action. This object will be read by the UI that will then show the message sent.

5.2.4 Logout

After the user has finished his/her work with the application, he/she needs to log out in order to close the session (cf. figure 5.7).

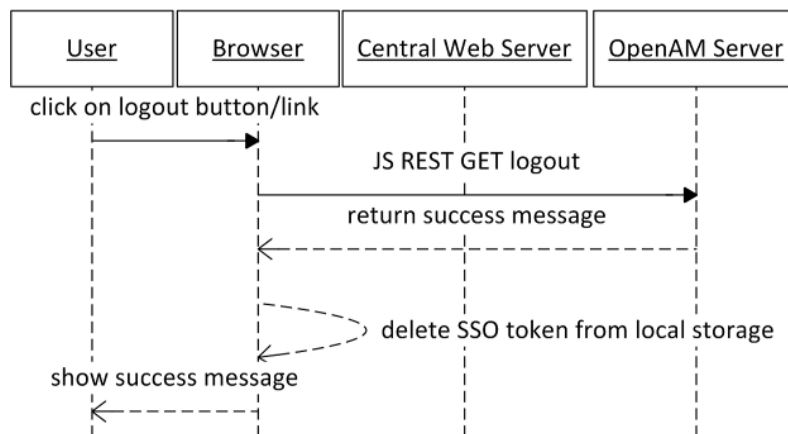


Figure 5.7: Sequence diagram - Logout

Similar to the login process, the logout process is performed via the REST interface of the central authentication service. The UI just has to pass the SSO token to this interface which thus invalidates it. After that, the UI deletes the saved SSO token from the local storage, as it is of no further use.

5.2.5 Shutdown

Finally, at some point in time the framework has to be shut down, which can be seen in figure 5.8.

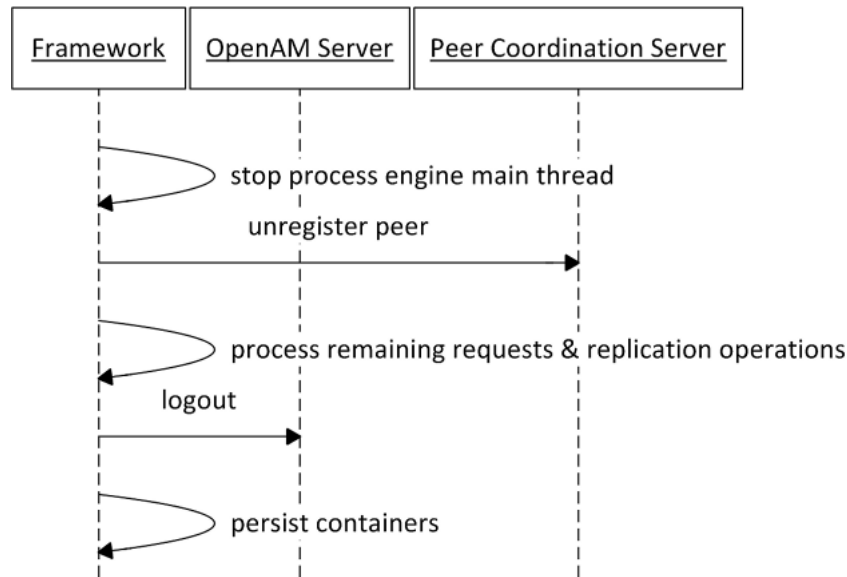


Figure 5.8: Sequence diagram - Shutdown

When the signal for termination is caught, the framework first stops the process engine main thread and unregisters itself at the peer coordination server. From this moment on, the framework will neither receive any further requests from the UI nor replication requests from other peers. Then, the remaining requests will be handled, followed by a logout of the framework itself at the central authentication service, thus invalidating its SSO token. Finally, the `suspend()` method of all modules will be called and the framework persists its internal containers (including all modules).

5.3 Implementation Details

Now that we have described the basic implementation, we will discuss chosen problems that occurred during implementation and how they have been solved.

5.3.1 Security

As security is one of the central requirements, it is important to properly protect the framework against malicious requests and unauthorized access. This is ensured by access rules that are defined via XVSM Security (cf. section 3.2.1). In the following, we will present the most important rules.

User Interface Interaction

First of all, we will discuss securing the interface over which the UI communicates with the framework. It is important to remark that all of the configured information about who is allowed to call which action, is read from the business logic files and stored within the `Permission Container` at startup. Additionally, there is the `Denied Users Container`, which can be used to blacklist specified users for certain actions, e.g. to block them from writing onto your wall if they molest you.

Based on these two containers a set of seven rules was defined that allows us to protect both the `Request Container` as well as the `Response Container` in a proper way. The rules all use the action ID contained in the request, as well as the ID and all roles of the user obtained by the SSO token. Thereby, the action ID is the unique identifier of an action, represented in the form of “scenario name -> micro-room name -> action name”, e.g. “Book Borrowing Scenario->Library->borrowBook”. These three variables will be called `aId`, `uId`, and `roles` in the following. Additionally, the name of the target room will be stored in `rName` and the ID of the passed item in `iId`. All of the variables are set in a pre-write aspect, registered at the `Request Container` and the `Response Container`. The other names that are mentioned in the following, correspond to the names of the coordinators that will be used. Their names all have been listed in section 5.1.3. The seven rules are described in the following:

Name: `request_deniedUsersRule`

Subjects: `all`

Resources: `Request Container`

Actions: `write`

Condition: `Denied Users Container | label(userId = uId) | label(actionId = aId)`

Scope: `-`

Effect: `DENY`

This rule denies all users that have explicitly been blacklisted for the requested action and thus overrules all of the other rules. Basically, it just selects all entries from the `Denied Users Container` with a matching user ID/action ID pair and if at least one is found, the rule will match.

Name: `request_userIdRule`

Subjects: `all`

Resources: `Request Container`

Actions: `write`

Condition: `Permission Container | label(userId = uId) | label(actionId = aId)`

Scope: -

Effect: PERMIT

The rule above allows access specified on a user ID level. Therefore, it checks whether at least one matching user ID/action ID pair exists in the `Permission Container`.

Name: `request_roleRule`

Subjects: all

Resources: `Request Container`

Actions: write

Condition: `Permission Container | query(role elementOf roles) | label(actionId = aId)`

Scope: -

Effect: PERMIT

Permitting access on a role level works a little bit different. As a user can be member of several roles, all of them will be contained within the passed SSO token. Thus, a simple comparison whether the passed roles are equal to the role(s) specified in the business logic file, is not sufficient. Otherwise, a user that is a member of the role “student” won’t be able to access an action that needs the role “student” or “teacher”, because the two strings “student” and “student,teacher” are not equal.

Hence, for each role specified in the business logic file, a separate entry is created in the `Permission Container`, e.g. one for “teacher” and one for “student”. The rule’s condition selects all entries of the `Permission Container` that contain at least one of the roles that the user actually is a member of and that also have the requested action ID. If at least one entry is fetched, the rule will match and thus access will be granted. Please note that the specified roles in the business logic files are thereby also connected with logical OR, equally to other combined access restrictions. So, at least one of the roles specified will be sufficient to access the action.

Name: `request_roomMembersRule`

Subjects: all

Resources: `Request Container`

Actions: write

Condition: `Permission Container | label(specialPermission = “ROOM_MEMBERS”) | label(actionId = aId) AND Room Member Container | label(userId = uId) | label(roomName = rName)`

Scope: -

Effect: PERMIT

The first of the three special permission rules checks whether there is an entry in the `Permission Container` that is labeled with both the special permission “ROOM_MEMBERS” (a constant) and the requested action ID. If so, it will also verify whether the requesting user is a member of this room by looking at the `Room Member Container`, which contains all room members as user ID/room name pairs.

Name: request_roomOwnerRule
Subjects: all
Resources: Request Container
Actions: write
Condition: Permission Container | label(specialPermission = "ROOM_OWNER") | label(actionId = aId) AND Room Owner Container | label(userId = uId) | label(roomName = rName)
Scope: -
Effect: PERMIT

Similar to the first special permission rule, the second one verifies whether there is an entry in the Permission Container that is labeled with both the special permission "ROOM_OWNER" (a constant) and the requested action ID. Additionally, it validates whether the requesting user is the owner of this room by looking at the Room Owner Container, which contains all room owners as user ID/room name pairs.

Name: request_itemOwnerRule
Subjects: all
Resources: Request Container
Actions: write
Condition: Permission Container | label(specialPermission = "ITEM_OWNER") | label(actionId = aId) AND Item Owner Container | label(userId = uId) | label(itemId = iId)
Scope: -
Effect: PERMIT

The last special permission rule checks whether there is an entry in the Permission Container that is labeled with both the special permission "ITEM_OWNER" (a constant) and the requested action ID. It also examines if the requesting user actually is the owner of the item that is passed as an argument to the action by looking at the Item Owner Container, which contains all item owners as user ID/item ID pairs.

These six rules are sufficient to secure the Request Container. It might be possible to reduce this number even more, however, this would decrease either the understandability or the flexibility in contrast to the proposed rules.

Name: response_userIdRule
Subjects: all
Resources: Response Container
Actions: take
Condition: -
Scope: label(userId = uId)
Effect: PERMIT

To secure the `Response Container`, only one, relatively simple rule is required. It ensures that each requesting peer can only read responses for requests that have been initiated by him/herself. Due to the defined scope, only entries labeled with the user ID that has been provided with the passed SSO token will be returned. All the other responses are completely invisible to this peer.

Replication Propagation

As has been described in section 4.6, we use a special `Replication Request Container` for inter-peer communication. This container is secured exactly in the same way as the `Request Container`, i.e. with a copy of the first six rules presented in the section above. Therefore, it can be ensured that other peers can only replicate actions that they are actually allowed to call (in terms of the configuration). Hence, peers can't forge replication requests in a harmful way.

Please note that also impersonation of other peers is impossible, since the identity of a peer is validated via the SSO token attached to each replication request. This SSO token is retrieved by the framework itself according to the credentials stored within its `config.properties` file (cf. section 5.2.1). A forged SSO token will be recognized immediately, thereby skipping the corresponding request.

Peer Management

The last area that might be vulnerable is the peer coordination server itself as well as all the corresponding communication. Concerning the peer coordination server's architecture, the most important container is the `Peer Container`. It has registered a `FIFO` as well as a `Key Coordinator`, representing the user ID. During runtime the container includes "user ID -> space URI" mappings of all peers that are currently online. Again, the user ID of the SSO token is stored in the variable `uId` in a pre-write, pre-take and pre-read aspect on this container. Additionally, the pre-write aspect checks whether the user ID passed in the entry is equal to the user ID specified in the `Key coordinator`.

Referring to section 5.2, each peer stores such a mapping into this container at startup and removes it again on shutdown. It is important to ensure that no other peer can write a forged entry into the container to receive replication requests not intended for him/her. On the other hand, other peers should also be prevented from removing entries, as this would set the target peer offline to all the other peers. Both problems are taken care of by the following access rule:

Name: `writeTakePeerRule`

Subjects: `all`

Resources: `Peer Container`

Actions: `write, take`

Condition: `-`

Scope: `label(userId = uId)`

Effect: `PERMIT`

Similar to the rule of the `Response Container`, it will only allow to write or take entries that are labeled with the same user ID as contained in the passed SSO token. Therefore, forged entries or disallowed deletion of entries are prevented.

After a peer has registered, a list of all other currently online peers is fetched. To hinder attackers from getting this sensitive data (in terms of privacy), the following rule ensures that only those peers can read from the `Peer Container` that are already registered at the peer coordination server, i.e. their user IDs are contained within this container:

Name: `readPeerRule`
Subjects: `all`
Resources: `Peer Container`
Actions: `read`
Condition: `Peer Container | label(userId = uId)`
Scope: -
Effect: `PERMIT`

Besides the `Peer Container` located on the peer coordination server, every peer has a `Peer Callback Container`. This container is used by the peer coordination server to inform registered peers about other peers that have gone on- or offline. The following simple rule prevents other peers from manipulating this important container:

Name: `writeTakeCallbackRule`
Subjects: `userId = p2p-server`
Resources: `Peer Callback Container`
Actions: `write, take`
Condition: -
Scope: -
Effect: `PERMIT`

It uses the subject attribute to permit actions only to the user with the ID “p2p-server”. This ID is reserved for the peer coordination server.

As can be seen, all containers accessible from the outside have been secured, and access to the other containers is denied completely. Thus, a decent level of security is ensured.

5.3.2 JavaScript Wrapper Library

The JavaScript Wrapper Library encapsulates all calls from a HTML-based UI to the REST API of the XVSM Micro-Room Framework. Details about its configuration and usage are presented in section 6.3.2. Two special problems concerning UIs generated with the JavaScript wrapper library and HTML5 are described in the following.

Same-Origin-Policy Restrictions

The *Same Origin Policy* (SOP) is a security concept of JS that prevents access to resources belonging to locations different from the originating location. This means that e.g. a JS script running on `my.server.com` cannot embed a HTML file located on `your.server.com` as those two locations differ from each other. This is true even if it's just a different sub domain or port.

Without the SOP, it would – for example – be possible to access local session cookies, stored for other sites. In our case, the SOP will disallow REST calls from the JavaScript wrapper library to the local framework if the UI is hosted on a central web server (e.g. for maintainability reasons).

There are several possibilities to circumvent the SOP as described in [SB11]. One approach is to use *JSON with Padding* (JSONP), which loads an external JS file via the `<script>` tag. This tag is excluded from the SOP and thus can be used for communication with a different domain. Thus, one could embed query parameters within the URL to define the request. The target of the URL is of course no static JS file (as intended), but a web service generating dynamic responses, based on the query parameters. This allows to establish some kind of communication channel. More information about JSONP can be found under [19].

Another method is to use the *postMessage* interface [20] which enables communication between different tabs within the same browser window. So, if you open the URL to the UI files in one tab and the URL to the locally running framework instance in the other one, communication between those two tabs and thus their underlying resources will be possible.

However, both of these methods can be seen as dirty hacks, as they use functions in a way they were not intended to be used. Also, they are impractical in our case, as they don't provide transparent support for all *Hypertext Transfer Protocol* (HTTP) methods (GET, POST, PUT, DELETE) that are needed for the REST calls. It would only be possible to somehow rebuild GET functionality. Considering the second approach, automatically opening a second tab to communicate with `localhost` is completely infeasible, as the common user wouldn't understand this odd behaviour of his/her browser.

Fortunately, another approach to overcome the limits of SOP exists, namely *Cross-Origin Resource Sharing* (CORS) [21]. Thereby, the web server sends a specific header, that specifies which other sites are allowed to use resources from its domain. The only crucial part is that the browser needs to support CORS, i.e. it needs to understand this header. Fortunately, today all major browsers support this technology, including Firefox 3.5, Safari 4, Google Chrome 4, Opera 12, Internet Explorer 8, iOS Safari 3.2, and Android Browser 2.1 [22].

By using this method, all HTTP methods can be used on external resources transparently. Also, it is no dirty hack, but intended to be used when resources from different domains are needed within one application. That's why it seems to be the best solution. In our case, we adjusted the Jetty server used by the XVSM's REST implementation to allow access from other domains. As the REST API is secured thoroughly as described in section 5.3.1, this doesn't cause any security issues. The additional headers sent by the Jetty server (and thus the framework instance) are shown in listing 5.1. This solves the SOP problem for all browsers that understand the header.

```

1 Access-Control-Allow-Origin: *
2 Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
3 Access-Control-Allow-Headers: Content-Type, X-requestId, isolation

```

Listing 5.1: CORS Header sent by the framework

File Handling

Additionally, file handling needs special treatment. Therefore, an own micro-room type (i.e. `FileRoom`) exists, that stores files in the local file system instead of in containers (and thus the persistence database). For exchanging files with the UI, this room offers a `write`, `read` and `delete` action. The `write` action has a byte array as argument that represents the file, whereas the `read` action returns the same byte array again.

Now, theoretically it would be possible for the UI to access the files directly from the local directory where the framework stores them. However, this circumvents the defined communication channels and would violate the framework's design. Thus, a possibility needed to be found how the browser could show files that are not stored on any web server, but only received as a byte array. Again, fortunately there is a way to do this, i.e. by using Blobs [23].

Blobs are in-memory objects with a unique ID that represent files. They can be constructed by using raw data such as a byte array. As the interface is rather complex and depending on the browser, we encapsulated the whole functionality within the JavaScript wrapper library (cf. section 6.3.2 for detailed usage).

In figure 5.9, you can see the whole download process of a file from a framework instance in detail. The file is retrieved by calling the `read` action of the `FileRoom`. The room reads the file from the hard disk and extracts its byte array. This is then forwarded to the REST interface which automatically Base64 encodes it. In the browser, the JavaScript Wrapper Library Base64 decodes the received byte array and creates an in-memory Blob from it. The URL of this object can be used as a normal URL, e.g. to show an image or to provide a download link for a file.

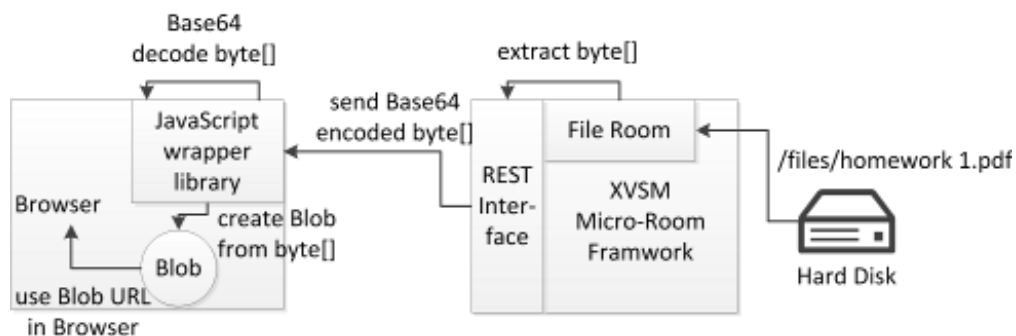


Figure 5.9: File download process

The upload of a file to a framework instance is visualized in figure 5.10. The file is chosen in the browser's common file chooser dialog and forwarded to the JavaScript wrapper library.

There the byte array of the file is extracted and Base64 encoded. This content is then passed via the REST interface to the `write` action of the `FileRoom`. In the REST interface, the byte array is Base64 decoded. Finally, the decoded byte array is written to the hard disk as a new file by the room.

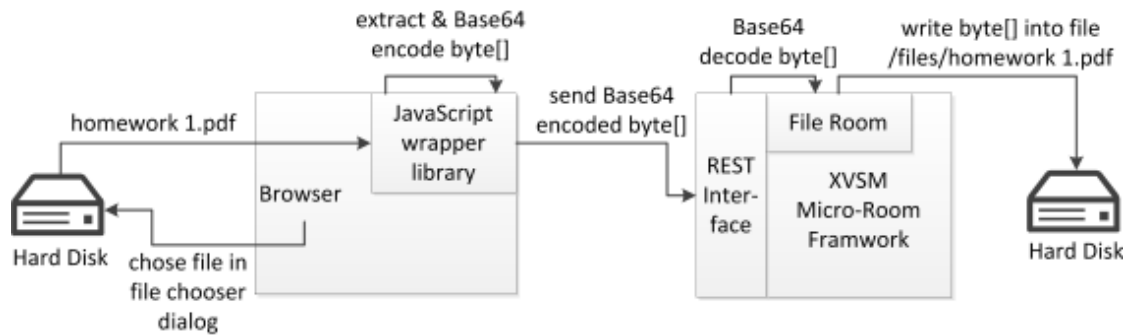


Figure 5.10: File upload process

Blobs are not supported by as many browser versions as CORS, but still by all modern browsers, i.e. Firefox 6, Safari 6, Google Chrome 8, Opera 12.1, Internet Explorer 10, iOS Safari 6.0, and Android Browser 3.0 [24]. So, to use the full set of functionality of the JavaScript wrapper library, at least the browser versions listed above are required.

Deployment

After dealing with the implementation-related parts of the XVSM Micro-Room Framework, we will now describe how it can be used by end users, application developers, and UI designers to create P2P applications. Also, the deployment possibilities of applications developed with the framework will be discussed in this chapter.

6.1 End User Manual

Due to its modularity and simplicity, the XVSM Micro-Room Framework allows end users to configure the business logic of the application by specifying business logic files in the XML format. Additionally, end users can configure the basic settings of the framework by modifying the `configuration.properties` file. Both file types are described in detail in the following.

6.1.1 Configuration File

The configuration file `config.properties` needs to be located in the root directory of the framework. It is used for configuring the framework itself and to set application-wide settings. An example file can be seen in listing 6.1:

```
1 module.loader=org.xvsm.microroom.framework.loaders.module.  
   ↳RemoteModuleLoader  
2 module.remote.url=https://www.geek-world.at/xmrf/modules  
3 module.remote.username=test  
4 module.remote.password=password  
5  
6 business-logic.loader=org.xvsm.microroom.framework.loaders.  
   ↳businesslogic.RemoteBusinessLogicLoader  
7 business-logic.remote.url=https://www.geek-world.at/xmrf/business-  
   ↳logic  
8 business-logic.remote.username=test  
9 business-logic.remote.password=password
```

```

10
11 peer.lookup.url=xvsm://my.lookup-server.com:50330
12 peer.userId=joe
13 peer.password=passwordjoe
14
15 openAm.query.userId=amAdmin
16 openAm.query.password=passwordamAdmin

```

Listing 6.1: Sample configuration file - config.properties

Table 6.1 describes all possible configuration entries:

Key	Possible Values	Description
module.loader	org.xvsm.microroom.framework. ↔loaders.module. ↔RemoteModuleLoader <i>or</i> ↔LocalModuleLoader	Defines the class of the module loader that should be used (cf. section 4.2.2).
module.remote.url	URL	Only required if RemoteModuleLoader is used. Specifies where to look for the modules. Target needs to be a directory listing generated via Apache web server.
module.remote.username	HTTP Access user name	Only for RemoteModuleLoader if the URL is protected with HTTP Basic Authentication.
module.remote.password	HTTP Access password	Only for RemoteModuleLoader if the URL is protected with HTTP Basic Authentication.
business-logic.loader	org.xvsm.microroom.framework. ↔loaders.businesslogic. ↔RemoteBusinessLogicLoader <i>or</i> ↔LocalBusinessLogicLoader	Defines the class of the business logic loader that should be used (cf. section 4.2.3).
business-logic.remote.url	URL	Only required if RemoteBusinessLogicLoader is used. Specifies where to look for the business logic files. Target needs to be a directory listing generated via Apache web server.

business-logic.remote. ↪username	HTTP Access user name	Only for RemoteBusinessLogicLoader if the URL is protected with HTTP Basic Authentication.
business-logic.remote. ↪password	HTTP Access password	Only for RemoteBusinessLogicLoader if the URL is protected with HTTP Basic Authentication.
peer.lookup.url	URL starting with xvsm://	Defines the location of the peer coordination server.
peer.userId	OpenAM user ID of this peer	User ID used for authenticating the framework instance itself at the central authentication service.
peer.password	OpenAM password of this peer	Password used for authenticating the framework instance itself at the central authentication service.
peer.upload.dir	folder name	Defines the folder in which all files uploaded to this peer are stored. If not defined, “files” is used.
peer.persistence.dir	folder name	Defines the folder in which the persisted state of the framework is stored on shutdown. If not defined, “db” is used.
openAm.query.userId	OpenAM user ID with query permissions	User ID used by the framework to perform query operations on the central authentication service (e.g. to retrieve all roles of an user).
openAm.query.password	OpenAM password with query permissions	Password used by the framework to perform query operations on the central authentication service (e.g. to retrieve all roles of an user).

Table 6.1: All possible entries for config.properties

Since it is just a text file, the `config.properties` file can be changed easily by the end user. However, usually this won’t be necessary as the file is rather static and thus can be pre-configured by the application developer. The only varying keys are `peer.userId` and

`peer.password` that need to be adjusted for every user. But this is no problem if you consider that an application website exists from which every user can download the whole client software package needed to participate. There, the application developer could, e.g. write a script that customizes these properties according to the credentials the user is currently logged in with at the website. Hence, every user can download his/her personalized client software package.

6.1.2 Business Logic File

Business logic files are used to configure and connect micro-rooms. This can be done via XML files that are structured as simple as possible. Therefore, even end users should be able to configure such a file on their own. These files need to be either stored directly in the `business-logic` folder on the peer, or at a central web server from which the peers automatically fetch the files on startup (cf. `RemoteBusinessLogicLoader`).

The exact *XML Schema Definition* (XSD) of a business logic file is shown in appendix A. It has been visualized in figure 6.1 for better understandability.

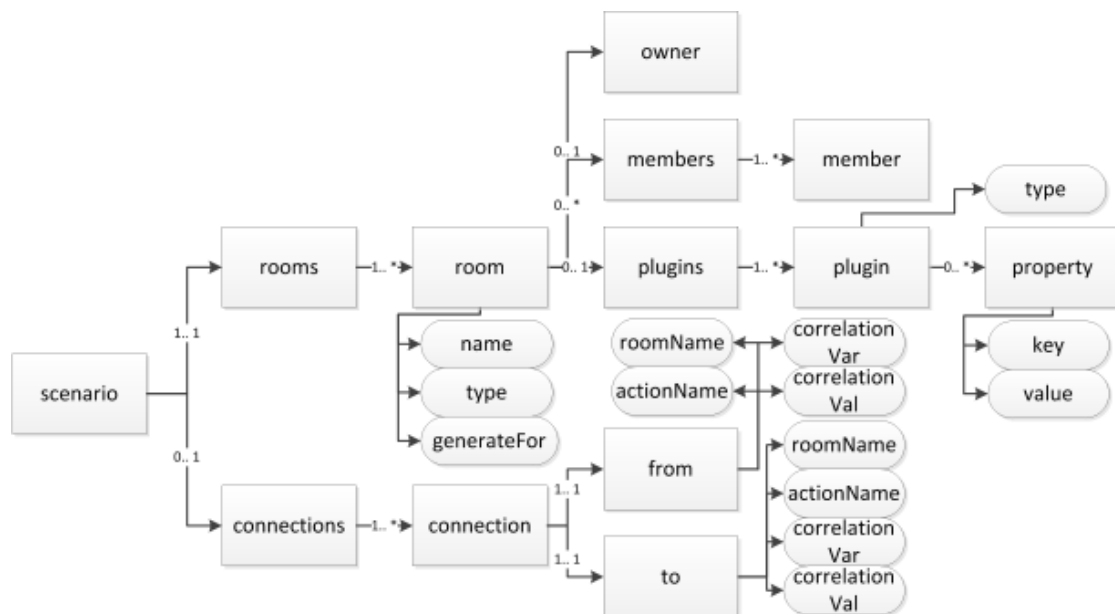


Figure 6.1: Visualized XSD of business logic files

The root node of the file is the `scenario` which has to contain at least one `room` (itself contained within the `rooms` node). The other nodes are all optional to, e.g. pre-define the room owner, room members, or set connections. A room can contain several `plugin` nodes, encapsulated within a `plugins` parent node. Both `room` and `plugin` nodes are used to configure the modules themselves, whereas `connection` nodes allow to connect actions between rooms to generate a workflow. Thus, they contain a single `from` and a single `to` node to specify the desired rooms, actions, and correlation settings (cf. section 4.7). The attributes (round shapes) will be described in detail later on.

In listing 6.2 a sample business logic file is shown:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <scenario>
3   <rooms>
4     <room name="Chat Room" type="org.xvsm.microroom.modules.rooms.
        ↳ConferenceRoom" generateFor="$none" getOfflineMessages="true">
5       <plugins>
6         <plugin type="org.xvsm.microroom.modules.plugins.Replication">
7           <property key="replicateTo" value="$roomOwner, $roomMembers" />
8         </plugin>
9         <plugin type="org.xvsm.microroom.modules.plugins.Access">
10          <property key="tell" value="$roomOwner, $roomMembers" />
11          <property key="listen" value="$roomOwner, $roomMembers" />
12          <property key="stopListening" value="$roomOwner, $roomMembers"
        ↳/>
13        </plugin>
14        <plugin type="org.xvsm.microroom.modules.plugins.Creatable">
15          <property key="actionName" value="create" />
16          <property key="accessibleBy" value="role.student,role.teacher"
        ↳/>
17        </plugin>
18        <plugin type="org.xvsm.microroom.modules.plugins.Closable">
19          <property key="actionName" value="close" />
20          <property key="accessibleBy" value="$roomOwner" />
21        </plugin>
22      </plugins>
23    </room>
24    <room name="Chat Room Overview"
25    type="org.xvsm.microroom.modules.rooms.Reception">
26      <plugins>
27        <plugin type="org.xvsm.microroom.modules.plugins.Replication">
28          <property key="replicateTo" value="role.student,role.teacher" />
29        </plugin>
30        <plugin type="org.xvsm.microroom.modules.plugins.Access">
31          <property key="add" value="role.student,role.teacher" />
32          <property key="remove" value="$itemOwner" />
33          <property key="lookup" value="role.student,role.teacher" />
34        </plugin>
35      </plugins>
36    </room>
37  </rooms>
38  <connections>
39    <connection>
40      <from roomName="Chat Room" actionName="create" correlationVar="A"
        ↳correlationVal="$params[0]" />
41      <to roomName="Chat Room Overview" actionName="add" />
42    </connection>
43  </connection>
```



```

44 <from roomName="Chat Room Overview" actionName="remove" />
45 <to roomName="Chat Room" actionName="close" correlationVar="A" />
46 </connection>
47 </connections>
48 </scenario>

```

Listing 6.2: Sample business logic file - Chatting.xml

The example above corresponds to the model visualized in figure 6.2. As can be seen, the graphical notation is even a lot more user-friendly than the already simple XML file. So, a graphical modeling tool to create business logic files would even improve simplicity of the framework. However, this is out of scope in this work and will be a matter of future work.

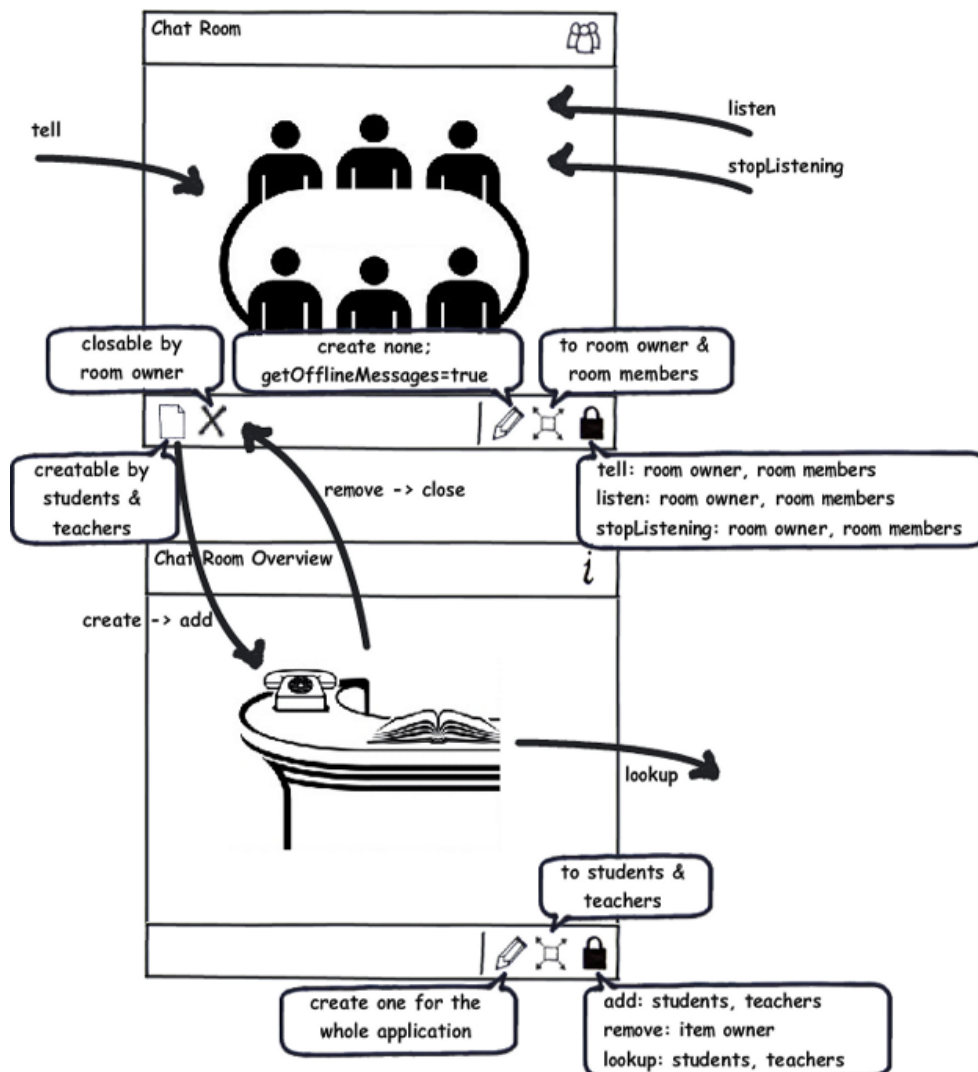


Figure 6.2: Visualized sample business logic file

In the example, there are two different rooms called “Chat Room” and “Chat Room Overview”. As can be seen in line 4, the chat room is a micro-room of type `ConferenceRoom`, which has to be provided by the application developer (cf. section 6.2). The attribute `generateFor` defines how much instances of this room should be created by the framework. In this case, no room will be created, as the chat room will only serve as a model room from which the user can create copies during runtime.

The attribute `getOfflineMessages` is a custom attribute defined by the application developer and will be forwarded to the `ConferenceRoom` implementation. It basically tells the micro-room that it should collect all messages sent by other peers while the user is offline. If it was false, the user would only receive messages that were sent while he/she was online.

Concerning the plugins, the `Replication` plugin defines whom this room should be replicated to. More specifically, it defines to which peers (replicable) actions called on this room should be replicated to. In the example, they will be replicated to the room owner as well as all room members (cf. line 7). A plugin allows to specify arbitrary properties, which are all forwarded to the plugin’s implementation, similar to custom attributes in micro-rooms.

The `Access` plugin allows to specify access restrictions on each action of the room. Thus, for each action a `property` node needs to be specified, containing the action name in its `key` attribute and the access string in its `value` attribute. In lines 10 to 12 access to the three actions `tell`, `listen`, and `stopListening` is only permitted to all students and teachers. Access to all the other actions defined in the `ConferenceRoom` implementation will be denied.

As already indicated, the chat room is only a model room from which new instances can be created by the user during runtime. This is enabled via the `Creatable` plugin, which adds a new action to the room that allows to create a new instance. In line 15 the name of this action is defined, whereas line 16 specifies the access string, similarly to the `Access` plugin. In this case, every student and teacher is allowed to create new chat rooms.

The counterpart to the `Creatable` plugin is the `Closable` plugin that allows to ultimately destroy a room. Just as the `Creatable` plugin, it thus adds a new action to the room, which is called `close` (cf. line 19). This is also a good example for an action whose access should be restricted to only the room owner him/herself, as defined in line 20.

As new chat rooms can be created and destroyed dynamically during runtime, this information has to be stored somewhere, so that the users can ask which rooms exist. Therefore, a `Reception` is defined in line 25. This time it will be sufficient if only one application-wide room exists. Hence, the `generateFor` attribute doesn’t need to be specified, as its default value is `$application`, i.e. that only one room instance is generated for the whole application.

This room contains only meta data about which chat rooms currently exist and thus will be replicated to all students and teachers (line 28). All students and teachers are allowed to add new room entries and to query all of them (line 31 and 33), but only the owner of such an entry is allowed to remove it (line 32).

In order to fill this room with the desired information (e.g. add a new entry if a new chat room is created and remove the right entry again if a chat room is closed), we use connections. In lines 40 and 41, we connect the `create` action of the model `ConferenceRoom` with the `add` action of the `Reception`. Thus, the return value of the `create` action (i.e. an item

containing the user-defined name of the newly created chat room instance) will be passed as argument to the `add` action, which stores this information in a way accessible for all peers. The important thing to mention is that this connection will store a correlation variable `A` according to the first parameter that is passed to the `create` action, i.e. the user-defined name of the chat room instance to create.

The second connection, defined in lines 44 and 45, connects the `remove` action of the `Reception` to the `close` action of the `ConferenceRoom`. So, if the owner of a chat room removes the information about his/her room from the `ConferenceRoom`, the room will be closed automatically. Please note that this connection could also be defined in the opposite direction. Also, direct calls on `close` are possible in this scenario and could thus lead to data inconsistencies. However, this could be prevented by defining access restrictions on this action.

For the connection to operate properly, it is important that the correlation variable `A` is again specified, so that the framework knows which chat room instance should be closed. This works, since the correlation variable stored the name when the chat room was created and the item ID is exactly the same within the `add` and the `remove` action.

After explaining the example, table 6.2 summarizes all commonly used attributes and their possible values:

Attribute	Possible Values	Description
<code>room.name</code>	arbitrary string	Defines the name of this micro-room instance. Needs to be unique within the scenario.
<code>room.type</code>	class name of the micro-room implementation class	Defines the type of the micro-room. Needs to correspond to the class name defined by the application developer (cf. section 6.2.1).
<code>room.generateFor</code>	<code>\$none</code> <code>\$application</code> <code>role.roleName</code> <code>user.userId</code>	Defines how many room instances should be generated, i.e. either none, one for the whole application, one for each member of a specified role, or one for each user specified.
<code>room.customAttribute</code>	attribute defined by the micro-room's API	Any further attributes within the <code>room</code> node will be forwarded directly to the room's implementation class, thus allowing custom configurations (cf. section 6.2.1).

plugin.type	class name of the plugin implementation class	Defines the type of the plugin. Needs to correspond to the class name defined by the application developer (cf. section 6.2.2).
plugin.customAttribute	attribute defined by the plugin's API	Any further attributes within the plugin node will be forwarded directly to the plugin's implementation class, thus allowing custom configurations (cf. section 6.2.2).
plugin.property.key	key defined by the plugin's API	Key of a plugin-specific key-value pair. Should be preferred over custom attributes for better readability.
plugin.property.value	value defined by the plugin's API	Value of a plugin-specific key-value pair. Should be preferred over custom attributes for better readability.
connection.from/to. ↪roomName	name of a room within this scenario	Defines the source/target room of this connection.
connection.from/to. ↪actionName	name of an action within the specified room	Defines the source/target action of this connection
connection.from/to. ↪correlationVar	arbitrary string	Name of the correlation variable that should be used (cf. section 4.7).
connection.from/to. ↪correlationVal	<code>\$params[index]</code> <code>\$params[index].owner</code> <code>\$params[index].content.key</code> <code>constantVal</code>	Specifies which value should be stored in the correlation variable, i.e. either the defined parameter passed to the action (in string representation), its owner, a defined content field (both only if the parameter is an <code>Item</code>), or a constant string value.

Table 6.2: All relevant attributes of a business logic file

Concerning the `Access` and `Replication` plugin, the following access/replication strings are possible:

\$roomMembers: Action is accessible by/will be replicated to all room members.

\$roomOwner: Action is accessible by/will be replicated to the room owner.

\$itemOwner: Action is accessible by the item owner. Doesn't make sense for replication.

role.roleName: Action is accessible by/will be replicated to all members of the defined role.

user.userId: Action is accessible by/will be replicated to the specified user.

6.2 Application Developer Manual

The implementation-related work that has to be done for extending the framework's functionality is described in this section. Therefore, the application developer has to provide the implementations of all micro-rooms and plugins that are used within the business logic files. This is done by implementing them as Java classes, compiling them to `class` files and packaging these `class` files into a JAR file. This file can then either be stored locally in the peer's `modules` folder or on a web server from which the peers automatically fetch them on startup (cf. `RemoteModuleLoader`).

6.2.1 Micro-Room File

Basically, a micro-room can simply be defined by a class that implements the `MicroRoom` interface. To further ease the implementation, application developers are urged to extend their class from the abstract `BaseMicroRoom` class instead. This class implements a lot of common features, such as initiation, restore, suspend, and shutdown procedures, as well as general member, permission, and replication management. An example for such an implementation is shown in listing 6.3.

```
1 public class Reception extends BaseMicroRoom{
2     private ContainerReference recC;
3
4     @Override
5     public void initializeRoom() {
6         ...
7         recC = capi.createContainer(roomName + "_ReceptionC",
8             ↪obligatoryCoords, optionalCoords);
9     }
10
11     @Override
12     public void restoreRoom(ModuleStorage storage){
13         ...
14         recC = capi.lookupContainer(roomName + "_ReceptionC");
15         ...
16     }
17
18     @Override
19     public void suspendRoom(ModuleStorage storage){}
20
21     @Override
```

```

22 public void destroyRoom() {
23     ...
24     capi.destroyContainer(recC);
25     ...
26 }
27
28 @Override
29 public void onEnter(Set<String> replicateTo) {
30     List<Item> items = (List<Item>) processEngine.
        ↳ getReplicationManager().callRemoteAction(roomName, "lookup",
        ↳ null, replicateTo);
31     add(items);
32 }
33
34 @Override
35 public void onLeave(Set<String> replicateTo) {
36     remove(lookup());
37 }
38
39 @Replicate
40 public void add(Item room) {
41     try {
42         String roomId = room.getId();
43         Entry e = new Entry(room, FifoCoordinator.newCoordinationData()
        ↳ , KeyCoordinator.newCoordinationData(roomId));
44         capi.write(recC, e);
45     } catch (MzsCoreException e) {
46         throw new ActionException("Couldn't add room reference to
        ↳ reception \"" + roomName + "\".", e);
47     }
48 }
49
50 public List<Item> lookup() {
51     ...
52 }
53
54 ...
55 }

```

Listing 6.3: Sample micro-room file - Reception.java

The listing represents the implementation of the `Reception` used by the business logic file in listing 6.2. The methods listed from line 4 to 37 have to be overwritten and are described in table 6.3.

In the example, a container is created on initialization, looked up again when restoring the room, and deleted if the room is destroyed. If a peer enters a `Reception`, the micro-room instance of that peer automatically synchronizes with the room replicas of the other members. This is done by calling the `lookup` action on one of these peers (line 30) and adding the result

Method	Description
<code>initializeRoom()</code>	Defines which operations should be performed on the initial creation of the room and thus is only called once.
<code>restoreRoom(ModuleStorage storage)</code>	Specifies which operations should be performed on each startup of the framework, on that an instance of this room is restored from its persisted state. The <code>ModuleStorage</code> object allows to retrieve user-defined values that have been stored there in the <code>suspendRoom()</code> method.
<code>suspendRoom(ModuleStorage storage)</code>	Defines which operations should be performed on each shutdown of the framework, before all containers are persisted. The <code>ModuleStorage</code> object allows to save all values that are not stored within a container, so that they can be restored again on the next framework startup by <code>restoreRoom()</code> .
<code>destroyRoom()</code>	Specifies which operations should be performed on the final destruction of the room and thus is only called once. Please note that a room is never destroyed by the framework automatically.
<code>onEnter(Set<String> replicateTo)</code>	Can be used to perform special synchronization tasks on the room instance of this peer, if it enters the room. The <code>replicateTo</code> parameter contains a list of all other peers that are currently a member of this room and thus have the required information for synchronization.
<code>onLeave(Set<String> replicateTo)</code>	Can be used to perform special clean-up tasks on the room instance of this peer, if it leaves the room. The <code>replicateTo</code> parameter contains a list of all other peers, that are currently a member of this room. This method is especially important to ensure privacy (e.g. to enforce deletion of all messages if a user leaves the chat room).

Table 6.3: Abstract methods derived from `BaseMicroRoom`

to the local room instance (line 31). If the peer leaves the room, all items currently stored in the local replica of the `Reception` will be removed (line 36).

After the implementation of all six abstract methods derived from `BaseMicroRoom`, the application developer can add arbitrary further custom methods. These represent the actions supported by the micro-room. In the listing, only two examples are shown for simplicity. The method (i.e. action) `lookup` in line 50 returns a list of all items stored in the `Reception`.

The action defined in line 40 adds the passed item to the `Reception`'s internal storage. In contrast to the `lookup` action, this action needs to be replicated to all other replication targets (defined by the `Replication` plugin), because it modifies the internal storage. This can be signaled to the framework by adding a `@Replicate` annotation to the method (line 39). Another feature that can be seen in the example, is the error handling. Error messages that are intended to be seen by the user, can be passed through all framework layers directly to the UI by encapsulating them into an `ActionException` (cf. line 46).

In the same line also the undefined variable `roomName` is used. This variable belongs to one of the instance variables defined in `BaseMicroRoom`. All of these instance variables are described in table 6.4, including their content during runtime.

Instance variable	Description
ReplicationCapi capi	Customized <code>Capi</code> object with integrated Paxos Commit support that can be used to perform operations on containers.
Configuration config	Configuration object encapsulating all values of the business logic file corresponding to this room instance.
String roomName	Name of this room instance.
Request request	Current <code>Request</code> object fetched from the <code>Request Container</code> . Will be injected by the framework before an action is called and can be used to e.g. get the user ID of the requestor.
List<Plugin> plugins	A list of all plugins that are registered on this micro-room instance.
List<Connection> connections	A list of all connections that are registered on actions of this micro-room instance.
Map<String, Action> additionalActions	A Map in "actionName -> Action" format that contains additional actions available in this room. This list should only be used by plugins to add/remove actions dynamically (cf. section 6.2.2).
ModuleStorage storage	The instance of the <code>ModuleStorage</code> correlated with this micro-room instance. Is exactly the same object as in the <code>restoreRoom()</code> and <code>suspendRoom()</code> method.
ProcessEngine processEngine	A reference to the current <code>ProcessEngine</code> object, thus granting access to all important features of the framework, such as all managers and available modules.

Table 6.4: Instance variables derived from `BaseMicroRoom`

A special note is required for the `config` variable. It allows to access all settings made by the end user within the corresponding business logic file. So, based on example listing 6.2, one could access the name of the room by `config.getValue("room.name")` or the value of the custom attribute `getOfflineMessages` by `config.getValue("room.getOfflineMessages")`. Please note the “room.” prefix that is used to access attributes within the room XML node of the business logic file. Other XML node attributes can be accessed with other prefixes (cf. section 6.2.2).

As can be seen, any functionality can be implemented within micro-rooms, since they are normal Java classes. It doesn’t even need to be based on XVSM and containers, but could e.g. use a database, files, or web services.

6.2.2 Plugin File

Equally to micro-rooms, plugins are created by a Java class implementing the `Plugin` interface. Again, we provide an easier-to-use abstract `BasePlugin` class from which developers should derive their plugin class. An example for a plugin is shown in listing 6.4.

```

1 public class Creatable extends BasePlugin {
2     @Override
3     public void initialize() {
4         ...
5         Action action = new CreateAction(microRoom);
6         String actionName = config.getValue(Configuration.
7             ↳PREFIX_PLUGIN_PROPERTY + "actionName");
8         microRoom.getAdditionalActions().put(actionName, action);
9         String accessibleBy = config.getValue(Configuration.
10            ↳PREFIX_PLUGIN_PROPERTY + "accessibleBy");
11         microRoom.setPermissions(actionName, accessibleBy);
12         ...
13     }
14
15     @Override
16     public void restore(ModuleStorage storage) {
17         ...
18         Action action = new CreateAction(microRoom);
19         String actionName = config.getValue(Configuration.
20            ↳PREFIX_PLUGIN_PROPERTY + "actionName");
21         microRoom.getAdditionalActions().put(actionName, action);
22         ...
23     }
24
25     @Override
26     public void suspend(ModuleStorage storage) {
27     }
28
29     @Override
30     public void destroy() {
31         ...
32     }
33 }
```

```

29 |     String actionName = config.getValue(Configuration.
    |         ↪PREFIX_PLUGIN_PROPERTY + "actionName");
30 |     microRoom.getAdditionalActions().remove(actionName);
31 |     microRoom.clearPermissions(actionName);
32 | }
33 | }

```

Listing 6.4: Sample plugin file - Creatable.java

The listing shows the implementation of the `Creatable` plugin used in listing 6.2 in line 14. As every plugin, it consists only of the implementation of the four derived methods from `BasePlugin`. These have exactly the same semantics as the corresponding methods derived from the `BaseMicroRoom` described in table 6.3.

The first time a new instance of the sample plugin is created (cf. lines 2 to 11), it creates a new `CreateAction` and adds it to the list of additional actions of the micro-room that contains this plugin. Additionally, the permissions of the new action are set according to the business logic file. To obtain the access string and the action name, the `config` instance variable derived from the `BasePlugin` class is used, similarly to the `BaseMicroRoom`.

If the plugin is restored from persistence (cf. lines 13 to 20), only the `CreateAction` is re-assigned to the corresponding micro-room, as this object is not saved within any container and thus lost after shutdown. The access string on the other hand is persisted within the `Permission Container` and therefore can be restored at startup. In case the plugin is destroyed, the action is removed (cf. lines 26 to 32) from the micro-room’s list of additional actions and the permissions are cleared.

Again, there are several important instance variables available within a plugin, allowing access to the environment. They are described in table 6.5.

Instance variable	Description
ReplicationCapi capi	Customized <code>Capi</code> object with integrated Paxos Commit support that can be used to perform operations on containers.
Configuration config	Configuration object encapsulating all values of the business logic file corresponding to this plugin instance.
MicroRoom microRoom	The instance of the <code>MicroRoom</code> to which this plugin instance is assigned to.
ProcessEngine processEngine	A reference to the current <code>ProcessEngine</code> object, thus granting access to all important features of the framework, such as all managers and available modules.

Table 6.5: Instance variables derived from `BasePlugin`

This time, the `config` variable needs a “plugin.” prefix to access attributes defined directly within the `plugin` node. Key-value pairs defined via `property` nodes can be accessed

via the “plugin.property.” prefix (i.e. the `Configuration.PREFIX_PLUGIN_PROPERTY` constant).

The implementation of the `CreateAction` mentioned above can be seen in listing 6.5.

```
1 public class CreateAction extends BaseAction{
2     public CreateAction(MicroRoom microRoom) {
3         super();
4         this.microRoom = microRoom;
5         this.replicate = true;
6     }
7
8     @Override
9     public Object execute(Object... params) throws
10         ↳InvalidParametersException {
11         //create a new room instance and return an Item with meta data
12         ...
13     }
```

Listing 6.5: Sample action implementation file - `CreateAction.java`

Fixed actions of a micro-room are normally implemented directly via methods defined within the micro-room class. The mechanism shown in the listing above should only be used by plugins to add actions dynamically to micro-room instances.

Thus, it is required to write a Java class implementing the `Action` interface or - preferably - extending the abstract `BaseAction` class. From this class, two instance variables are derived, i.e. `MicroRoom microRoom` and `boolean replicate`. The `microRoom` variable contains a reference to the micro-room instance this action is contained in and `replicate` fulfills the same purpose as the `@Replicate` annotation mentioned in the last section, i.e. to define whether the action should be replicated to the other replication targets or not.

In the example above, this is done in line 5. To define the logic of the action itself, it is only required to overwrite the abstract `Object execute(Object...params)` method derived from `BaseAction` (cf. line 9). Due to its generic method signature, it is possible to pass any arguments and return any values.

With this mechanism, it is possible to create plugins that dynamically extend already existing micro-rooms with new actions. Thus, you can add functionality to arbitrary micro-rooms without even modifying them.

6.3 User Interface Designer Manual

Usually, the UI is highly customized to the desired application and thus can't be generated automatically. Technically, this would be possible, but due to the wide range of possible use cases, the generated UI is likely to be unsatisfying. Therefore, we decided not to generate the UI automatically, but to provide an easy API for the UI designer to interact with the XVSM Micro-Room Framework.

6.3.1 REST API

As REST is one of the most popular and thus known technologies for web-based communication, we use it as a basis for our API. It can be used within any programming language that at least supports some sort of network communication. In the following, it is described how the API can be used.

First of all, the UI (and thus the user) needs to authenticate at the central authentication service to obtain a valid SSO token. This can be done with a simple HTTP GET request to the URL

```
"http://my.openam-server.com:8080/openam/authenticate?
    username=" + uId + "&password=" + pwd
```

where `uId` represents the user ID and `pwd` the password, both requested from the user. The returned value will be a valid SSO token that has to be attached to each following request.

In order to send requests and retrieve responses, the UI first needs to determine the exact container IDs of the Request Container and the Response Container. This can be done by sending both HTTP GET requests, shown in listings 6.6 and 6.7, to the running framework instance. By default, the framework can be reached via `http://localhost:9877`. A different port can be configured in the `mozartspaces.xml` configuration file (`rest` node).

```
1 Request URL:      "/containers/container/lookup?name=requestC"
2 Request header: Content-Type="application/json"
```

Listing 6.6: REST API - Request container lookup

```
1 Request URL:      "/containers/container/lookup?name=responseC"
2 Request header: Content-Type="application/json"
```

Listing 6.7: REST API - Response container lookup

The answer to both lookup requests will be an object in *JavaScript Object Notation* (JSON) format. The corresponding container URL is contained within the following nested property of the answer object (we assume that this object is named `resp`):

```
resp["org.mozartspaces.rest.resources.RESTContainerResponse"].
    response.link
```

Due to the current XVSM Security implementation, it is necessary to pass the user ID and all of his/her roles along with the request. These attributes are verified within XVSM Security's authentication mechanism and thus mandatory to interact with the framework. To retrieve the roles of the logged-in user the following HTTP GET request has to be sent to the authentication service:

```
"http://my.openam-server.com:8080/openam/identity/attributes?
    subjectid=" + SSOToken + "&attributenames=memberof"
```

The response of this call will contain a line starting with `userdetails.attribute.value` that includes all roles of the current user. With a valid SSO token, the URLs of both Request and Response Container, and the roles of the user, the UI is ready to send requests to the framework. The required HTTP POST request is shown in listing 6.8.

```

1 Request URL:      request container URL + "/write"
2 Request header: Content-Type="application/json",
3                  X-requestId=unique request ID,
4                  isolation="READ_COMMITTED"
5 Request body:
6 {"Write":
7  {"context":
8   {"properties":[{"@class":"java.util.concurrent.ConcurrentHashMap
9     ↪","@serialization":"custom","unserializable-parents":"","
10     ↪java.util.concurrent.ConcurrentHashMap":{"default":{"
11     ↪segmentMask":0,"segmentShift":0,"segments":[{"java.util.
12     ↪concurrent.ConcurrentHashMap$Segment":[{"sync":{"@class":"
13     ↪java.util.concurrent.locks.ReentrantLock$NonfairSync","
14     ↪@serialization":"custom","java.util.concurrent.locks.
15     ↪AbstractQueuedSynchronizer":{"default":{"state":0}},"java.
16     ↪util.concurrent.locks.ReentrantLock$Sync":{"default":""}},"
17     ↪loadFactor":0.75}]}}]},
18   "string":[
19     "REQ.tokenId",SSO token,
20     "REQ.remoteRequest",true,
21     "REQ.authAttributes"], "set":[{"
22       "org.mozartspaces.core.authorization.NamedValue":[{"
23         "name":"user",
24         "value":user ID
25       },
26       {
27         "name":"role",
28         "value":role name 1
29       },
30       {
31         "name":"role",
32         "value":role name n
33       }
34     ]
35     },
36     "null":["",""]
37   }]}],
38   "isolation": "READ_COMMITTED",
39   "container": request container,
40   "timeout": request timeout in ms,
41   "entries":
42   [{"Entry":
43     {"value":
44       {"@class": "org.xvsm.microroom.framework.objects.Request",
45        "requestId": unique request ID,

```

```

36         "scenarioName": scenario name,
37         "roomName": room name,
38         "actionName": action name,
39         "params": parameters
40     },
41     "coordData": [{"FifoData": {"name": "FifoCoordinator"}}]
42 }
43 }
44 }
45 }

```

Listing 6.8: REST API - Send request

Most of the request body is static and its complexity is owed to the current XVSM REST implementation. However, these parts can just be copied as only the bold entries need to be replaced with the desired values.

The unique request ID needs to be unique only for this peer and thus can be created, e.g. by combining the current timestamp with a local counter. The parameters of the action need to be specified in a JSON format readable by the framework. In the XVSM REST implementation XStream is used for JSON to Java conversion, thus the documentation of XStream [25] will provide sufficient help. However, for defining complex objects it is likely that you will have to specify several lines of code (cf. the multiply nested Map in line 8).

Listing 6.8 represents a write operation of a Request object into the Request Container. To read the corresponding Response object and thus obtain the return value of the called action, listing 6.9 shows the according HTTP POST request.

```

1 Request URL:      response container URL + "/read"
2 Request header: Content-Type="application/json",
3                 X-requestId=unique request ID,
4                 isolation="READ_COMMITTED"
5 Request body:
6 {"Read":
7   {"context":
8     {"properties":[{"@class":"java.util.concurrent.ConcurrentHashMap
9       ↳","@serialization":"custom","unserializable-parents":"","
10        ↳java.util.concurrent.ConcurrentHashMap":{"default":{"
11        ↳segmentMask":0,"segmentShift":0,"segments":[{"java.util.
12        ↳concurrent.ConcurrentHashMap$Segment":[{"sync":{"@class":"
13        ↳java.util.concurrent.locks.ReentrantLock$NonfairSync","
14        ↳@serialization":"custom","java.util.concurrent.locks.
15        ↳AbstractQueuedSynchronizer":{"default":{"state":0}}, "java.
16        ↳util.concurrent.locks.ReentrantLock$Sync":{"default":""}}, "
17        ↳loadFactor":0.75}}]}]}],
9     "string":[
10       "REQ.tokenId",SSO token,
11       "REQ.remoteRequest",true,
12       "REQ.authAttributes"], "set":[{"
13         "org.mozartspaces.core.authorization.NamedValue":[{"
14         "name":"user",

```

```

15         "value":user ID
16     },
17     {
18         "name":"role",
19         "value":role 1
20     },
21     {
22         "name":"role",
23         "value":role n
24     }
25 ],
26 "null":["", ""]
27 }]],
28 "isolation": "READ_COMMITTED",
29 "container": response container,
30 "timeout": request timeout,
31 "selectors": [{
32     "LabelSelector": [{
33         "label":request ID,
34         "name":"requestId",
35         "count":1
36     },
37     {
38         "label":user ID,
39         "name":"userId",
40         "count":-2
41     }
42 ]
43 }
44 }

```

Listing 6.9: REST API - Read response

This request is very similar to the other one, but differs in important parts. This time we don't call an action, but specify which `Response` object we want to retrieve. This is done in lines 33 and 38 by specifying the ID that has been used for the request (cf. listing 6.8, line 35) and our user ID. Please note that by assigning the user ID of another user, you won't be able to retrieve other `Response` objects, but receive an error due to security constraints (cf. section 5.3.1).

The answer will contain a `Response` object in JSON format that allows to access the return value by accessing the following field (we assume that the `Response` object itself has the name `resp`):

```

resp.Response.result["org.xvsm.microroom.framework.objects.
                                Response"].retVal

```

So in the UI, first a HTTP POST request needs to be performed for sending the request. In case a return value is expected, an additional HTTP POST request is necessary for retrieving the

return value. Ideally this second request is performed in an asynchronous way, in order to avoid polling.

Eventually, when the user logs out of the UI, the following HTTP GET request to the central authentication service will invalidate the SSO token:

```
"http://my.openam-server.com:8080/openam/identity/logout?  
subjectid=" + this.SSOTokenId
```

6.3.2 JavaScript Wrapper Library

As we decided to develop the UI of the P2P OSN in HTML5, we created a JavaScript wrapper library that encapsulates all of REST API calls described above. Thus, UI development for web-based UIs is eased even more.

When using the JavaScript wrapper library, it first needs to be included via a `<script src="js/xmrf.js" />` tag. The library provides a global `xmrf` object that can be used for making settings and calling actions. Before the library can interact with the corresponding framework instance it needs to be initialized by specifying the application-dependent URLs and the user credentials provided by the user. Additionally, the `requestTimeout` for calling an action and the `pubSubTimeout` for reading asynchronous event notifications (cf. section 6.3.2) can be defined. The initialization can be seen in JS listing 6.10.

```
1 xmrf.spaceUri = "http://localhost:9877";  
2 xmrf.openAmUri = "http://my.openam-server.com:8080/openam";  
3 xmrf.requestTimeout = 10000;  
4 xmrf.pubSubTimeout = 3600000;  
5 xmrf.login(userId, password);
```

Listing 6.10: JavaScript wrapper library - Initialization

After this phase, arbitrary action calls can be sent to the framework. As there are different requirements for interacting with the framework, four different request types have been defined. These are very similar to the *Message Exchange Patterns* (MEP) supported by the *Windows Communication Foundation* (WCF) as presented in [26].

One-way

One-way requests are intended for asynchronous action calls that won't return any value. Even though the request is asynchronous, it is ensured that the `Request` object actually is stored in the `Request Container` and will thus be handled by the framework. If any error occurs while writing to the container (e.g. the UI is not allowed to call the specified action), the REST call will be answered with an error message that is then shown in an `alert()` message box. Hence, the UI can trigger actions immediately without blocking and at the same time be sure that the action will be performed by the framework.

The `tell` action of a created "My Chat" chat room, as defined in listing 6.2, could be called as follows:


```

1 var item = new Item(["string":["text", "hi there!"]]);
2 xmrf.action("Chatting", "Chat Room->My Chat", "tell", msgType.ONEWAY,
    ↪ item);

```

Listing 6.11: JavaScript wrapper library - Oneway call

Asynchronous Request-Response

In contrast to the previous request type, the `Asynchronous Request-Response` request can be used for action calls that will return a value to the UI. Thereby, the request handling works exactly as with `One-way` requests, but additionally the corresponding `Response` object will be fetched from the `Response Container`. This is done via an asynchronous HTTP POST request that thus doesn't block the UI. If the response arrives, the specified callback function will be called for user-defined handling of the return value.

Calling the `lookup` action of the `Reception` defined in listing 6.2 and showing all stored items could be realized by the code fragment shown in listing 6.12. If the action required parameters, they would have to be inserted after the `msgType` and before the callback function.

```

1 xmrf.action("Chatting", "Chat Room Overview", "lookup", msgType.
    ↪ASYNC_REQRESP,
2     function(retVal){
3         if(!(retVal instanceof Error)){
4             $.each(retVal, function(i, room){
5                 alert(JSON.stringify(room));
6             });
7         }
8     }
9 );

```

Listing 6.12: JavaScript wrapper library - Asynchronous request-response call

Synchronous Request-Response

This request type is equal to the `Asynchronous Request-Response` request, except that this time the action call blocks the UI until the response can be fetched from the `Response Container`. Due to its synchronous behavior, it is easier to understand and can be used if the user really has to wait for a certain action response. However, in the normal case one should prefer the `Asynchronous Request-Response` request type, as a blocking UI might be frustrating for most users.

The following listing shows how the `lookup` call from the example above could be performed in a synchronous way. Again, if parameters are needed for the action, they can be added after the `msgType` argument. Please note the small but mighty difference in line 1 and 2. This time, no callback function is passed as an argument to the `action` function, but line 1 blocks until the return value is available.

```

1 var retVal = xmrf.action("Chatting", "Chat Room Overview", "lookup",
    ↪msgType.SYNC_REQRESP);
2 if(!(retVal instanceof Error)){
3     $.each(retVal, function(i, room){
4         alert(JSON.stringify(room));
5     });
6 }

```

Listing 6.13: JavaScript wrapper library - Synchronous request-response call

Publish-Subscribe

Finally, there is the Publish-Subscribe request type, which allows the UI to register for event-based actions. In listing 6.2 the `listen` action of the chat room is an example for that. After this action is called, the micro-room will create `Response` objects with the request ID of the initial `listen` call, for all upcoming chat messages. The JavaScript wrapper library will then asynchronously and continuously read all responses with this ID and call the passed callback action whenever a response is fetched. Thus the UI is notified immediately, if a new chat message has been sent by another user. The event registration process can be seen in listing 6.14.

```

1 var pubSub = xmrf.action("Chatting", "Chat Room->My Chat", "listen",
    ↪msgType.PUBSUB,
2     function(retVal){
3         if(!(retVal instanceof Error)){
4             alert(JSON.stringify(retVal));
5         }
6     }
7 );

```

Listing 6.14: JavaScript wrapper library - Publish-subscribe registration

Please also note the `pubSub` object that is returned from the call of the `action` function. This object can be used to unregister from the event notifications and stop the continuous reading of event notifications, which is shown in listing 6.15. Therefore we have to specify the deregistration action of the room (i.e. `stopListening`) and whether the call itself should be asynchronous or not.

```

1 pubSub.unregister("Chatting", "Chat Room->My Chat", "stopListening",
    ↪false);

```

Listing 6.15: JavaScript wrapper library - Publish-subscribe deregistration

Finally, if the user logs out and/or closes the UI, the JavaScript wrapper library can be completely cleaned up by the following line:

```

1 xmrf.logout();

```

Listing 6.16: JavaScript wrapper library - Clean up

File Download

As has already been mentioned in section 5.3.2, the JavaScript wrapper library provides special functions to ease file handling. Listing 6.17 shows how a file can be downloaded from the framework instance.

```
1 var item = new Item(["string":["fileName", "homework 1.pdf"]]);
2 xmrf.action("Homework", "Homework Files", "read", msgType.
    ↪ASYNC_REQRESP, item,
3     function(retVal){
4         if(!(retVal instanceof Error)){
5             var uri = xmrf.retValToFileUri(retVal, "application/pdf");
6             $("#downloadLink").attr("href", uri);
7         }
8     });
```

Listing 6.17: JavaScript wrapper library - File download

The item defined in the first line contains the name of the file that should be downloaded. This object is passed as parameter to the `read` action of the `FileRoom` in line 2. In the callback function of the `Asynchronous Request-Response` request the return value `retVal` is passed to the `retValToFileUri()` function.

This function is provided by the JavaScript wrapper library and contains all the relevant steps necessary to properly create a `Blob` object from the return value of a `FileRoom`'s `read` action. It includes Base64 decoding the passed byte array and considering the different browser implementations of the `Blob` API. The return value of the function is a simple URL that can be used just as a normal URL. In line 6, it is injected into an existing `<a>` tag with the ID `downloadLink`. Thus, the user can show or save the PDF if he/she clicks onto this link. If we were dealing with an image, the URL could also be injected directly into an `` tag to show the downloaded picture.

File Upload

The upload of a file to a framework instance can be seen in listing 6.18.

```
1 $("#fileChooser").change(function(e) {
2     e.preventDefault();
3     var file = e.target.files[0];
4     xmrf.fileToParam(file.name, file, function(param){
5         xmrf.action("Homework", "Homework Files", "write", msgType.ONEWAY
        ↪, param);
6     });
7 });
```

Listing 6.18: JavaScript wrapper library - File upload

Thereby, `fileChooser` is just a standard `<input type="file" />` element, defined in the HTML file. A click onto the generated button will show the browser-specific file chooser

dialog. After a file has been chosen, the `change` event will be triggered and thus call the code shown above. The file chosen can be accessed as shown in line 3.

Again, the sophisticated part (i.e. the conversion from the file to a Base64 encoded byte array) is done within the `fileToParam()` function. It will store the byte array in the `param` object and call the user-defined callback function if the conversion is complete. In the callback function the `write` action of the `FileRoom` is called with the generated byte array `param` (line 5).

6.4 Deployment Possibilities

In this section we will discuss several ways of how applications developed with the XVSM Micro-Room Framework can be deployed. Even though the framework is intended for P2P applications, it can also be used to realize traditional client-server applications, thin clients or a system of replicated servers. Figure 6.3 shows four of the most interesting deployment scenarios.

6.4.1 Fully autonomous peers

This is the deployment scenario that the XVSM Micro-Room Framework was intended for in the first place. The peer holds everything locally, including the back-end (i.e. the running framework instance), the UI and all configuration files required. Thereby, the UI could be written in an arbitrary programming language such as Java, C# or C++. The probably most simple approach is to implement it with HTML5 and use the JavaScript wrapper library as shown in the figure 6.3. The implemented HTML and JS files can be stored locally and accessed by each browser via the `file://` protocol, without the need of setting up a web server.

The approach described allows the peer to work fully autonomously, but has the disadvantage that, e.g. the correction of bugs in provided modules becomes difficult, as every peer needs to download the new files on his/her own.

6.4.2 Centralized configuration and/or user interface

In contrast to deployment possibility a) this approach uses a central web server (e.g. Apache) to store the configuration and UI files. By using the `RemoteBusinessLogicLoader` and the `RemoteModuleLoader` the configuration files are fetched automatically by the framework instance of each peer. Due to the use of CORS (cf. section 5.3.2) the different locations of both the framework and the HTML files are no problem concerning SOP.

This scenario makes it easier for both the application developer and the UI designer to make adjustments, as they don't need to distribute the changed files to each peer. Changes on the central web server are sufficient, thereby reducing the maintenance effort significantly.

6.4.3 Client-server

The client-server deployment scenario can be seen as a special case where only one peer (i.e. one framework instance) exists. This instance is located on a central web server, among the required configuration and HTML files. As can be seen in the figure, the communication between peer

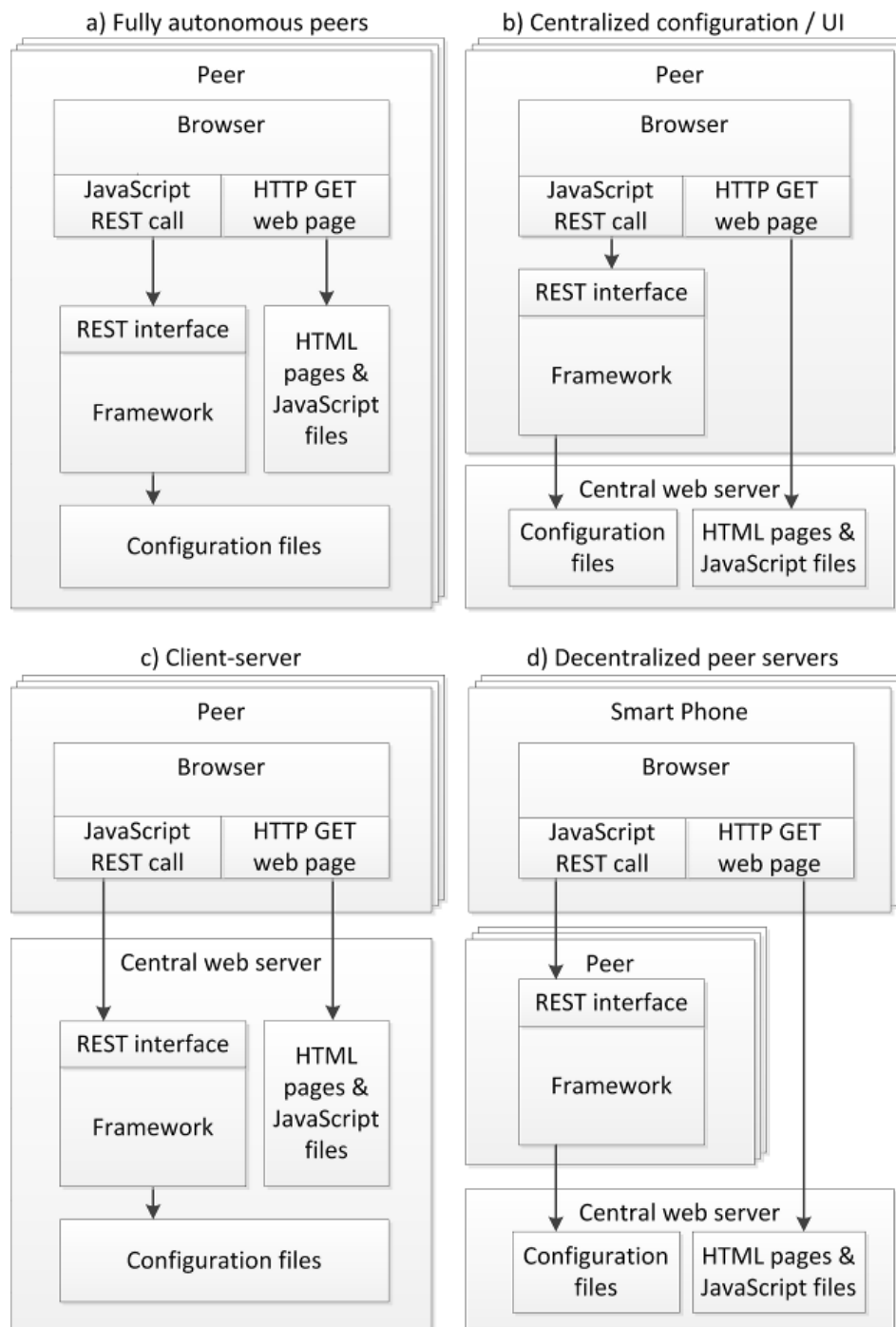


Figure 6.3: Deployment possibilities

and web server is similar to common web-based client-server technologies, such as Java EE or Microsoft ASP.NET. Thus, also in traditional client-server scenarios users can benefit from the advantages of the XVSM Micro-Room Framework.

By not using HTML5 but another technology for designing the UI, it is possible to generate thin clients. One could e.g. implement the UI within a Java client. This software runs locally on the peer and only communicates with the “server” via the REST API. The client only is responsible for allowing users to trigger actions on the server and to visualize the results, thus acting as a thin client. The whole business logic itself is carried out on the central framework instance, i.e. the “server”.

6.4.4 Decentralized peer servers

The last deployment possibility described can be compared to the architecture of Diaspora as described in section 2.2.1. It consists of multiple machines that each run a framework instance. These machines could be seen as “peer servers” and are equal to the Pods in Diaspora. On the other hand, they could also be used to create a highly-available cluster of replicated servers. Additionally, they could also provide their custom UI if they also run e.g. a web server. In the figure, this is not the case as the UI is hosted on a central web server. As in scenario c), a locally running client software could be used as front-end to avoid a central web server.

The figure also shows another important aspect: By using HTML, even smart phones can use applications developed by the XVSM Micro-Room Framework without additional effort. This is true since all required features of the JavaScript wrapper library are supported by common smart phone browsers (cf. section 5.3.2). In the visualized example, the smart phone therefore connects to both the central web server delivering the UI and the framework instance located on another machine, e.g. another server or even the user’s home PC.

Evaluation

After describing the design, implementation and deployment possibilities of the XVSM Micro-Room Framework, we will now evaluate its usefulness. Therefore, we will first describe the proof-of-concept P2P OSN that we developed with the framework and thus outline its capabilities. Then, we will compare the generated P2P OSN with the existing solutions, based on the criteria defined in section 2.2.7. After evaluating the proof-of-concept implementation, we will also compare the XVSM Micro-Room Framework itself with the few existing alternatives described in section 2.3. Additionally, the overall framework architecture, including the technologies used, will be assessed critically. Finally, we will conclude this chapter by performing some performance benchmarks, showing that the framework is usable in real life situations.

7.1 Use Case Overview

In the following we will outline our proof-of-concept P2P OSN , implemented with the XVSM Micro-Room Framework. We decided to settle the use case in an e-learning environment, that is capable of both OSN and e-learning features, as this scenario allows more collaboration features and complex workflows. The features implemented have been chosen to show different sets of the framework's capabilities. Most of them are recreated based on popular OSNs and e-learning platforms.

For the OSN part we heavily rely on Facebook in order to create the same user experience and thus ensure a familiar UI. Concerning e-learning features, popular platforms such as moodle¹ have been used as inspiration. A comparable P2P OSN with e-learning support doesn't yet exist. The Academic Social Network developed in [JP10] is oriented in the same direction, but lacks of P2P support and misses collaboration features required for e-learning.

¹<http://moodle.org/>

7.1.1 Borrowing Books

The first scenario (i.e. business logic file) allows the librarian to manage his/her library and students to borrow books (cf. figure 7.1).

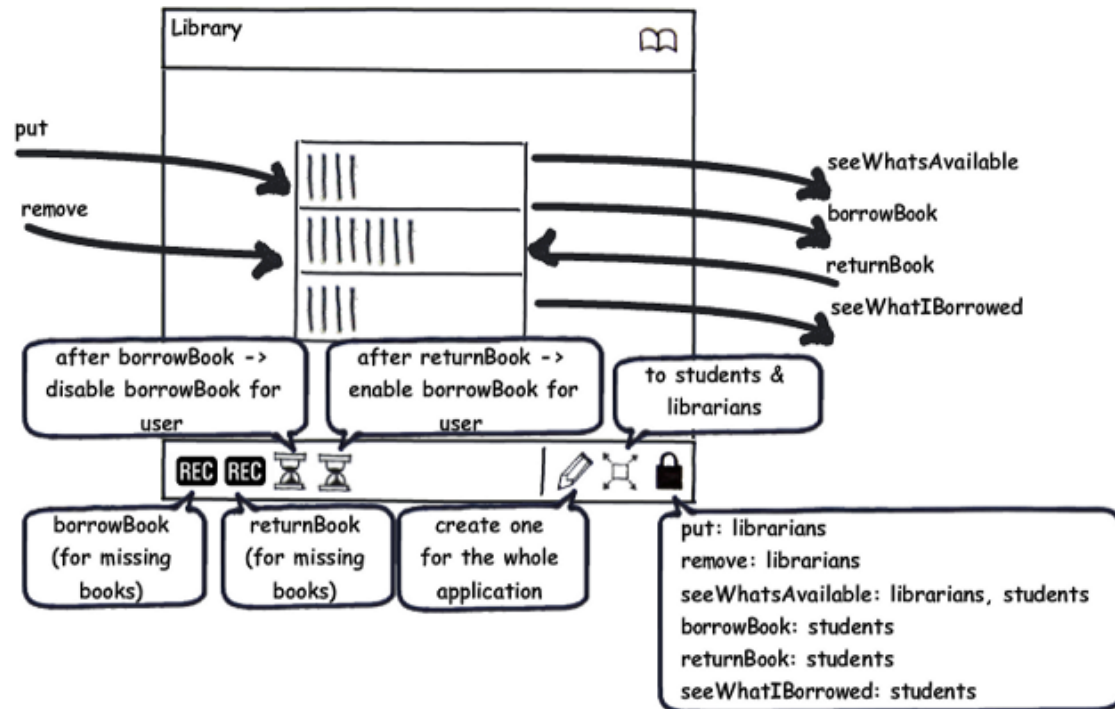


Figure 7.1: P2P OSN scenario - Borrowing books

As can be seen, the whole business logic required for this scenario can be created by just one micro-room of type `Library`. Actions to manage books (`put`, `remove`) are restricted to the managing librarian only, while those required for borrowing books are accessible for all students.

Some specialities of this scenario are the four plugins used. The first two are `History` plugins and the other two are `Event` plugins. The `History` plugin can be attached to any action of a room and logs every call of this action into a newly created container. The logged entries can be accessed at any time by a configurable and automatically added action. In the scenario, this plugin is used to log the `borrowBook` and `returnBook` actions. Thus, the librarian always knows who currently has borrowed which book.

The `Event` plugin on the other hand is a rather flexible plugin. It consists of two parts: the event source and the event action. Basically, the event source defines when the event action should be triggered. Currently, defined event sources are before or after an action is called by the user, or when a defined point in time is reached. Event actions can be the call of an action or the modification of the permissions of an action.

In the scenario, this plugin is used in a multi-staged way. If a user calls `borrowBook`, the first plugin will revoke his/her permission to call this action again. If he/she calls `returnBook`,

the permission will be restored. Thus, a simple restriction to borrow only one book at the same time is established without any changes of the room itself.

Figure 7.2 shows the UI of this scenario from the librarian’s point of view, including the log entries created by the `History` plugins.

Welcome to the XMRF Demo Use Case!

[Logout](#)

Books available:

A Song of Ice and Fire, written by George R. R. Martins

Genre: Fantasy

Borrow history:

Title:	Borrowed by:	Borrowed on:	Returned on:
A Song of Ice and Fire	joe	2013-06-24 14:23:23	2013-06-24 14:23:56
The Lord of the Rings	max	2013-06-24 14:24:23	

Add new Book

Title:

Author:

Genre:

Figure 7.2: P2P OSN user interface - Borrowing books

7.1.2 Chatting

The second scenario deals with chatting between users and is shown in figure 7.3. It is based on the example presented in section 6.1.2 and allows users to create their own chat rooms (of type `ConferenceRoom`) by using the `Creatable` plugin. Thereby, all configured settings including access restrictions of this room are copied to the new instance. The user can specify the members of the new chat room on creation and thus define who is able to chat in the room.

Since a room is never destroyed by the framework itself, this ability is added by the `Closable` plugin, as seen in the figure. It allows to finally destroy an instance of the room and therefore is only allowed to be called by the room owner him/herself.

Besides the two plugins, the scenario also shows how simple workflows can be created. If a room is created by the `create` action, an item containing meta data of the new instance is added to the chat room overview of type `Reception`. Otherwise the new room couldn’t be found again by other users. If this entry is removed again, the corresponding chat room instance will also be removed. More details about this workflow have already been discussed in section 6.1.2.

Finally, the model chat room also contains two special actions, i.e. `listen` and `stopListening`. `Listen` can be used to subscribe to all messages that other users `tell` in the room. If a user is done with listening, he/she can end the subscription by calling `stopListening`. These two methods refer to the event registration and deregistration actions described in section 6.3.2.

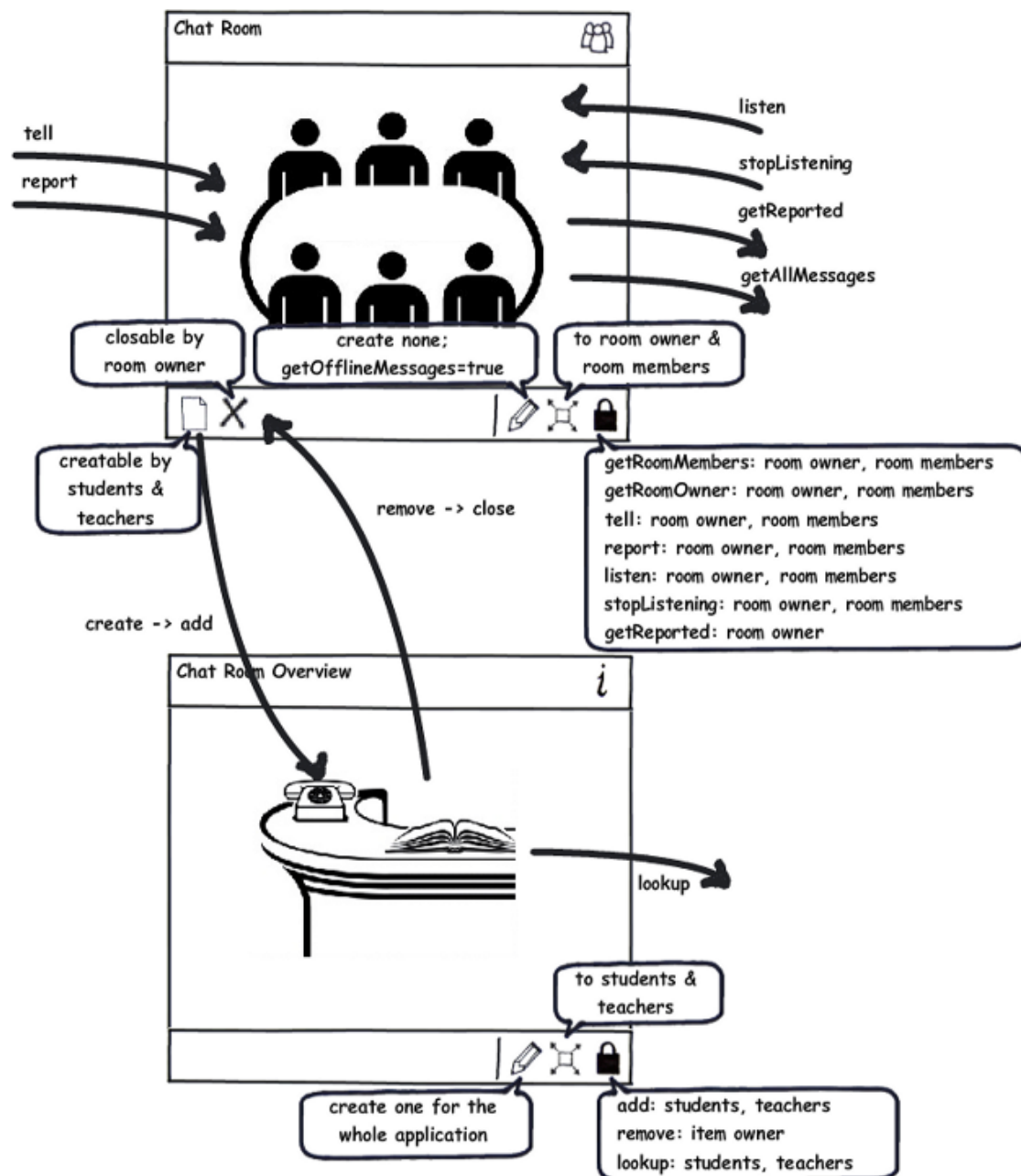


Figure 7.3: P2P OSN scenario - Chatting

The `getOfflineMessages` attribute exists only in this room type and defines whether messages of other users should still be recorded after `stopListening` or not. So one can easily change the behavior of the whole room by setting a simple flag.

In figure 7.4, one can see the UI for the chatting scenario. A list of all currently available chat rooms is shown on the right. All of them have been created previously by the corresponding

user, which is shown between the brackets.

The `ConferenceRoom` used as model chat room also provides the ability to report inappropriate messages. In the UI this can be done by right-clicking on the respective chat message. The appearing context menu is shown in the figure and displays an entry to report the message. The room owner can then click on the “Show reported messages” button to highlight reported messages as shown in the figure.

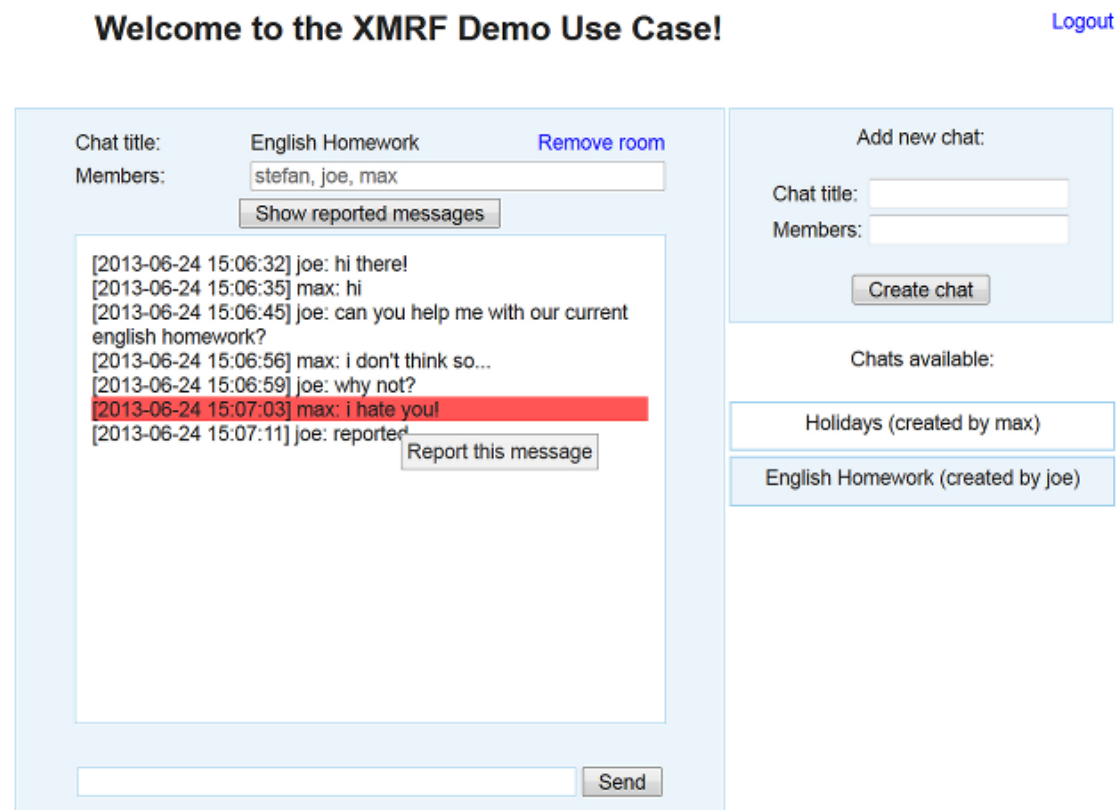


Figure 7.4: P2P OSN user interface - Chatting

7.1.3 Discussions

For enabling teachers to create announcements, another communication channel needs to be established, as those messages would get lost within a chat room. Therefore, discussions have been added that are similar to common message boards (cf. figure 7.5).

Apparently, the scenario is almost equal to the chatting scenario and uses only two room types, i.e. `ConferenceRoom` and `Reception`. This demonstrates how the same room types can be used for different scenarios. In this case, only several actions such as `listen` and `stopListening` are not required and thus not accessible by anyone. Please note that not specifying permissions for an action means that the action will be denied for everyone. The

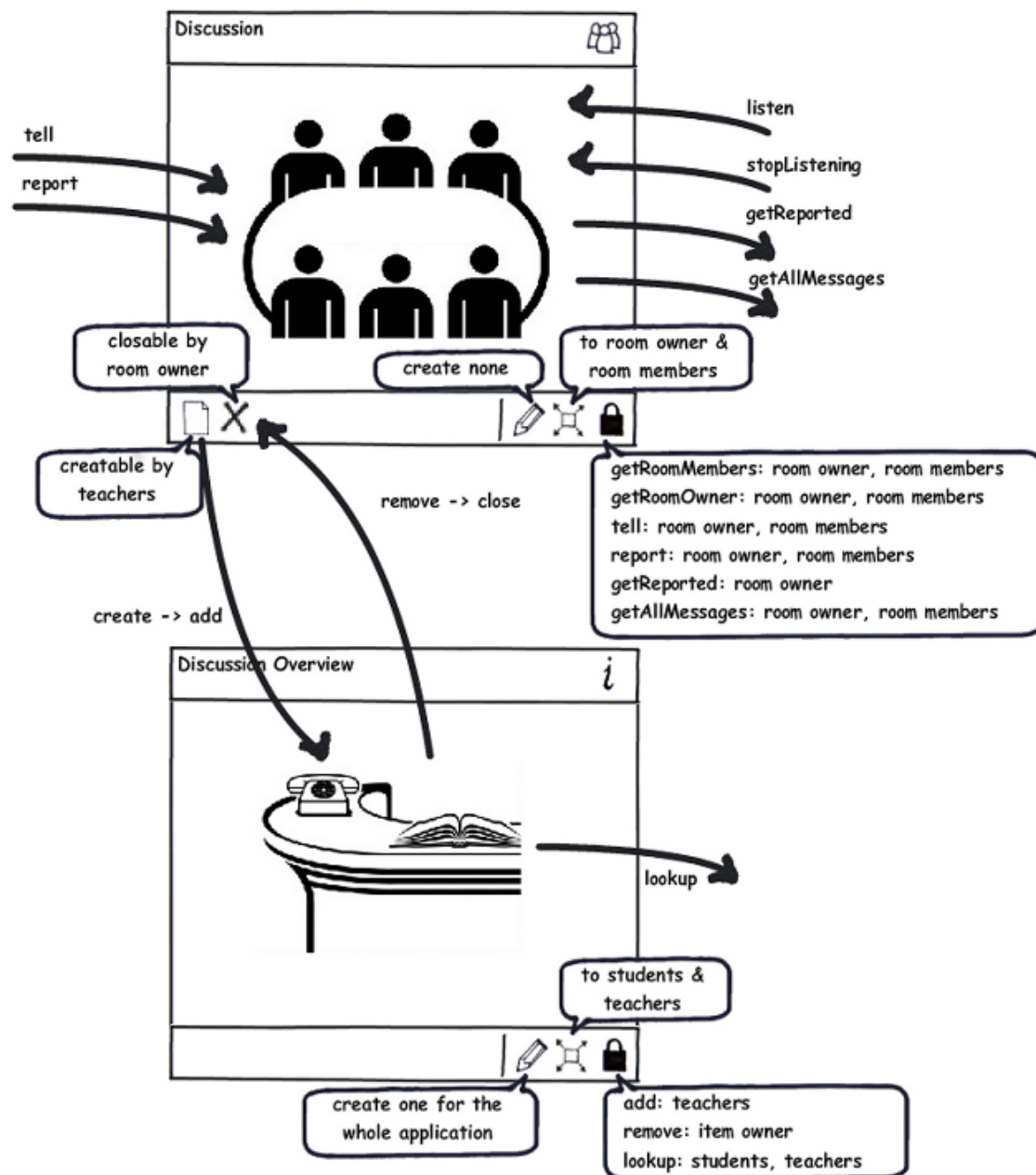


Figure 7.5: P2P OSN scenario - Discussions

creation of new rooms is limited to teachers this time, while communication within the room is still enabled for all room members as well as the room-owning teacher.

Even though the business logic file is almost identical, the UI for discussions looks completely different and is oriented towards message boards (cf. figure 7.6).

Discussion: Replacement Lesson [Remove discussion](#)

Members: stefan, joe

[Show reported messages](#)

englishprof
2013-06-24
15:48:45

Dear students of the 4b, tomorrow there will be a replacement lesson by Mrs. Rennington.

joe
2013-06-24
15:50:34

yeah!!

[Post message](#)

Add new discussion:

Title:

Members:

[Create discussion](#)

Discussions available:

Replacement Lesson
(created by englishprof)

Current Homework
(created by germanprof)

Figure 7.6: P2P OSN user interface - Discussions

7.1.4 Voting

An additional collaboration tool that is required in both OSNs and e-learning systems is a voting mechanism. This is realized in the scenario shown in figure 7.7.

The workflow is equal to the chatting and the discussion scenario and required to rediscover polls generated by users during runtime. The `PollSite` allows each room member to vote once. Therefore, the room owner has to define the question, all answers, and the maximum number of selectable answers, when he/she creates the room. This works by extending the configuration of the room with the `create` action, injected by the `Creatable` plugin. Besides the room name and the room members, the `create` action takes a `Configuration` object as a third parameter. All values defined there will be injected into the cloned instance of the new micro-room.

Considering the `Event` plugin, this works similarly. The plugin disallows access to the `vote` action at a specified time and thus can be used to specify how long the vote should take place. Just as with the attributes mentioned above, a fixed value in the business logic file wouldn't make sense here. Therefore, the value is represented in the business logic file as a variable (i.e. `$B`), that is resolved during the plugin's `initialize()` method. Please note that the injection of a value for `$B` into the room's (and thus also the plugin's) configuration during the `create` action call takes place before this resolving process.

With this mechanism, any attributes that use variables within the business logic file can be adjusted on creation of a new room instance. Hence, the flexibility of both micro-rooms and plugins is further increased.

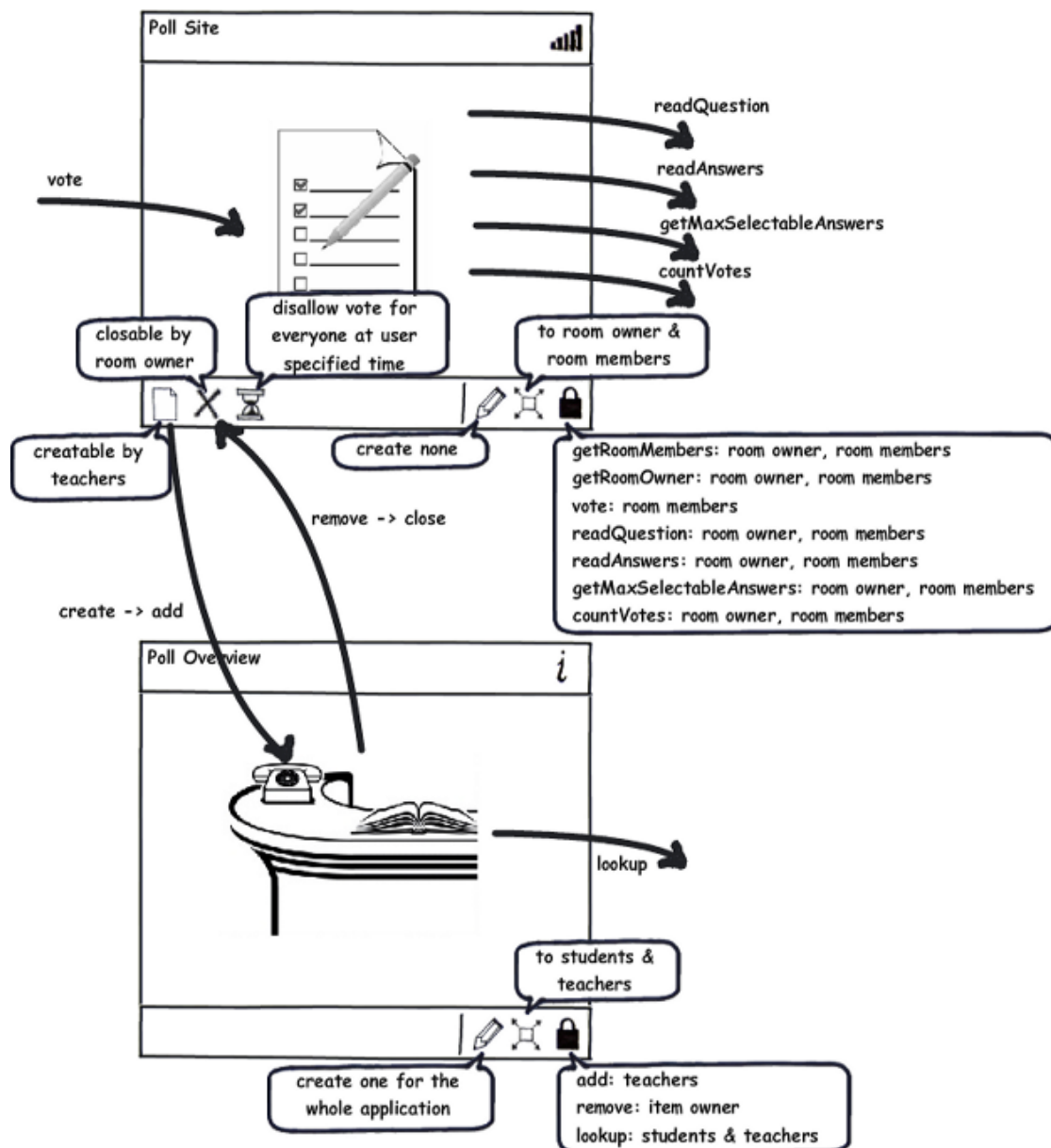


Figure 7.7: P2P OSN scenario - Voting

Figure 7.8 shows how such a poll is visualized. On the right you can see the various options that can be set when creating a new poll site. Another variable in the business logic file (i.e. \$A) is used for the access restrictions of the `countVotes` action that retrieves all votes given. It depends on the creator's preferences whether the poll results should be available only for him/herself (i.e. `$roomOwner`) or e.g. for all attendees of the poll (i.e. `$roomMembers`).

Should we make an excursion in October? [Remove poll](#)

Members:

Choose at most 1 answers:

Yes ☒

No ☐

Add new poll:

Question:

Members:

Close poll at:

Allow viewing results by:

Number of selectable answers:

Answer: [\[X\]](#)

Answer: [\[X\]](#)

Answer: [\[X\]](#)

Polls available:

Should we make an excursion in October?

(created by englishprof)

Figure 7.8: P2P OSN user interface - Voting

7.1.5 Profile

The basic feature of any OSN is the profile page of a user, including his/her wall, where all of his/her friends can write messages onto (cf. figure 7.9). In this scenario we use a very simple room of type `StorageRoom` that allows to store and retrieve arbitrary data. Of course, one could also use a room specialized for the needs of a profile. However, this hasn't been done in this case, due to the better re-usability of the `StorageRoom`. An important aspect is the `FileRoom` (cf. section 5.3.2) that can be seen at the bottom of the figure. It will be used to store the avatar image shown in the user's profile.

The `History` plugin defined on the `StorageRoom` is used to record all `seeWhatsInStock` action calls. This action will be called every time the profile is shown and the recording thus enables the profile owner to see all profile visitors.

The second plugin is of type `ChangableAccess`. It allows to specify which users can modify the access of a defined action during runtime. In this case the room owner (i.e. the profile owner) is allowed to change the access of the `store` action. This means that he/she can configure exactly which users are allowed to post messages onto his/her wall. Please note that friends of the user are represented by the room members of his/her profile room, which can change dynamically during runtime.

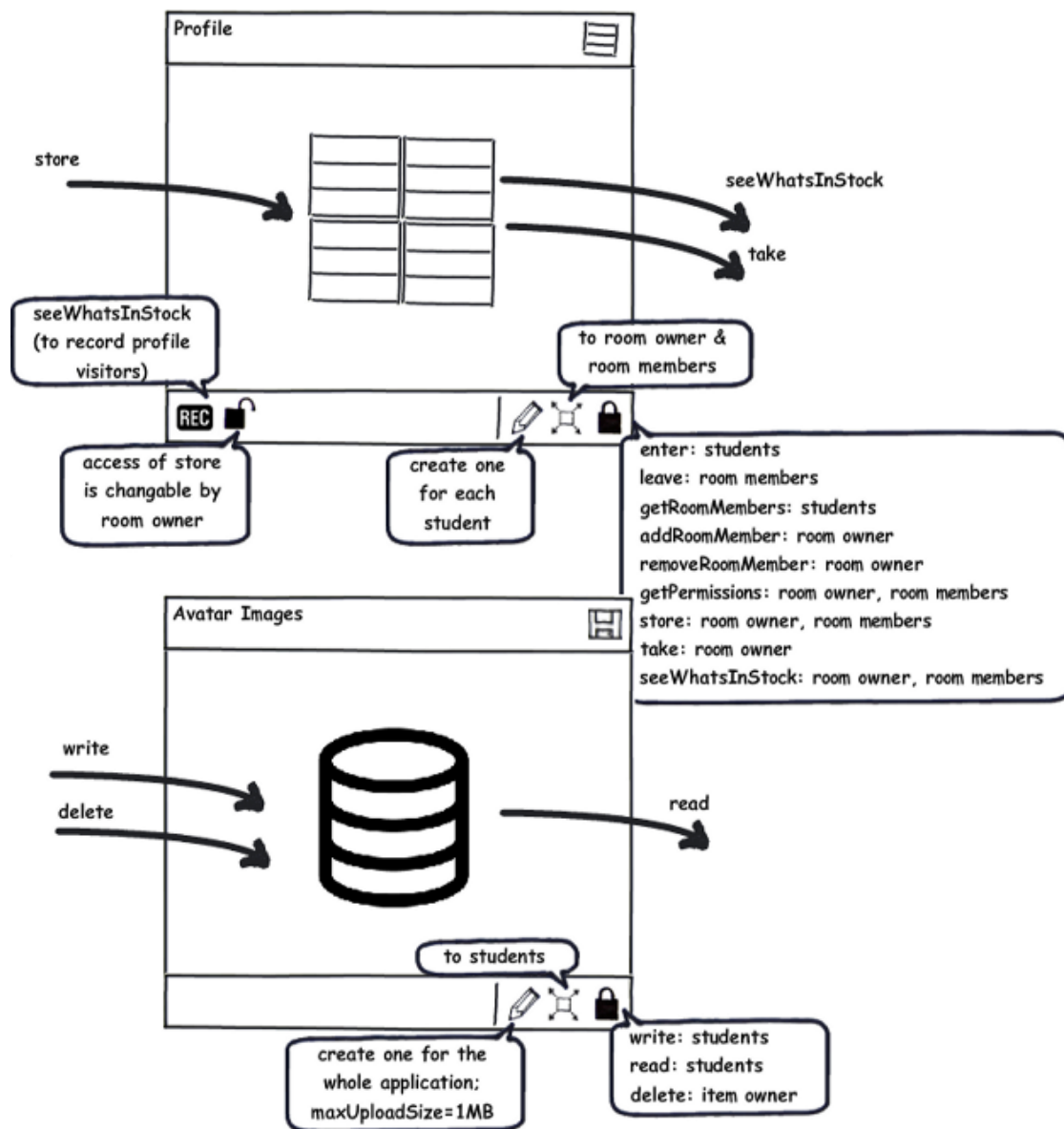


Figure 7.9: P2P OSN scenario - Profile

Figure 7.10 shows the user interface of a profile. It basically consists of some attributes, the avatar of the user, and his/her wall. On the right, there is the possibility to send friend requests and configure which of the user's friends should be allowed to post onto his/her wall. Beneath, all the recorded visits of other users to the profile are listed.

You can also see the tooltip of the `age` attribute that shows the privacy information related to this item (cf. `privacyInformation` in section 5.1.2). Such tooltips are present for every `Item` object that is used within the whole application and show the replication targets as well as to whom the item has already been replicated to.

Welcome to the XMRF Demo Use Case!

[Logout](#)

Search for other profiles:

Search

Name: joe

Age: 25


Gender: ☒ male ☐ female

Save

Replication targets: stefan,max,joe

Replicated to: stefan,max,joe

Change



Post message

Hey, how's it going?

Posted by: max

2013-06-24 18:58:22

Add friend

Friends:

stefan ☒ [remove](#)

max ☒ [remove](#)

Allow posting messages

Profile visitors:

Visitor:	Visit time:
stefan	2013-06-24 18:56:46
max	2013-06-24 18:56:54
max	2013-06-24 18:58:11
max	2013-06-24 18:58:23

Figure 7.10: P2P OSN user interface - Profile

Please note that the avatar image is neither hosted on a web server, nor accessed directly via the file system. Furthermore, it is transferred via the framework's REST interface and "hosted" within the browser itself as a Blob object. The technical details about this method have already been described in section 5.3.2. As mentioned in this section, the URL of the image has a special format, depending on the browser. An example for Firefox can be seen in figure 7.11.

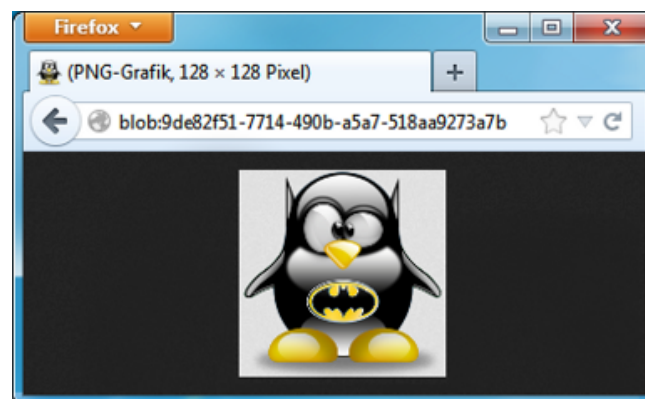


Figure 7.11: Blob URL example

7.1.6 Homework

The last scenario deals with the management of homework (cf. figures 7.12 and 7.13).

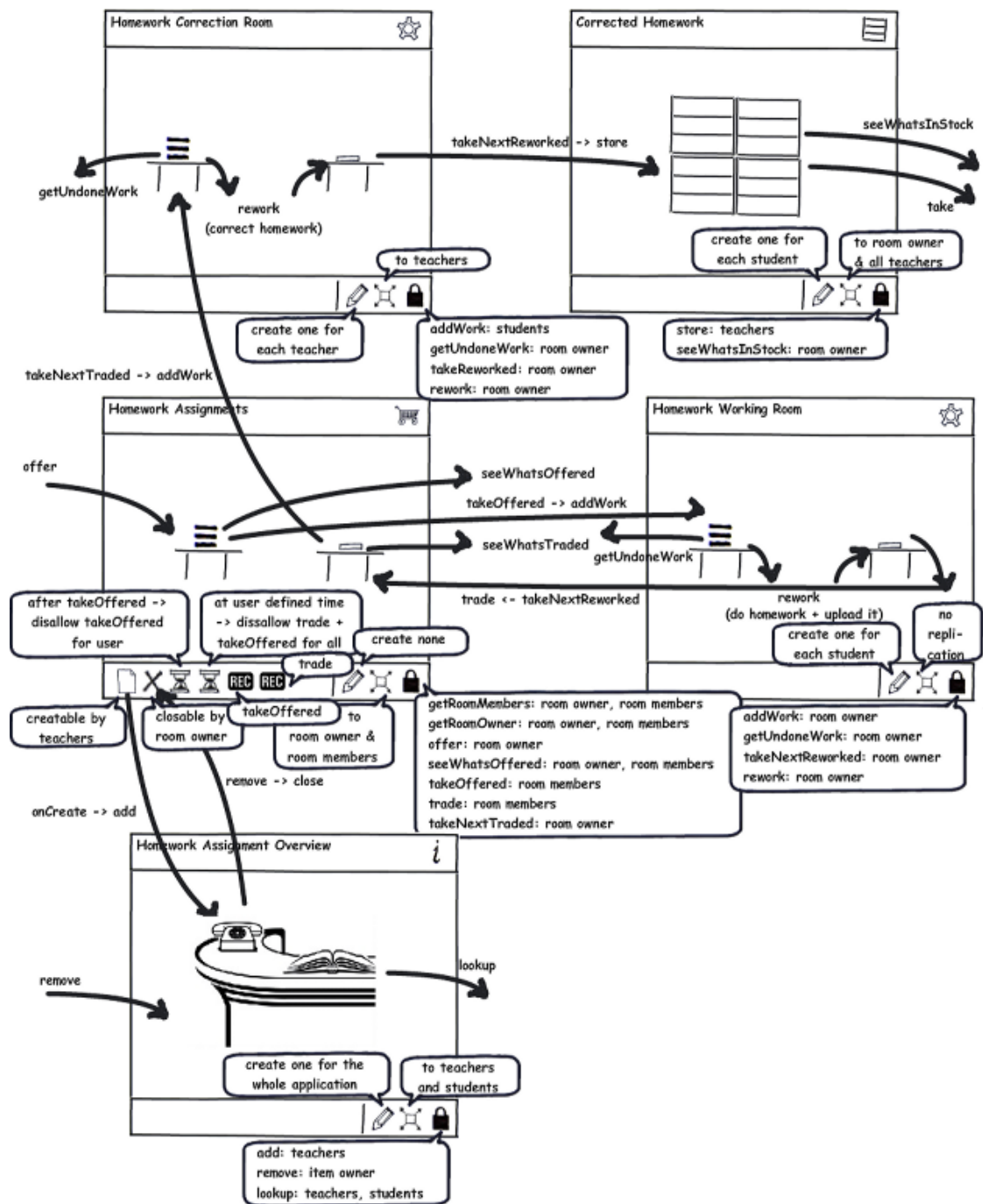


Figure 7.12: P2P OSN scenario - Homework 1/2

Corrected Homework This `StorageRoom` is used for collecting all corrected assignments for a student and thus generated automatically for each student.

Homework Submission Files A `FileRoom` that is created for each teacher automatically. Is used to store all submission files (i.e. *Portable Document Format* (PDF) files) targeting the corresponding teacher. Is replicated only between teachers to avoid cheating by students.

Homework Correction Files A `FileRoom` that is created for each student automatically. Is used to store all correction files (i.e PDF files) targeting the corresponding student.

The whole workflow is centered around an assignment `Item` that is passed along the connections from the initial `Homework Assignments` room to the eventual `Corrected Homework` room. Since this `Item` always has the same ID, the correlation (which is very important in this scenario) can be maintained between the different connections.

Further specialities that can be seen in this scenario are the `AdditionalAction` plugin and the `Scheduler` plugin. As its name implies, the `AdditionalAction` plugin allows to add an additional action to a micro-room. This is done, e.g. at the `Homework Correction Files` room where a new action `addOlderThanDays(Integer days)` is added. This action deletes all files that are older than the specified number of days. Now in order to save disk space, this action will be periodically called by the `Scheduler` plugin. It allows to call actions or to change the access of certain actions in a recurring way. In this case, it calls the newly added action once a day with the parameter 60. This means that every day, all files that are older than 60 days will be deleted from this room.

Figure 7.14 shows the UI of a `Homework Assignments` room from the viewpoint of a student that has to choose one of the available assignments. The “Take” button will call the `takeOffered` action of the room.

Welcome to the XMRF Demo Use Case!

[Logout](#)

Homework title: English Homework

Assignments available:

Write a summary (10 available)

Write an essay (10 available)

Detailed description:

Write an essay about William Shakespeare.

Take

Homework available:

English Homework
(created by englishprof)

German Homework
(created by germanprof)

Figure 7.14: P2P OSN user interface - Take homework assignment

In figure 7.15, one can see the UI for a student to submit a processed assignment. The “Upload” button will show the browser’s file chooser dialog. After selecting the submission file, the UI will upload the file to the teacher’s Homework Submission Files room and put the assignment on the “done” stack in the student’s Homework Working Room.

The screenshot shows a web interface titled "Welcome to the XMRF Demo Use Case!" with a "Logout" link in the top right. The main content area is divided into two panels. The left panel, titled "Assignment: Write an essay", contains a "Detailed description:" section with a text box containing "Write an essay about William Shakespeare." and an "Upload solution" button. The right panel, titled "Unworked homework:", contains a box with the text "Write an essay (assigned by englishprof)".

Figure 7.15: P2P OSN user interface - Upload homework submission

Finally, figure 7.16 shows the UI for correcting a student’s submission from the viewpoint of a teacher. Via the link, the teacher is able to view or download the student’s submission. Please note again that this works via Blobs as described in section 5.3.2. He/She then has to enter the amount of points achieved and to upload a PDF containing all of his/her corrections.

The screenshot shows a web interface titled "Welcome to the XMRF Demo Use Case!" with a "Logout" link in the top right. The main content area is divided into two panels. The left panel, titled "Assignment: Write an essay", contains a "Detailed description:" section with a text box containing "Write an essay about William Shakespeare." Below this, it shows "Submitted homework: /joe/Write_an_essay.pdf" and "Achieved points: 95" with a text input field. An "Upload corrected homework" button is at the bottom. The right panel, titled "Submitted homework:", contains a box with the text "Write an essay (submitted by joe)".

Figure 7.16: P2P OSN user interface - Correct homework submission

7.2 Comparison with Related Work

After outlining the capabilities of the implemented P2P OSN, we will now compare both this implementation and the XVSM Micro-Room Framework itself with the related work, presented in chapters 2.2 and 2.3.

7.2.1 E-Learning P2P Online Social Network

Based on the criteria defined in chapter 2.2.7 we will now compare the implemented P2P OSN with the P2P OSNs already discussed.

Communication: As all the other OSNs, our implementation supports several ways of communication. In specific, these are self-creatable chat rooms, discussions, and the walls on the profile page of each user.

Collaboration: Also collaboration features have been added, as they are easy to implement due to the support of the XVSM Micro-Room Framework. Concretely, these are the possibility to create polls and the whole homework scenario that relies heavily on file collaboration. Further use cases, such as picture exchange or calendars, could be added easily, e.g. by using a `File Room` or implementing a new room type. However, as adding these features wouldn't show new capabilities of the framework, they have been neglected in the current implementation.

Availability: This is probably the most crucial point concerning our implementation. In contrast to the alternatives, the XVSM Micro-Room Framework and thus our implementation of the P2P OSN is not based on encryption. Therefore, data can't be replicated to any peer due to privacy concerns. Our approach thus relies on the proper configuration of the replication targets within the business logic files (cf. `Replicate` plugin). In case of the e-learning P2P OSN, this configuration has been set up very carefully, thus allowing a decent level of availability while preserving privacy at the same time.

For example, messages on the user's wall are replicated among all of his/her friends, so they will be available to each of his/her friends, even if the wall-owning user is offline. Submitted and corrected homework is replicated among all teachers to ensure that those files are available while ensuring that none of the students has the possibility to see submissions of other students. Basically, the replication among all room members seems to be a good default replication strategy, as it ensures both availability and privacy.

Familiar UI: A big advantage of using the XVSM Micro-Room Framework is that the UI is completely independent from the back-end and thus can be implemented with basically any technology. In our case, we used HTML5 to create a Facebook-like user experience. By using the same *Cascading Style Sheets* (CSS), one could theoretically even create a complete copy of Facebook. Concerning the UI, our approach clearly allows the most flexibility among all the approaches compared.

Easy deployment: Until he/she can actually use the application the effort of the user is rather little. The deployment strategy is based on approach b) as presented in section 6.4. This means that we still have a P2P OSN that benefits from improved privacy, but at the same time ease deployment and maintenance by centralizing the configuration files and the UI. So, we as an OSN provider host a website that contains all HTML, module, and business logic files.

If a new user wants to join our OSN, he/she can register at this website that also registers this user to the central authentication service in the background. He/She then only needs to download his/her personal version of the framework and start it. The personalization of the framework can be done automatically by the website. The only thing that needs to be changed is the `peer.userId` and the `peer.password` entry in the `config.properties` file, as described in section 6.1.1.

This is the most convenient way of deploying a P2P OSN, as it consists only of a registration, as well as the download and start of the peer client software. The registration is needed for traditional client-server OSNs like Facebook as well, and the obtaining and starting of the peer client software is required for all P2P-based applications.

Plugin/module support: In our case, additional features can be added easily by providing new module and business logic files on the central web server. They will then be fetched automatically on the next startup of the framework instances on the corresponding peers.

Protection against data abuse: Given that the replication of the rooms is configured thoroughly, data abuse can be excluded in applications created with the XVSM Micro-Room Framework. Concerning the implemented e-learning P2P OSN, all micro-rooms are replicated only among room members that can see the data anyways or across trusted peers such as teachers. Thus, data abuse is avoided effectively, while at the same time availability of this data is still maintained.

Fine-grained access restrictions: As already described, by using the XVSM Micro-Room Framework, arbitrary access restrictions can be implemented. An example for this is given in the profile scenario where the user can specify who is able to post on his/her wall. Additional configurable access restrictions can be defined on any attribute by adding further plugins to the `Storage Room` used or by using an adjusted micro-room implementation for the profile. Also group-based visibility settings can easily be implemented. E.g. when creating a chat room, the user can define who shall be a member of the new room, thus generating a custom group.

Assured deletion: Deletion is enforced by the XVSM Micro-Room Framework itself for every data entry. As described in section 4.7, the framework's replication mechanism ensures that all missed actions (including the delete action) will be performed as soon as the other peer instances get online. Technically, the deletion can't be ensured to 100%, as the other peer instance might stay offline forever and thus won't receive the required replication requests. However, this is a problem inherent to the P2P architecture and can't be solved by the XVSM Micro-Room Framework. Please note that all of the other solutions also encounter this problem. If a peer keeps offline, he/she will receive neither a retraction request (Diaspora), nor the tombstone object (LifeSocial), nor the re-encrypted (due to key revocation now unreadable) data entry (PeerSon, Persona).

Knowledge about physical data distribution: The second novelty of our implementation (beside the broad UI support) is the `Item`-inherent meta data about how many replicas of the item exist and where they are located at. This data can be used to raise user awareness

concerning privacy. In our P2P OSN, we visualize the privacy data just via tooltips to show the capabilities of the XVSM Micro-Room Framework. Of course the visualization could be done in a more noticeable way to achieve a greater effect. However, this is out of scope of this work.

Protection against manipulated clients/requests: Finally, the whole application has been secured rigorously by specifying access restrictions, based on the security mechanisms of the XVSM Micro-Room Framework. These restrictions can be seen in the business logic file visualizations in section 7.1. The security of the framework itself has been discussed thoroughly of section 5.3.1. Thereby, no peer is able to access data that is not located within his/her own framework instance. To gain access to the data of other peers, he/she would either need to forge his/her identity or action requests, which both is prevented by the secured containers and the SSO tokens used for every request.

To sum it up, the capabilities provided by the XVSM Micro-Room Framework allowed us to fulfill all of the criteria defined, even in our simple proof-of-concept implementation. The comparison matrix including also the alternatives already presented in section 2.2.7, is shown in figure 7.17.

		Facebook	Diaspora	Mailbook	PeerSoN	Persona	Safebook	LifeSocial	Our P2P OSN
Functionality	Communication	✓	✓	✓	✓	✓	✓	✓	✓
	Collaboration	✗	✗	⚠	✗	✓	✗	✓	✓
	Availability	✓	✓	✓	⚠	✓	✓	✓	✓
Simplicity	Familiar UI	✓	✓	✗	⚠	✗	✓	✗	✓
	Easy deployment	✓	✗	⚠	✗	⚠	⚠	✓	✓
Extensibility	Plugin/Module support	✓	✗	✗	✗	✓	✗	✓	✓
Privacy	Protection against data abuse	✗	✗	✓	✓	✓	✓	✓	✓
	Fine-grained access restrictions	✓	⚠	⚠	✗	✓	✗	⚠	✓
	Assured deletion	✗	✓	✗	✗	⚠	⚠	✓	✓
	Knowledge about physical data distribution	⚠	⚠	✗	⚠	✗	⚠	✗	✓
Security	Protection against manipulated clients/requests	✓	⚠	⚠	✓	✓	✓	✓	✓

Figure 7.17: Comparison of P2P OSNs including our implementation

7.2.2 XVSM-Micro-Room-Framework

As already stated in section 2.3, there are only very few other P2P frameworks that don't only deal with the communication layer, but also try to abstract the business logic and data layer. Nevertheless, we will now compare our approach with the alternatives described in the beginning of this work.

Concerning TOMSCOP, the XVSM Micro-Room Framework can be seen as a generalisation of this approach. TOMSCOP already allows users to create rooms and to communicate and collaborate over them, which is similar to our framework. However, our approach has far more capabilities, like the ability to create arbitrary room archetypes, extend them with plugins and to orchestrate workflows between rooms. Additionally, the support of different UI technologies, privacy features and the focus on building a secure framework can be seen as further advantages over TOMSCOP.

Considering the P2P Application Framework, our solution has even more advantages. In addition to all benefits already stated, our approach also allows more flexibility concerning deployment. The P2P Application Framework requires index peers that thus limit the possible deployment scenarios. In case of a P2P OSN, this is a strong restriction, as no common user of a OSN is online all the time. Hence, it is very difficult to implement such an application with the P2P Application Framework.

To conclude this section, the XVSM Micro-Room Framework is a new and unique approach to support the creation of P2P applications. None of the few alternatives available has so many capabilities.

7.3 Evaluation of Architecture and Technologies

After showing all the benefits of our approach, we will now point out the weak spots of the framework's architecture and outline the pros and cons we experienced concerning the technologies used.

7.3.1 Architecture

Fortunately, there are just very few weak points concerning the XVSM Micro-Room Framework's architecture. One of these is fault tolerance, which is directly related to the current replication mechanism. The problem is that if an online peer crashes during a distributed transaction, the other peers will have to wait until the transaction timeout is reached and the distributed transaction is rolled back. During this time period the micro-room instances of the executing action are locked at all peers. This is necessary to avoid data inconsistencies that could arise if two actions on the same room instance were called at the same time. So if a user calls another action on this room during the described time period, he will have to wait until the transaction timeout is reached and the room instance is released again. However, this problem only arises because we ensure data consistency at any point in time between all online peers.

Data inconsistencies can only occur in special circumstances: Assume that peer A and B are online and A modifies data D. Now, both A and B go offline and C goes online. As all other peers are offline, C can't synchronize its data and thus a modification of D would cause a replication

conflict as soon as A or B gets online again. As has already been stated in section 4.6.3, this is an inherent problem of asynchronous replication propagation and can only be countered by reconciliation mechanisms. These are out of scope of this work, but required for using the framework in productive environments. Therefore, they are a matter of future work (cf. section 8.3).

Another problem that will occur if a peer crashes is that it doesn't log off from the peer coordination server. Therefore, the peer appears to be online for all other peers until it is removed from the peer coordination server's `Peer Container`. In our implementation, this is done by an own monitor thread running on the peer coordination server that periodically checks whether a peer disconnected. Technically, the thread performs a simple `read` operation on the `Response Container` of each peer. Therefore, we ensure that the system remains available and doesn't enter a permanent blocking state.

Regarding security, the interaction with the framework is protected very well by XVSM Security and the access rules presented in section 5.3.1. Nevertheless, additional actions could be taken to further increase security. E.g. all connections between the UI and the framework instances could be encrypted by using *Hypertext Transfer Protocol Secure* (HTTPS) instead of HTTP. Also, the connections between different framework instances could be encrypted. However, both of these security extensions rely on capabilities of XVSM (i.e. XVSMP and the REST API) and therefore are not directly related to the architecture of the framework.

What also has to be mentioned is that the overall password management has to be implemented in a secure way. Currently, the password of the framework instance as well as the password used for OpenAM queries are stored in clear text in the `config.properties` file. For practical purposes these passwords have to be replaced either by hashed passwords or separate key files.

An additional question that arises is whether users with direct access to the framework instance could gain additional information. These users can access and modify all internal meta and data containers. However, in the intended P2P deployment approach, only that data is replicated to a peer which the peer can read anyways. Thus, no additional knowledge can be achieved by examining the internal containers.

Concerning remote containers on other framework instances, the user can only access those containers accessible by the local framework instance itself. As already stated in section 5.3.1, these containers are rigorously secured via access rules and thus no malicious operations are possible. The user can only perform the same actions as within the UI. Even if the user loosens the access permissions of actions in his/her local business logic files, this won't affect the other framework instances and thus will have no effect. He/She could also configure an action to be replicated to all peers in order to perform this action on the other framework instances. Again, this won't work, as the other instances don't have the modified business logic file and therefore won't accept these requests. This is exactly the same for normal requests. Also, replacing a module won't affect the other peer instances, as they clearly will use their own local modules. As can be seen, it is architecturally impossible to leverage the security concept in a way compromising another framework instance.

Of course, if you think of the other potential deployment scenarios (cf. section 6.4), data abuse possibilities arise. Until now, we have assumed that each framework instance is used by

only one peer. Also in the client-server deployment scenario, other users don't have a chance to compromise the only framework instance, as it runs remotely. However, the "server" provider has full access to the user data, as all internal containers can be read locally. In the decentralized peer servers scenario, the same problem arises, as several "admin" peers that host the framework instances have access to all data of their users. Besides that, they could also send forged requests of their users to the other peer servers. However, Diaspora faces exactly the same problems, showing the fundamental flaws of such an architecture.

7.3.2 Technologies

Considering the technologies used the decisions made in chapter 3 proved to be proper. The micro-room concept is very flexible and allows to add functionality on the fly with a very low level of abstraction. By using Java and XML, the XVSM Micro-Room Framework is kept simple and applicable by a majority of developers at the same time. HTML5 allows the creation of platform-independent UIs that are easy to develop.

XVSM ensured that we could focus on the core aspects of the XVSM Micro-Room Framework, i.e. realizing a declarative P2P framework based on the micro-room concept. It eased the development of the framework significantly by providing a proper communication support via distributed containers using flexible coordinators. The provided persistence support and the REST API further reduced the work load. However, the REST API should be revised to better harmonize with XVSM Security. E.g. the complex REST request shown in listing 6.8 could be simplified significantly.

XVSM Security provides a lot of possibilities to secure XVSM-based containers. In the XVSM Micro-Room Framework, we used only some of its capabilities to ensure a proper level of security. The rules presented in section 5.3.1 are rather general and dynamic as they rely on a lot of runtime information. Therefore, our approach shows that even such "meta rules" can be handled without problems by XVSM Security. However, in the future this XVSM feature should be integrated in a better way. E.g. currently the rule generation is rather unintuitive and the general rule decision algorithm cannot be configured without modifying the core implementation.

Finally, the Paxos Commit support proved to be very valuable for our replication mechanism. As this feature of XVSM is rather new, we needed to integrate it ourselves into a modified version of Mozartspace. Thereby, also the central `Capi` class needed to be extended by a custom `ReplicationCapi` class in order to integrate all of the new features. Since this solution causes unnecessary complex code, the direct integration of the distributed transactions into XVSM is advisable.

Other low-level features such as decentralized peer discovery and management capabilities or an easy-to-use replication support should also be integrated in future version of XVSM.

7.4 Performance Benchmarks

Finally, we will present some performance benchmarks, showing that the framework and the implemented P2P OSN are practicable, i.e. usable in real-world scenarios. Therefore, we mea-

sure the *Central Processing Unit* (CPU) usage, *Random Access Memory* (RAM) usage, request acceptance time, and replication delay in a predefined test scenario.

The scenario consists of a simple chat room, similar to the one in our P2P OSN implementation (cf. section 7.1.2). This room allows anyone to `tell` new messages and is replicated among all users. For the test, one user initiates a variable amount of `tell` requests per second (i.e. 1, 10, 25 and 50). The second parameter of the test scenario is the number of currently online users (i.e. 5, 10 and 20). By combining all parameter settings, a total of 12 different test scenarios can be measured for each of the four criteria stated in the beginning.

Please note that the whole test runs locally on the same machine, as we only want to show that each framework instance can handle large loads regardless of possible network latencies. The test scenario is realized in form of a unit test and thus runs fully automatically. Every framework instance is started in an own thread and uses a different port. After the initialization of all instances, the specified number of requests is sent every second for a total period of 10 seconds. Then, the values measured are shown in the console and all instances are shut down.

For running the benchmarks, we use a system with an Intel Core i7 3820 (quad-core) CPU running at 3.6 GHz and 4 GB of RAM. The operating system used is Windows 7 64-bit edition and the *Java Virtual Machine* (JVM) has version 1.7.0_25-b16.

To get accurate values, each of the 12 test scenarios is run 5 times and the average of all measurements is used. The results will be presented in the following.

7.4.1 Total CPU Usage

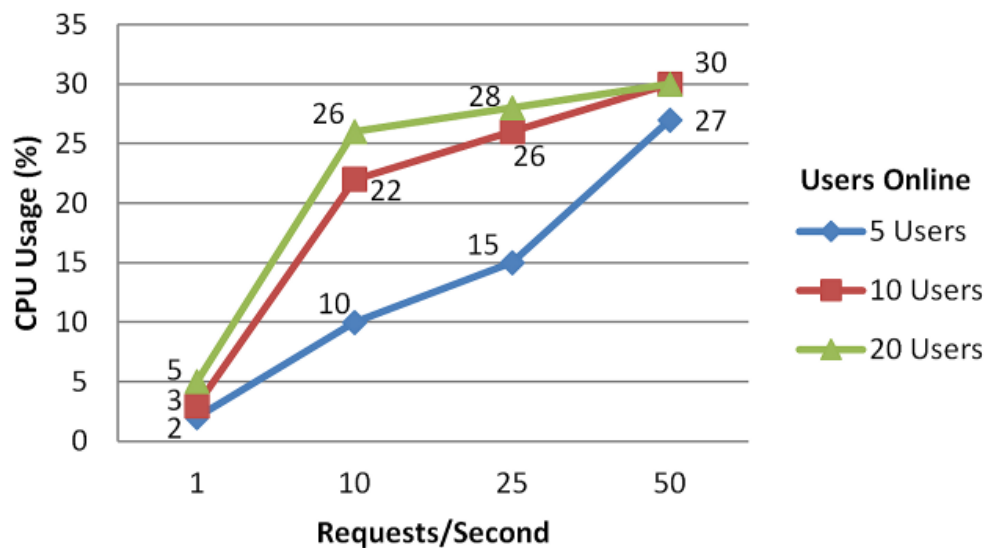


Figure 7.18: Benchmark - Total CPU usage

Figure 7.18 shows the total CPU usage of all framework instances together. It can be seen that the framework causes almost no usage when only 1 request is sent per second, regardless

of how many users are online. The values shown are similar to other peer clients. Skype for example requires 2% CPU usage on the same testing system for sending 1 request per second to 1 other user.

If more than 10 requests are sent per second, the CPU usage rises significantly, but it is still far from critical. Especially if you keep in mind that the values correspond to the CPU usage of all framework instances together. Thus, the average usage on each peer is certainly below the shown values.

The figure also shows the following interesting phenomenon: The number of requests per second only seems to have an effect on the CPU usage if there are less than 10 users online. With more users, the usage stagnates around 25 to 30%, regardless of the requests per second. As we can see later, this border of 30% is never crossed, due to limitations within the framework's current replication mechanism, i.e. the framework runs at its limit.

So concerning CPU usage, there is no reason to worry about the framework's scalability and real-life usability.

7.4.2 Total RAM Usage

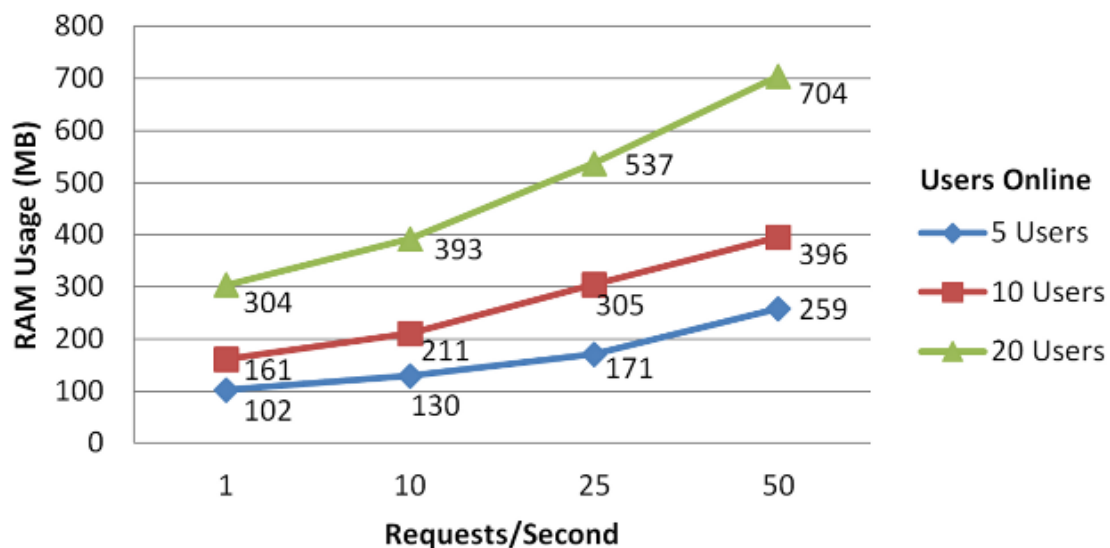


Figure 7.19: Benchmark - Total RAM usage

Regarding the total RAM usage of all framework instances, figure 7.19 gives similar insights. By looking at the “5 users” line, it can be seen that the framework uses only a decent part of memory (i.e. less than 100 MB on a single peer on average). For comparison, the Skype client uses 108 MB for sending 1 request per second to 1 user.

Considering more online users, the total RAM consumption of course rises, as more framework instances have to be generated for testing. Also note, that for twice the number of users (and thus framework instances) not exactly twice the amount of RAM is required, but a little less. This is because of already loaded Java classes and libraries.

What can be concluded by the figure is that the RAM usage is rather low on the average peer. However, the number of requests per second clearly impacts the RAM consumption. The more requests are processed, the more RAM is required. However, the peak of 704 MB is still low if you consider that a total of 10,000 replicated objects ($50 \text{ requests / second} * 10 \text{ seconds} * 20 \text{ replication targets}$) have been created.

The number of online users won't influence the RAM usage significantly if you neglect the RAM used by additionally created instances. What can be mentioned is that the "20 users" curve seems to have a little stronger upward trend than the others. So, the replication of very many requests to more users seems to increase the RAM usage disproportionally, but this effect is minimal.

To sum it up, just as the CPU usage, the RAM usage gives no cause to feel concerned about.

7.4.3 Request Acceptance Time

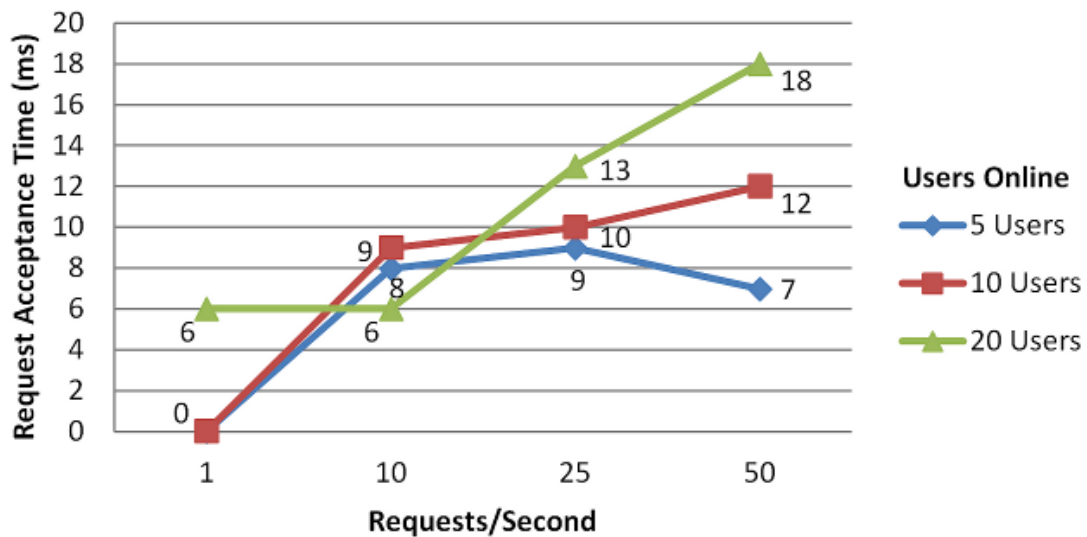


Figure 7.20: Benchmark - Request acceptance time

In figure 7.20, one can see the request acceptance time, which is the time that is required by the framework itself to accept a new request. This value is very important, as it also describes the time that the user will have to wait in our P2P OSN if the UI sends an asynchronous request. Even if it is only a `Oneway` request (cf. section 6.3.2), the UI will still wait and thus block until the receipt of this request will have been confirmed.

By looking at the values of the figure, it can be concluded that the request acceptance time is in general rather low and thus acceptable. Also, it seems to vary regardless of the number of requests per second or the users online. One could interpret a linear trend into the results, dependent on the requests per second. However, the numbers are so dense that it seems more like they vary randomly.

Summarizing the results, the framework is always capable of accepting new requests, regardless of the load that it is currently under.

7.4.4 Replication Delay

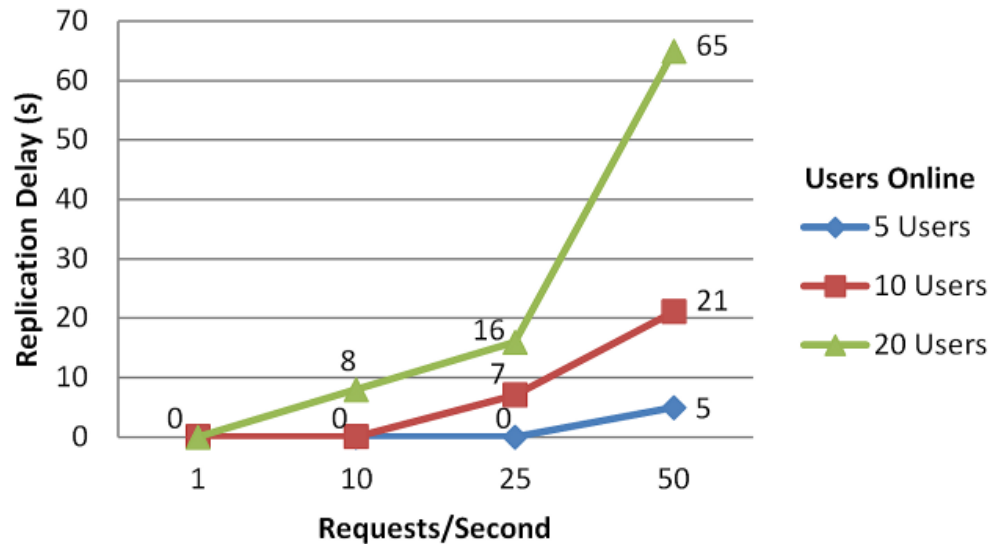


Figure 7.21: Benchmark - Replication delay

The measurements of the replication delay is shown in figure 7.21. By replication delay, we mean the total time that all framework instances require for replication, after the last request has been sent. So, after the last `tell` request has been sent by the initiating peer, time is measured until each one of the other peers has also processed this request.

For asynchronous requests, this is the time after that the response object (for the last request) can be read and thus will trigger the callback function. For synchronous requests, the UI would block exactly for this time. E.g. the last of the overall 100 requests (10 requests / second * 10 seconds) will block the UI for 8 seconds, if 20 users are currently online.

The reason for this is that the framework's replication mechanism (for currently online users) can't handle so many requests at once. Please remember that the replication between online peers is done via distributed transactions and thus very resource-consuming. It seems as if the current implementation of the Paxos Commit protocol (on which the framework's replication is based on) doesn't scale very well.

As can be seen, up to 100 transaction participants per second (25 requests / second * 4 other users) cause no problem. However, just 190 transaction participants per second (10 requests / second * 19 other users) cannot be handled and thus generate a total delay of 8 seconds for the replication of all requests. All delay times greater than 0 seconds (i.e. all test scenarios including more than 100 transaction participants) are unacceptable for productive systems.

However, to put the results into perspective, this limit of 100 transaction participants is not shared between all peers, but applies for every single one. The whole load of the distributed transaction has to be handled by the initiating peer that acts as a transaction coordinator. He/She is responsible for writing a copy of the request to the `Replication Request Container` of all other peers and for managing the overall transaction. Thus, the workload on the other peers is very low, as they only need to execute their local request. They therefore could start transactions with up to 100 participants per second themselves.

Also, a simultaneous replication of one request to more than 100 peers is rather uncommon. First, the XVSM Micro-Room Framework is not intended to be used for replicating data to everybody. Moreover, data should only be replicated to those peers that are allowed to read it anyway, in order to ensure availability, while maintaining privacy. Second, all of these 100 peers would have to be online at the same time, as only online peers are included in the distributed transactions. Offline peers that go online don't affect the load of the other peers significantly, as they run the whole synchronization logic themselves.

Concluding the benchmarks, it can be stated that the XVSM Micro-Room Framework is suitable for real-life scenarios. The CPU and RAM usage are very low, which allows the framework to be used on arbitrary computers or even smart phones. The request acceptance time always keeps below 20 milliseconds and thus indicates that the framework can accept new requests immediately even under heavy load. Finally, it has to be stated that the current implementation of the replication doesn't scale very well and thus should be revised in the future (cf. section 8.3). Nevertheless, the limit of 100 transaction participants per second is sufficient for common small- to mid-scale scenarios.

Future Work

Some drawbacks of the XVSM Micro-Room Framework have already been mentioned in the previous chapters. In this section all of them are summarized and it is outlined, how they could be resolved in the future.

8.1 Security

The current implementation of XVSM Security uses a central service (i.e. an OpenAM server instance) for authentication. Alternative ways of authentication would increase the framework's deployment flexibility. E.g. it could then be used in a pure P2P way without any servers involved.

However, if you don't have a centralized trusted authentication service, it will be difficult to ensure the authenticity of a peer. Common approaches rely on a PKI that itself requires a *Certificate Authority* (CA) to verify the certificates used by the peers. As one can see, this CA itself is again some kind of centralized trusted authentication service and thus doesn't solve the problem.

Only few approaches exist to establish a pure P2P-based authentication protocol. One that seems to be very promising, relies on the Byzantine Generals Problem and is described in [PETHCR06]. In the approach, the peers more or less overtake the functionality of the CA together. Just as in the Byzantine Generals Problem, at most m traitors/malicious peers can be overcome if at least $3m + 1$ loyal generals/honest peers exist.

8.2 Peer Management

Besides the authentication service, a second central service exists, i.e. the peer coordination server. If it was replaced/extended by a pure P2P approach (e.g. like Gnutella), it would again increase the framework's flexibility regarding deployment. Also, if no servers existed, this would enable full partition tolerance, as already stated in section 4.2.10.

As peer discovery and coordination are rather low-level concerns, they should be integrated into the space-based middleware rather than the XVSM Micro-Room Framework. Thus, in the future, XVSM should be extended by these features.

8.3 Replication

In section 7.4.4, we could see that the current implementation of the replication mechanism of the framework is sufficient in most cases, but could be improved regarding large-scale scenarios. Also, reconciliation mechanisms have to be added, as stated in section 7.3. Again, as replication between peers is a rather low-level task, XVSM's capabilities (i.e. an extended version of the current replication support [Hir12]) should be used.

Therefore, several different replication strategies (e.g. synchronous, asynchronous or hybrid-like replication, as currently implemented) could be offered by the space-based middleware. These could be integrated into the framework by simply writing several new `ReplicationManager` classes. The framework then could let the user decide which approach he/she wants to use, e.g. by configuring it within the `config.properties` file.

8.4 Workflow Support

As has been stated in section 4.7, currently only 1:1 connections are possible between different actions. In the future, this feature could be extended by 1:n or even m:n connections. So, one action could trigger several different actions and could itself be called by multiple other actions.

8.5 Business Logic & Module Adaption

Currently, business logic and module files can only be altered in a way that doesn't require additional reconciliation logic. Concerning business logic files, this includes everything except changing the name of scenarios or micro-rooms. Those two operations would cause the `TranslationUnit` to create new room instances with the new names and thus would lead to data inconsistencies.

Regarding module files, adding new module files or deleting unused module files is possible. Also, minor changes within module files (e.g. bug fixes) that don't affect any containers, are allowed. However, greater changes, including especially the renaming of actions or internally used containers, are very likely to cause problems with the persisted containers.

To allow larger adjustments, additional information about the adjustments needs to be provided to the framework, e.g. by a `.diff` file. This information then has to be parsed by the framework to properly adjust containers restored from persistence and thus maintain data consistency.

8.6 XML Modeler

As has been addressed in section 6.1.2 and can also be seen in section 7.1, the graphical notation of micro-rooms and thus business logic files overall is far more intuitive than the (already simple) XML file. Therefore, in the future a XML modeling tool should be developed, that allows end users to easily configure and connect micro-rooms in a graphical way. The tool then creates the XML output file that represents the business logic file of the designed scenario and thus can be used by the framework.

8.7 Case Study

Finally, the current evaluation focuses on the alternatives and the technical limits of the XVSM Micro-Room Framework. The user-related benefits haven't been verified yet, which is why a user testing should be carried out in the future. Thereby, not only its simplicity can be verified, but also valuable feedback can be collected.

Conclusion

The initial goal of this work was to create an alternative to existing P2P OSNs that fulfills all of the analyzed requirements, i.e. functionality, simplicity, extensibility, privacy, and security. As these requirements are the same for almost all P2P applications and no other P2P frameworks exist that cover all of them, we decided to create one ourselves.

The resulting XVSM Micro-Room Framework comes with many benefits, including the support of both arbitrary communication and collaboration features. It isn't based on encryption, as PKIs are not feasible in the real world at the moment. To ensure data availability while maintaining privacy, the framework relies on intelligently distributed replication. For configuring privacy on the user-level, it allows to define fine-grained access restrictions. Additionally, it raises user awareness concerning privacy by providing detailed privacy information about every data item to the user.

Generating applications with the XVSM Micro-Room Framework is separated cleanly between end users, application developers, and UI designers. All of the artifacts required are modular and thus can be added to the framework easily. The UI can be developed with basically any technique, that allows to access the REST interface of the framework, thus allowing a broad flexibility. To prevent abuse by unauthorized users, all external interfaces have been rigorously secured. Regarding the framework's deployment possibilities, many different scenarios are possible, including pure P2P, client-server, thin clients, or decentralized peer servers.

By implementing the initially intended P2P OSN, we could show that the XVSM Micro-Room Framework can be used to create complex P2P applications that fulfill all of the stated requirements. From a technical point of view, the performed benchmarks indicate that applications generated with the XVSM Micro-Room Framework are also capable of handling large loads.

The next step will be to implement the XML modeler and evaluate the usability of the framework. With the thus gathered feedback, the XVSM Micro-Room Framework will be further improved.

References

- [AAB⁺10] Tristan Allard, Nicolas Anciaux, Luc Bouganim, Yanli Guo, Lionel Le Folgoc, Benjamin Nguyen, Philippe Pucheral, Indrajit Ray, Indrakshi Ray, and Shaoyi Yin. Secure Personal Data Servers: A Vision Paper. *Proc. VLDB Endow.*, 3(1-2):25–35, 2010.
- [ADBS09] Jonathan Anderson, Claudia Diaz, Joseph Bonneau, and Frank Stajano. Privacy-enabling Social Networking over Untrusted Networks. In *Proceedings of the 2Nd ACM Workshop on Online Social Networks*, WOSN '09, pages 1–6, New York, NY, USA, 2009.
- [Bar10] Martin-Stefan Barisits. Design and Implementation of the next Generation XVSM Framework – Operations, Coordination and Transactions. Master's thesis, Vienna University of Technology, 2010.
- [BB11] Oleksandr Bodriagov and Sonja Buchegger. Encryption for Peer-to-Peer Social Networks. In *SocialCom/PASSAT*, pages 1302–1309, 2011.
- [BBS⁺09] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: An Online Social Network with User-defined Privacy. *SIGCOMM Comput. Commun. Rev.*, 39(4):135–146, 2009.
- [BC09] Justin Becker and Hao Chen. Measuring Privacy Risk in Online Social Networks. In *Proceedings of the Web 2.0 Security and Privacy 2009 Workshop (W2SP)*, 2009.
- [BD09] Sonja Buchegger and Anwitaman Datta. A Case for P2P Infrastructure for Social Networks - Opportunities & Challenges. In *Proceedings of the Sixth international conference on Wireless On-Demand Network Systems and Services*, WONS'09, pages 149–156, Piscataway, NJ, USA, 2009.
- [BHG86] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [BMM02] Özalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proceedings of the*

22nd International Conference on Distributed Computing Systems (ICDCS'02), ICDCS '02, pages 15–21, Washington, DC, USA, 2002.

- [BRCA09] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing User Behavior in Online Social Networks. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference, IMC '09*, pages 49–62, New York, NY, USA, 2009.
- [Bre00] Eric A. Brewer. Towards Robust Distributed Systems (Abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, PODC '00*, pages 7–, New York, NY, USA, 2000.
- [Brü13] Andreas Brückl. Relaxed non-blocking Distributed Transactions for the eXtensible Virtual Shared Memory. Master's thesis, Vienna University of Technology, 2013.
- [BSBK09] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All Your Contacts Are Belong to Us: Automated Identity Theft Attacks on Social Networks. In *Proceedings of the 18th International Conference on World Wide Web, WWW '09*, pages 551–560, New York, NY, USA, 2009.
- [BSVD09] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peer-SoN: P2P Social Networking: Early Experiences and Insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems, SNS '09*, pages 46–52, New York, NY, USA, 2009.
- [CDJ⁺13] Stefan Craß, Tobias Dönz, Gerson Joskowicz, Eva Kühn, and Alexander Marek. Securing a Space-Based Service Architecture with Coordination-Driven Access Control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76–97, 2013.
- [CDJK12] Stefan Craß, Tobias Dönz, Gerson Joskowicz, and Eva Kühn. A Coordination-Driven Authorization Framework for Space Containers. In *Proceedings of the 2012 Seventh International Conference on Availability, Reliability and Security, ARES '12*, pages 133–142, Washington, DC, USA, 2012.
- [CHKSC13] Stefan Craß, Jürgen Hirsch, Eva Kühn, and Vesna Sesum-Cavic. An Adaptive and Flexible Replication Mechanism for Space-Based Computing. In *International Conference of Software and Data Technology*, Reykjavik, Iceland, 2013.
- [CK12] Stefan Craß and Eva Kühn. A Coordination-based Access Control Model for Space-based Computing. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC '12*, pages 1560–1562, New York, NY, USA, 2012.
- [CMO11] Leucio Antonio Cutillo, Refik Molva, and Melek Önen. Safebook: A distributed privacy preserving online social network. In *WOWMOM*, pages 1–3, 2011.

- [CMS09a] L. A. Cutillo, R. Molva, and T. Strufe. Safebook: A Privacy-preserving Online Social Network Leveraging on Real-life Trust. *Comm. Mag.*, 47(12):94–101, 2009.
- [CMS09b] Leudo Antonio Cutillo, Refik Molva, and Thorsten Strufe. Privacy Preserving Social Networking Through Decentralization. In *Proceedings of the Sixth International Conference on Wireless On-Demand Network Systems and Services*, WONS’09, pages 133–140, Piscataway, NJ, USA, 2009.
- [Cra03] Lorrie Faith Cranor. ’I didn’t buy it for myself’ Privacy and Ecommerce Personalization. In *Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, WPES ’03, pages 111–117, New York, NY, USA, 2003.
- [Cra10] Stefan Craß. A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification. Master’s thesis, Vienna University of Technology, 2010.
- [CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66, New York, NY, USA, 2001.
- [Dö11] Tobias Dönz. Design and Implementation of the next Generation XVSM Framework – Runtime, Protocol and API. Master’s thesis, Vienna University of Technology, 2011.
- [DW11] Michael Dürr and Kevin Wiesner. A Privacy-Preserving Social P2P Infrastructure for People-Centric Sensing. In Norbert Lüttenberger and Hagen Peters, editors, *KiVS*, volume 17 of *OASICS*, pages 176–181, 2011.
- [GA05] R. Gross and A. Acquisti. Information Revelation and Privacy in Online Social Networks. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, pages 71–80, 2005.
- [Gel85] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
- [GGS⁺11] Kalman Graffi, Christian Groß, Dominik Stingl, Daniel Hartung, Aleksandra Kovacevic, and Ralf Steinmetz. LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks. In *Proceedings of the IEEE Consumer Communications and Networking Conference*, 2011.
- [GKB12] Benjamin Greschbach, Gunnar Kreitz, and Sonja Buchegger. The Devil is in the Metadata - New Privacy Challenges in Decentralised Online Social Networks. In *Proceedings of SESOC 2012*, 2012.
- [GL06] Jim Gray and Leslie Lamport. Consensus on Transaction Commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

- [GTF08] Saikat Guha, Kevin Tang, and Paul Francis. NOYB: Privacy in Online Social Networks. In *Proceedings of the first workshop on Online social networks*, WOSN '08, pages 49–54, New York, NY, USA, 2008.
- [HBKC11] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL Landscape: A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 427–444, New York, NY, USA, 2011.
- [Hir12] Jürgen Hirsch. An Adaptive and Flexible Replication Mechanism for MozartSpaces, the XVSM Reference Implementation. Master's thesis, Vienna University of Technology, 2012.
- [HPD05] Manfred Hauswirth, Ivana Podnar, and Stefan Decker. On P2P Collaboration Infrastructures. In *Proceedings of the 14th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, WETICE '05, pages 66–71, Washington, DC, USA, 2005.
- [HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.
- [JP10] António L. Jorge and Filipe P. Porfírio. Building an academic social network: For bologna mobility. In *Proceedings of the 3rd Workshop on Social Network Systems*, SNS '10, pages 1:1–1:5, New York, NY, USA, 2010.
- [KM04] Tomomi Kawashima and Jianhua Ma. TOMSCOP - A Synchronous P2P Collaboration Platform over JXTA. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04) - Volume 7*, ICDCSW '04, pages 85–90, Washington, DC, USA, 2004.
- [KW08] Balachander Krishnamurthy and Craig E. Wills. Characterizing Privacy in Online Social Networks. In *Proceedings of the first workshop on Online social networks*, WOSN '08, pages 37–42, New York, NY, USA, 2008.
- [Lam01] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.
- [LCC⁺02] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and Replication in Unstructured Peer-to-peer Networks. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 84–95, New York, NY, USA, 2002.
- [LOP05] Javier Lopez, Rolf Oppliger, and Guenther Pernul. Why Public Key Infrastructures have failed so far? *Internet Research*, 15(5):544–556, 2005.
- [MKS10] Richard Mordinyi, Eva Kühn, and Alexander Schatten. Space-Based Architectures As Abstraction Layer for Distributed Business Applications. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '10, pages 47–53, Washington, DC, USA, 2010.

- [MPHD06] Alan Mislove, Ansley Post, Andreas Haeberlen, and Peter Druschel. Experiences in Building and Operating ePOST, a Reliable Peer-to-peer Application. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 147–159, New York, NY, USA, 2006.
- [MPV06] Vidal Martins, Esther Pacitti, and Patrick Valduriez. Survey of data replication in P2P systems. Rapport de recherche RR-6083, INRIA, 2006.
- [Nas10] Robayet Nasim. Privacy-enhancing Access Control Mechanism in Distributed Online Social Network. Master's thesis, Royal Institute of Technology, 2010.
- [PETHCR06] Esther Palomar, Juan M. Estevez-Tapiador, Julio C. Hernandez-Castro, and Arturo Ribagorda. A P2P Content Authentication Protocol Based on Byzantine Agreement. In *Proceedings of the 2006 International Conference on Emerging Trends in Information and Communication Security*, ETRICS'06, pages 60–72, Berlin, Heidelberg, 2006.
- [Pro11] Christian Proinger. Design and implementation of a rest-interface for mozartspaces. Bachelor Thesis, Vienna University of Technology, 2011.
- [RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001.
- [RGK⁺05] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '05, pages 73–84, New York, NY, USA, 2005.
- [Rip01] M. Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 99–, Washington, DC, USA, 2001.
- [SB11] Hossein Saiedian and Dan S. Broyles. Security Vulnerabilities in the Same-Origin Policy: Implications and Alternatives. *Computer*, 44(9):29–36, 2011.
- [WHR⁺08] James Walkerdine, Danny Hughes, Paul Rayson, John Simms, Kiel Gilleade, John Mariani, and Ian Sommerville. A Framework for P2P Application Development. *Comput. Commun.*, 31(2):387–401, 2008.
- [WL03] Chonggang Wang and Bo Li. Peer-to-Peer Overlay Networks: A Survey. Technical report, The Hong Kong University of Science and Technology, Hong Kong, 2003.

- [YJS⁺11] Cheng Yong, Wu Jiangjiang, Mei Songzhu, Wang Zhiying, Jun Ma, Ren Jiangchun, and Yan Ke. Mailbook: Privacy-Protecting Social Networking via Email. In *Proceedings of the Third International Conference on Internet Multimedia Computing and Service*, ICIMCS '11, pages 90–94, New York, NY, USA, 2011.
- [Zar12] Jan Zarnikov. Energy-efficient Persistence for Extensible Virtual Shared Memory on the Android Operating System. Master's thesis, Vienna University of Technology, 2012.

Web References

- [1] Global Web Index. Social Platforms GWI 8 Update: Decline of Local Social Media Platforms. <http://www.globalwebindex.net/social-platforms-gwi-8-update-decline-of-local-social-media-platforms/>. Accessed: 2013-05-07.
- [2] Bundeskanzleramt. Datenschutzgesetz 2000 Art. 2 § 4 Z. 2. <http://www.ris.bka.gv.at/Dokument.wxe?Abfrage=Bundesnormen&Dokumentnummer=NOR12017604>. Accessed: 2013-05-07.
- [3] Facebook. Terms of Use. <https://www.facebook.com/legal/terms>. Accessed: 2013-05-07.
- [4] Facebook. Data Use Policy. <https://www.facebook.com/about/privacy/your-info>. Accessed: 2013-05-07.
- [5] The Guardian. NSA Prism program taps in to user data of Apple, Google and others. <http://www.guardian.co.uk/world/2013/jun/06/us-tech-giants-nsa-data>. Accessed: 2013-06-30.
- [6] The New York Times. Four Nerds and a Cry to Arms Against Facebook. http://www.nytimes.com/2010/05/12/nyregion/12about.html?_r=0. Accessed: 2013-05-08.
- [7] The Diaspora Project. Architecture Overview. http://wiki.diaspora-project.org/wiki/Architecture_overview. Accessed: 2013-05-08.
- [8] Jo Bager. Soziale Netzwerke: Diese Facebook-Alternativen sind einen Blick wert. <http://www.spiegel.de/netzwelt/web/soziale-netzwerke-alternativen-zu-facebook-a-868293-2.html>. Accessed: 2013-05-08.
- [9] GitHub Diaspora. Installation Guides. <https://github.com/diaspora/diaspora/wiki/Installation-Guides>. Accessed: 2013-05-08.
- [10] GitHub Diaspora. Diaspora on a Windows Box. <https://github.com/diaspora/diaspora/wiki/Diaspora-on-a-Windows-Box>. Accessed: 2013-05-08.

- [11] Diaspora Alpha. How many users are in the Diaspora network? <https://diasp.eu/stats.html>. Accessed: 2013-05-08.
- [12] The Diaspora Project. FAQ for Users. http://wiki.diaspora-project.org/wiki/FAQ_for_Users. Accessed: 2013-05-08.
- [13] Peer to Peer Framework. A Toolbox for Building P2P-based Social Networks. <http://p2pframework.com/?lang=en>. Accessed: 2013-05-10.
- [14] Google Official Blog. Update on Google Wave. <http://googleblog.blogspot.co.at/2010/08/update-on-google-wave.html>. Accessed: 2013-07-11.
- [15] The Wall Street Journal. Europeans to Facebook: Where's My Data? <http://blogs.wsj.com/digits/2011/09/29/europeans-to-facebook-wheres-my-data/?mod=WSJBlog>. Accessed: 2013-05-17.
- [16] TIOBE Software. TIOBE Programming Community Index for July 2013. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Accessed: 2013-07-11.
- [17] w3schools.com. HTML5 Introduction. http://www.w3schools.com/html/html5_intro.asp. Accessed: 2013-05-21.
- [18] BBC News. Napster must stay shut down. <http://news.bbc.co.uk/2/hi/entertainment/1893904.stm>. Accessed: 2013-05-24.
- [19] w3resource. JSONP Tutorial. <http://www.w3resource.com/JSON/JSONP.php>. Accessed: 2013-06-22.
- [20] Mozilla Developer Network. window.postMessage. <https://developer.mozilla.org/en-US/docs/Web/API/window.postMessage>. Accessed: 2013-06-22.
- [21] W3C. Cross-Origin Resource Sharing. <http://www.w3.org/TR/cors/>. Accessed: 2013-06-22.
- [22] Can I use... Can I Use Cross-Origin Resource Sharing? <http://caniuse.com/cors>. Accessed: 2013-06-22.
- [23] Mozilla Developer Network. Blob. <https://developer.mozilla.org/en-US/docs/Web/API/Blob?redirectlocale=en-US&redirectslug=DOM%2FBlob>. Accessed: 2013-06-22.
- [24] Can I use... Can I Use Blob constructing? <http://caniuse.com/blobbuilder>. Accessed: 2013-06-22.
- [25] XStream. Converters. <http://xstream.codehaus.org/converters.html>. Accessed: 2013-06-20.

[26] Code Project. WCF - Message Exchange Patterns. <http://www.codeproject.com/Articles/566543/WCF-Message-Exchange-Patterns-MEPs>. Accessed: 2013-06-08.

Business Logic File XSD

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3   <xs:element name="scenario">
4     <xs:complexType>
5       <xs:sequence>
6
7         <xs:element name="rooms" minOccurs="1" maxOccurs="1">
8           <xs:complexType>
9             <xs:sequence>
10              <xs:element name="room" minOccurs="1" maxOccurs="unbounded">
11                <xs:complexType>
12                  <xs:sequence>
13
14                    <xs:element name="owner" type="xs:string" minOccurs="0"
15                      ↪ maxOccurs="1" />
16
17                    <xs:element name="members" minOccurs="0" maxOccurs="1">
18                      <xs:complexType>
19                        <xs:sequence>
20                          <xs:element name="member" type="xs:string" minOccurs="1"
21                            ↪ maxOccurs="unbounded" />
22                        </xs:sequence>
23                      </xs:complexType>
24                    </xs:element>
25
26                    <xs:element name="plugins" minOccurs="0" maxOccurs="1">
27                      <xs:complexType>
28                        <xs:sequence>
29                          <xs:element name="plugin" minOccurs="1" maxOccurs="
30                            ↪ unbounded">
31                          <xs:complexType>
```

```

29         <xs:sequence>
30             <xs:element name="property" minOccurs="0" maxOccurs="
                 ↳unbounded">
31                 <xs:complexType>
32                     <xs:attribute name="key" type="xs:string" use="
                         ↳required" />
33                     <xs:attribute name="value" type="xs:string" use="
                         ↳required" />
34                 </xs:complexType>
35             </xs:element>
36         </xs:sequence>
37         <xs:attribute name="type" type="xs:string" use="
                 ↳required" />
38         <xs:anyAttribute namespace="##any" processContents="lax
                 ↳" />
39     </xs:complexType>
40 </xs:element>
41 </xs:sequence>
42 </xs:complexType>
43 </xs:element>
44
45 </xs:sequence>
46 <xs:attribute name="name" type="xs:string" use="required" />
47 <xs:attribute name="type" type="xs:string" use="required" />
48 <xs:attribute name="generateFor" type="xs:string" />
49 <xs:anyAttribute namespace="##any" processContents="lax" />
50 </xs:complexType>
51 </xs:element>
52 </xs:sequence>
53 </xs:complexType>
54 </xs:element>
55
56 <xs:element name="connections" minOccurs="0" maxOccurs="1">
57     <xs:complexType>
58         <xs:sequence>
59             <xs:element name="connection" minOccurs="1" maxOccurs="
                 ↳unbounded">
60                 <xs:complexType>
61                     <xs:sequence>
62                         <xs:element name="from" minOccurs="1" maxOccurs="1">
63                             <xs:complexType>
64                                 <xs:attribute name="roomName" type="xs:string" use="
                                     ↳required" />
65                                 <xs:attribute name="actionName" type="xs:string" use="
                                     ↳required" />
66                                 <xs:attribute name="correlationVar" type="xs:string" />
67                                 <xs:attribute name="correlationVal" type="xs:string" />
68                             </xs:complexType>
69                         </xs:element>

```

```

70     <xs:element name="to" minOccurs="1" maxOccurs="1">
71         <xs:complexType>
72             <xs:attribute name="roomName" type="xs:string" use="
73                 ↳required" />
74             <xs:attribute name="actionName" type="xs:string" use="
75                 ↳required" />
76             <xs:attribute name="correlationVar" type="xs:string" />
77             <xs:attribute name="correlationVal" type="xs:string" />
78         </xs:complexType>
79     </xs:element>
80 </xs:sequence>
81 </xs:complexType>
82 </xs:element>
83 </xs:schema>
84
85 </xs:sequence>
86 </xs:complexType>
87 </xs:element>
88 </xs:schema>

```

Listing A.1: XSD of business logic files