Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

TU WIEN FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mobile Peer Model

A mobile peer-to-peer communication and coordination framework - with focus on scalability and security

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jörg Schoba, BSc
Matrikelnummer 01026309

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 10. Oktober 2017

Jörg Schoba	Eva Kühn

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Mobile Peer Model

A mobile peer-to-peer communication and coordination framework - with focus on scalability and security

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jörg Schoba, BSc
Registration Number 01026309

to the Faculty of Informatics

at the TU Wien

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

Vienna, 10th October, 2017

Jörg Schoba   Eva Kühn

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Jörg Schoba, BSc
Donaufelderstraße 8/1/12 1210 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.

Wien, 10. Oktober 2017

Jörg Schoba

Acknowledgements

I thank my thesis advisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eva Kühn and the thesis assistant Projektass. Dipl.-Ing. Stefan Craß of the Space Based Computing Research Group from the TU Wien for their support from the beginning of the diploma thesis. Without their expert knowledge the completion of the work would have hardly been possible. Special appreciation is also extended to Stefan for his great reviews of the thesis.

I also thank Martin Planer, Matthias Schwayer and Gerson Joskowicz for their participation in the technical board meetings, where they discussed issues and passed their comments on important decisions regarding the thesis. Furthermore I want to mention Konrad Steiner for testing early versions of the implemented framework and giving feedback on it. I hope he will also finish his thesis with success.

A big thank you goes to Peter Tillian, my best friend since years, who made this collaboration work a success. The synergy we developed in our whole study at the TU Wien was to our mutual advantage and a big factor for our accomplishments.

Last but not least I want to pay a tribute to my mother Dagmar, my father Heimo and all other members of my family for the support in all the years of school and study. Moreover I express my gratitude to my girlfriend Christina and her family. I hope after finishing this long period of hard work I can find more time for all of you and I want you to be aware that without your support and patience all this would not have been possible. Thanks!

Abstract

Peer-to-peer (P2P) communication and coordination is an emerging paradigm in next generation distributed systems. An analysis of the state of the art of conceptualized and existing P2P communication and coordination frameworks, which are applicable for widely distributed systems and support deployment on current mobile devices, reveals that most of these approaches do not provide sufficient developer support, are only applicable for specific use cases, are designed for local area networks, are deprecated or not easily deployable on mobile devices. Therefore, a coordination framework for mobile devices based on P2P communication is designed and implemented, intended to support developers in creating internet-scale, distributed mobile applications. The reference implementation is conducted for *Android* devices, because *Android* currently has the highest market share of mobile operating systems. Nevertheless, the system is designed to be implementable also on other mobile platforms and generally is compliant to mobile constraints. Compared to a system running on a desktop or server machine these constraints include, e.g., the battery consumption, the limited processing power and the limited network connectivity. These aspects have been considered in the technical system design. Also, because P2P communication shall be possible in an internet-scale network, appropriate security measures have been taken. Moreover, the system is designed to be able to traverse any network address translation (NAT) of a network without having to configure a router or firewall, which enables P2P communication in any network constellation.

This work delivers a framework that supports a developer of a mobile P2P application by providing the middleware components that abstract communication and coordination logic of the application and let the developer focus on application logic only.

Kurzfassung

P2P-basierte Kommunikation und Koordination ist ein zukunftsträchtiges Paradigma in verteilten Systemen der nächsten Generation. Eine Analyse der derzeit existierenden Frameworks für P2P-basierte Kommunikation und Koordination von mobilen Geräten in einem global verteilten Netzwerk zeigt, dass die meisten Ansätze entweder für Entwickler keine wirklichen Vorteile bringen, nur für spezielle Anwendungsfälle konzipiert sind, nur in lokalen Netzwerken angewandt werden können, veraltet sind oder nicht auf mobilen Geräten implementiert werden können. Deshalb wurde in dieser Arbeit ein auf P2P-Kommunikation basierendes Koordinations-Framework für mobile Geräte entworfen und implementiert. Die Referenz-Implementierung wurde für die Android-Plattform entwickelt, weil Android zurzeit den größten Marktanteil unter den mobilen Betriebssystemen hält. Nichtsdestotrotz ist das Framework so konzipiert, dass es auch auf anderen gängigen, mobilen Betriebssystemen implementiert werden kann und ist generell auf mobile Geräte optimiert. Einschränkungen im Vergleich zu einem System, welches auf einer Desktop- oder Server-Maschine läuft, sind z.B. die beschränkte Akkulaufzeit, die limitierte Prozessorgeschwindigkeit und die limitierte Netzwerkkonnektivität. Diese Aspekte wurden beim technischen Entwurf des Framework berücksichtigt. Da eine Kommunikation auch über das öffentliche Internet möglich sein soll, wurden auch spezielle Sicherheits-Maßnahmen getroffen, welche für eine sichere Kommunikation in diesem Fall unerlässlich sind. Das Framework wurde so entworfen, dass eine Kommunikation in Netzwerken mit beliebigen NAT-Konstellationen möglich ist, ohne einen Router oder eine Firewall konfigurieren zu müssen.

Das entwickelte Framework unterstützt den Programmierer bei der Entwicklung einer mobilen Applikation, welche auf P2P-Kommunikation basiert und liefert die Middleware-Komponenten, welche die Kommunikations- und Koordinationslogik abstrahieren und es dem Entwickler ermöglichen, sich auf die Applikationslogik zu fokussieren.

Contents

Abstract ix

Kurzfassung xi

Contents xiii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Problem statement . . . 2
1.3 Aim of the work . . . 2
1.4 Structure of the work . . . 3

2 Analysis of existing approaches and background technologies 5
2.1 Review process . . . 5
2.2 P2P systems characteristics . . . 6
2.3 Unstructured P2P overlay networks . . . 7
2.4 General purpose P2P protocols . . . 23
2.5 Coordination frameworks and models . . . 30

3 Requirements and selection of background technology 33
3.1 Requirements . . . 33
3.2 Evaluation and selection of background technology . . . 37

4 The Peer Model 43
4.1 Characteristics . . . 43

5 Design 47
5.1 Mobile profile of the Peer Model (PM) . . . 47
5.2 Architectural overview and separation of engineering tasks . . . 50
5.3 Architecture of communication and identity management . . . 54
5.4 Architecture of serialization . . . 59
5.5 Security concept . . . 63
5.6 Scalability concept . . . 69

6 Implementation **75**
6.1 Class overview . . . 75
6.2 Communication and identity management . . . 79
6.3 Serialization . . . 83
6.4 Security . . . 88
6.5 Scalability . . . 92
6.6 Integration with partner work . . . 94

7 Proof-of-concept implementation of mobile application **97**
7.1 Background . . . 97
7.2 Design . . . 98
7.3 Implementation . . . 99

8 Evaluation **105**
8.1 Scalability evaluation . . . 105
8.2 Performance of end-to-end (E2E) encryption . . . 112
8.3 Security assessment . . . 113
8.4 Fulfilment of imposed requirements . . . 117

9 Conclusion **119**
9.1 Summary . . . 119
9.2 Lessons learned . . . 120
9.3 Future work . . . 120

List of Figures **123**

List of Tables **124**

Acronyms **125**

Bibliography **129**

CHAPTER 1

Introduction

P2P communication and coordination approaches became popular in the early 2000s with file-sharing platforms, which did not require centralized servers or only a part of the network was centralized. Ever since, P2P communication has been a good alternative to usual server-client communication in specific scenarios. By design such systems are more dynamic, scalable and failure-resistant. Although the time of file-sharing networks seems to be over, P2P paradigms are still in wide use, e.g. in communication networks like instant messaging (IM) and voice over IP (VoIP). Moreover, coordination amongst different self-contained nodes seems to be a concept with much potential when integrated with the P2P paradigm. Kühn et al. have proposed the Peer Model (PM) in [KCJ+13] as a tool to design systems that incorporate both P2P communication and coordination patterns for communicating nodes. This work has been the inspiration to develop a framework which abstracts underlying P2P communication and coordination logic to a level that lets developers concentrate on the application logic and along with that enables them to create widely distributed applications much easier on mobile platforms.

1.1 Motivation

Internet-scale, P2P based communication networks like *Skype* are also available on mobile platforms and there exist mobile applications to e.g. share computational power of the device like *BOINC*[1]. This shows that mobile devices nowadays have enough potential to act as self-contained nodes in a network and also have enough computational power to contribute. Still, mobile devices have several constraints compared to stationary computers like e.g. limited connectivity or battery consumption. This means, that such applications need to be specially tailored for mobile usage. Despite there exists a large variety of protocols for P2P communication and there also exist frameworks and models

[1] `https://play.google.com/store/apps/details?id=edu.berkeley.boinc&hl=en` accessed 03.2017

to coordinate the participating peers, research shows that most of the frameworks and protocols are not designed for or have not yet been implemented on current mobile platforms.

1.2 Problem statement

Available resources on mobile devices are increasing continuously. This enables implementation of computational complex applications that can make use of steadily rising amounts of network bandwidth. It seems worthwhile to design models and frameworks for easy development of mobile applications that follow the P2P paradigm and offer functionality for coordination of those mobile peers. In such networks, application and coordination logic is solely running on the mobile peer and those peers communicate with other peers. Unfortunately, it seems that no frameworks for mobile application developers exist, which support the development of such applications that might be distributed over internet-scale P2P networks on current mobile platforms. Such a framework has to be compliant with the constraints of such devices. The technical challenges are to design the framework to use resources like battery and processing power efficiently as well as to be able to handle an instable network connectivity while still offering enough developer support. Also, security measurements have to be implemented in the framework to appropriately secure the communication over a public network. The public network might only be accessible using NAT to a local area network (LAN). Such constellations usually require configuration of a router or firewall to enable P2P communication traversing the NAT. To be able to use applications based on the framework in any network setup, it has to be designed so that no such configuration is necessary, because this is for example not possible in a public LAN. Furthermore, probably needed central components of the network based on the designed framework have to be scalable to cope with increasing load and to guarantee an appropriate amount of reliability.

1.3 Aim of the work

The aim of this work is to design and implement an open-source framework to develop mobile applications that are based on P2P communication of active peers running the application logic. The framework shall be designed to support internet-scale communication networks. Additionally, it shall offer possibilities to specify coordination patterns amongst those peers and shall be implementable on currently popular mobile operating systems.

Detailed requirements on the framework are provided in Chapter 3. Generally, the framework shall be open-source and implementable on popular mobile platforms with an appropriate amount of effort. Probably needed central parts shall be scalable and communication over the network shall offer a suitable amount of security for public networks. In comparison to not using the framework, a developer shall have a clear benefit in terms of abstractions, which allow the developer of a mobile P2P application to

focus on application logic and not on coordination and communication. Already existing background technology that has been researched and evaluated is selected considering the proposed requirements and integrated as components of the framework.

1.4 Structure of the work

This master thesis is realized as a cooperation work with Peter Tillian [Til17]. Responsibilities have been defined for both for the whole process of the literature reviews and also the software engineering process of the reference implementation. The general focus points of this work are scalability and security of the designed framework. The whole conceptualization and implementation of the communication layer are also part of this work. Detailed information on the distribution of work are made clear in each of the chapters of this work and each of us deals with his focus points in his own thesis. References are used whenever appropriate. Despite each work can be understood independently, all details of the whole project are clear when reading both.

Both works are structured the same way, so it is also possible to read chapter by chapter in each work, following the references provided. After the introductory first chapter, the second chapter is a systematic literature review of existing background technologies for usage in the framework. This involves P2P frameworks and protocols as well as coordination frameworks and models. Here, focus points have been set for both of us and are explained in the chapter. The requirements for the framework to be designed are specified in chapter three. Here also the found background technology is evaluated and selected based on those requirements. Chapter four provides insights on the PM that has been selected as coordination model to be used in the framework. Chapter five and six deal with the design and the implementation phases of the software engineering process, specifically the design and programming of the reference implementation for *Android*. Additionally, a proof-of-concept implementation of a mobile application that is based on the implemented framework is done by both of us in an own software engineering process in chapter seven. In chapter eight, evaluations on the framework are done related to the focus points of the thesis. The last chapter concludes about the whole work and gives incitements about future work.

CHAPTER 2

Analysis of existing approaches and background technologies

In this chapter an analysis of existing approaches on P2P networks and frameworks is done. The research is performed in a systematic way and shall firstly help to get an overview of the related work and background technologies in this field. Then, existing approaches, techniques, frameworks and protocols that could be used as a reference, core component or supporting technology for the proposed framework shall be identified. In the selection process the found approaches, frameworks and protocols shall be evaluated respecting the requirements imposed to the Mobile Peer Model (MPM) framework in Chapter 3.

2.1 Review process

The review is framed by specific inclusion and exclusion criteria as well as specific sources and search queries. Sources for the literature review have been mainly the search engine of IEEE Xplore[1], Springer[2], ACM Digital Library[3], ScienceDirect[4] and Google Scholar[5] as meta search index. The search terms used for querying papers proposing probably useful approaches were mainly: "peer-to-peer networks", "mobile peer-to-peer", "peer-to-peer middleware", "android peer-to-peer", "iOS peer-to-peer", "peer-to-peer protocol", "internet peer-to-peer", "mobile peer-to-peer communication", "coordination framework" and "mobile coordination framework".

[1] `http://ieeexplore.ieee.org/` accessed 09.2016
[2] `http://link.springer.com/` accessed 09.2016
[3] `http://dl.acm.org/` accessed 09.2016
[4] `http://www.sciencedirect.com/` accessed 09.2016
[5] `https://scholar.google.at/` accessed 09.2016

2.1.1 Inclusion criteria

Included are articles from scientific journals, books or book chapters and conference and workshop proceedings, which deal with communication protocols and frameworks that could contribute to this work, not necessarily also including coordination models. Here, only approaches are enclosed which are applicable in widely distributed systems over wide area networks (WANs) like the internet. When gathering papers from the named widely acknowledged publication platforms, care is taken that the papers also have been cited by other authors as a measurement for fair quality and appropriate impact factor. Additionally, uniform resource locators (URLs) from official websites, code repositories and wikis of found technologies are included if necessary, always providing dates of last access to the sites.

2.1.2 Exclusion criteria

Not included in the research process are informal surveys or information from online forums, weblogs and wikis with no respective references to the source of information. Approaches that deal with protocols and frameworks only applicable in LANs like mobile ad hoc networks operating over Wi-Fi, near field communication (NFC), bluetooth and comparable technology are also excluded.

2.2 P2P systems characteristics

P2P is a decentralized communication model. In comparison to standard client-to-server communication, where a client sends a request and a server answers with a response, in the P2P communication model each participant may act as a server and client at the same time. When talking about a *peer* in this work, this refers to that kind of fully qualified participant in a P2P network. Each peer may initiate a new communication session with another one. In typical P2P networks these peers may join and disconnect form the network at any time without destructing the network structure or disrupting other peers. In a P2P system, all or almost all of the needed computational power, storage and bandwidth are provided by the joint peers. Because of this, P2P networks can grow almost without any limitation. In a client-server environment it would be necessary to upgrade or scale the existing server infrastructure. The peers are not controlled by a central authority, but usually by independent individuals that join the network voluntarily. P2P networks are usually also more resilient to faults and attacks, because there are no or only a few nodes that are critical to the system's operation. Nonetheless, the P2P concept faces many challenges that originate from the decentralized design of these systems itself including manageability, security and jurisdiction. In the past there have been many applications based on the P2P paradigm, many of them designed for large-scale data sharing, content distribution and other multicast distributed systems. Typically, P2P overlay networks are divisible in two structural categories. First there are structured overlays, where content and logic is placed in a structured way on all joined peers. That means it is controlled by an algorithm which peers are responsible for which

content and logic and where it is possibly mirrored. Secondly there are unstructured overlays, where responsibilities are placed arbitrarily. Peers can join and leave the network without prior knowledge of the topology and without affecting other peer's responsibilities. Furthermore, one can categorize P2P networks as pure, hybrid or centralized. *Pure* in this context means that the network is completely decentralized with no server or hierarchically superior components. Decentralized networks consist only of completely equitable peers. Hybrid P2P networks consist of more than just one flat structure of equal peers, usually they have one more layer of overlying super-peers or servers, which are not fixed and which provide coordination services for the underlying peers like indexes for data localization. In many cases, these servers or super-peers are hosted by volunteers that upgrade their normal peer to a super-peer and probably use an own software for these components. In contrast to that, centralized P2P networks have a consistently addressable and dedicated server or server farm that provides the coordination services. These characteristics have been extracted and can be comprehended in more detail in [RD10] and [LCP+05]

Peter Tillian [Til17] in his literature research and evaluation focuses on structured P2P overlay networks while this work focuses on unstructured ones. Furthermore, there exist some more P2P frameworks and protocols that offer not just P2P communication but more services like e.g. peer discovery, peer grouping and so on. There exist also some important communication protocols that are not purely P2P but also interesting as background technologies. The found approaches that could not be categorized in structured or unstructured P2P communication protocols have been split up for detailed evaluation by both of us also in this chapter.

2.3 Unstructured P2P overlay networks

In this section an overview of existing P2P protocols used to create unstructured overlay networks is provided. The focus lies on protocols that also have gained publicity and have been used by P2P applications. Also, a comparison and analysis of these technologies is done. When gathering information about the approaches, already the requirements for the proposed MPM framework are considered, including such aspects as publication of the protocol, development state, existing ports to mobile platforms or NAT bypassing techniques. Scalability and security aspects, which are my responsibilities for the MPM framework, are also examined for the found technologies.

2.3.1 Napster

The idea of P2P networks was first implemented and used in a global way with the concept of *Napster* (see [LCP+05]) in 1999. The purpose of the system was decentralized file-sharing with a central registry for filenames, the network therefore can be categorized as centralized. The central registry offered a search facility and features for contributing peers to publish the names of the files available at their local storage. In that way, the demand for bandwidth at the central registry could be decreased drastically compared to

a system where data is directly hosted at a server. Users were able to search for keywords in the filenames on the registry and got the addresses for matching peers to download the files from. The actual sharing of files was then accomplished by a direct connection between the matched peers. Figure 2.1 shows the communication model used by *Napster*, where P defines a peer, S a central server, Q a request with a search query, R a response with addresses of matching peers and D the actual data exchange between peers. The architecture of *Napster* can be seen as a combination of client-server and P2P models because it uses a centralized server or server pool for discovery of resources. The client software typically has built-in server addresses to connect, but allows also adding of further servers. *Napster* also implemented P2P messaging, chat rooms and hot lists. Hot lists represent a kind of favorite peer register, which every peer can maintain. Regarding traversal of NATs, *Napster* used the server as a connection broker for clients, described by Son and Livny in [SL03]. When an uploading peer is located behind a firewall or NAT, the downloading peer asks the server to send a notification to the uploading peer to establish a connection and actively upload the file.

Finally, a lawsuit forced *Napster* to shut down their application in 2001 because of copyright infringement. *Napster* announced itself bankrupt in 2002. However, the idea of P2P file-sharing was carried on and improved in other systems. The main problem of *Napster* might be its bottleneck and single point of failure in the static centralized registry for file discovery. The brand of *Napster* was later sold to a digital media company and is now used as name for their music service[6].

As one might deduce easily, the architecture of *Napster* is vulnerable against censorship, technical failure and denial of service (DoS) attacks, because of its static central components like discussed in [ATS04]. Also, one can think about that it is easy to harvest internet protocol (IP) addresses of participating users by searching for popular files and collecting the matching peers. Later on, attacks could be driven against those peers. Because *Napster* focuses on media files only, distribution of malware was not easy to achieve, like stated in [TC06], but of course there is no guarantee that these media files are not altered or polluted.

There exist open-source implementations of the *Napster* server like *OpenNap*[7]. This server is implemented as a console application for multiple platforms. Unfortunately there has not been any ongoing development since many years. *Slavanap*[8] is another server implementation with a current version from 2013, but is only available for the *Microsoft Windows* platform and is closed source. A few other server implementations could be found, but none where open-source and in progressive state. Regarding *Napster* client software, many existing open-source implementations for various platforms exist, including implementations in Java like *XNap*[9] or *XNapster*[10] with current versions from

[6] `http://napster.com` accessed 09.2016
[7] `http://opennap.sourceforge.net/` accessed 09.2016
[8] `https://sourceforge.net/projects/slavanap2/` accessed 09.2016
[9] `https://sourceforge.net/projects/xnap/` accessed 09.2016
[10] `https://sourceforge.net/projects/xnapster/` accessed 09.2016

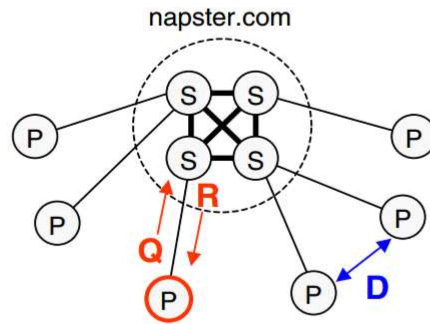


Figure 2.1: *Napster* communication model [SGG03]

2015. In the research no ports of *Napster* client applications for current mobile platforms were found, but most likely trying to port a Java implementation to *Android* would be the best choice here. For further insights on *Napster* see [LCP+05], [RD10] and [SMR12].

2.3.2 Gnutella

In contrast to the *Napster* P2P approach, the *Gnutella* open-source protocol (see the protocol definition[11]) does not support a client-server search mechanism, file discovery and exchange are both done by a pure P2P paradigm. Like stated in [LA10] and [LCP+05], the network forms an overlay of equal peers that can send messages to neighbouring peers. In the original protocol, *query flooding* was used for content and host discovery. This means that a peer sends a *query* message for a specific file to all its known neighbours, the neighbours send the query to their neighbours and so on until one peer matches the query and sends a *query hit* response to the originating peer back on the same route. After that, the actual file exchange happens between those two peers over hypertext transfer protocol (HTTP). Also, *query* messages are tagged with a time-to-live (TTL) property, which is decreased at each forwarding. If the TTL property reaches zero, the message gets discarded. Moreover, a message identifier (ID) is propagated with each message to be able to avoid rebroadcasting. Using these mechanisms, the network load can be reduced. For discovering new peers, specific *ping* messages are used. Peers that receive a *ping* message send a *pong* response to the sender, which then can add the peer as a valid neighbour. Using the same *flooding* approach for *ping* and *pong* messages like for query messages, a big set of currently available peers can be discovered. For initial peer discovery, applications that are based on the *Gnutella* protocol are usually shipped with a fixed number of well-known *Gnutella* hosts that are very likely available. Starting by connecting to them, the peer can then build the required search-neighbourhood. Figure 2.2 shows the *flooding* based communication model of *Gnutella*. The left one of the marked peers represents the initiator of a query. This structure overcomes the problem

[11] `http://rfc-gnutella.sourceforge.net/developer/stable/` accessed 10.2016

with the centralization described in the *Napster* network, but suffers from higher network load and the fact that existing files in the network might not be found, because the search method does not explore the whole search-space. Also, by design of the protocol, several disjoint *Gnutella* overlay networks might exist in parallel around the globe.

To overcome firewall and NAT problems, so-called *push* messages have been introduced in the protocol. Like Zeinalipour-Yazti stated in [ZY02], whenever it is not possible for a peer to connect directly to another peer to download a file, it sends a *push request* back on the same route where it received the answer to the query. If the servant peer then receives the *push request* containing the address of the requesting peer, it opens a connection to this peer and sends the data.

It was also investigated in [ZY02] that security was not a real concern when developing the *Gnutella* protocol. It is vulnerable for distributed denial of service (DDoS) attacks trough spamming peers with faked *query hit* messages, which then try to download non-existent files. Also, like mentioned, it is easy to discover a large amount of peers by using *ping* messages. This technique can be used by malicious attackers for IP address harvesting and performing malicious attacks on the discovered hosts like it has been shown by [SGG03]. Also, there exists an option for attackers to distribute malicious code to peers by using the described *push request* mechanism. When faking a *query hit* message from a peer behind a firewall, the attacked peer might send a *push request* to the attacker. Usually the requester will then blindly accept what is sent in the established connection by the sender peer, e.g. not the wanted file but a virus or trojan.

There are several applications supporting the *Gnutella* protocol but only a few that are not proprietary and in further development. One such example, that also supports multiple platforms, is *gtk-gnutella*[12]. Nevertheless, it is written in C, so it will not be easily runnable on current mobile devices unless it is cross-compiled for the concrete architecture used (e.g. advanced RISC machine (ARM) processor on Android). A further ongoing project is *WireShare*[13], which was written in Java. There have even been some efforts on porting *Gnutella* to the *Android* platform with some adjustments, like proposed in e.g. [TVHVL13].

There also exists a successor protocol named *Gnutella2*[14], but this protocol differs from the original *Gnutella* protocol. For more insights see [LA10, LCP$^+$05]. It was completely rewritten and introduced a hybrid part with super-peers called hubs. Every peer keeps only one or more connections to such hubs. Hubs maintain a register of the files that peers host and also distribute these meta-data to other hubs, as well as lists of available hubs. When querying for files, a peer first gets a list of hubs from its connected hub and then sequentially contacts those hubs with its search query, until possibly one hub has an address of a peer providing the file. The number of allowed hubs to contact in these search process is limited by the protocol. The used transport protocol in *Gnutella2* is

[12] `http://gtk-gnutella.sourceforge.net/` accessed 09.2016
[13] `https://sourceforge.net/projects/wireshare/` accessed 09.2016
[14] `http://g2.doxu.org/index.php/Main_Page` accessed 09.2016

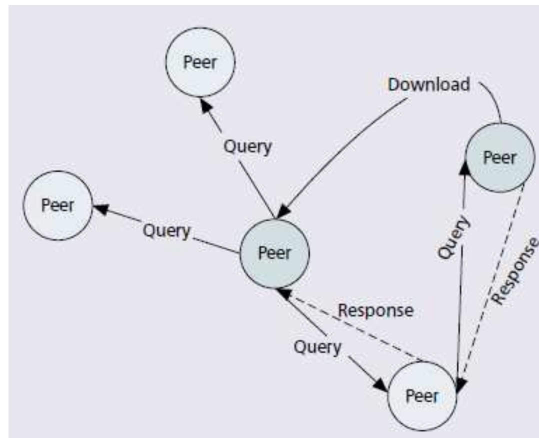


Figure 2.2: *Gnutella* communication model [LCP+05]

the user datagram protocol (UDP). *WireShare* and *gtk-gnutella* also support *Gnutella2*. There exists an implementation of a *Gnutella2* client for *Android* named *DroidG2*[15] from 2011, based on a C++ implementation of the protocol for Linux. Nevertheless, the application seems not to be in further development and seems not to work properly on every device. No implementations of the *Gnutella* protocol for the *iOS* platform could be found.

2.3.3 Freenet

Freenet[16], which also originated around the year 2000, is also an open-source P2P overlay network, but it differs from e.g. *Napster* and *Gnutella* in many aspects. *Freenet* is heavily focused on security and anonymity. It can be seen as a semi-structured overlay network, because individual peers do not decide which data of the network they store, but also there is no global algorithm for data location. In fact, peers provide a part of their local storage to the network. Then, whenever a data file gets uploaded to the network, it gets hashed and by this gets a unique content-hash key (CHK) identifying the file. The files are then stored on the peer that is most responsible for that CHK. This happens by a routing mechanism where a peer forwards the file to the neighbour which is responsible for storing files hashed to the most similar key. In a nutshell that means a file that one uploads might get stored at another peer which is reachable through neighbourhoods and holds similar CHKs. So after time, specific peers automatically specialize on specific keys. Moreover, the actual file is encrypted by a randomly generated encryption key. The idea behind that is that peer hosters could deny any knowledge of the *Freenet* data stored on their machine. For querying files, a descriptive text of the data is necessary. Therefore, so-called signed-subspace keys (SSKs) are generated out of a short descriptive text, that

[15] `https://play.google.com/store/apps/details?id=org.toxiclab.droidg2` accessed 09.2016
[16] `https://freenetproject.org/` accessed 09.2016

the uploader provides for the namespace where the corresponding CHKs are stored. Also, for each of these namespaces an asymmetric key-pair is generated. So, everyone can verify that a file really originates from this space by verifying with the associated public key. Therefore, the space owner signs the file keys with the space's private key. With this mechanism only the owner can upload and update files in the namespace. By design of the network also big files can be split up into many smaller parts. For each part an own CHK is generated. The owner then provides an indirect file, which points to all of these parts. Querying data happens with the same routing mechanism used for uploading data. When requesting a specific CHK, the query gets forwarded to only the one neighbour which is responsible for the most similar keys. If no more better neighbours are reachable by a peer, the peer returns a backtracking failed request message to the sender. The sender might then choose an alternative neighbour. Figure 2.3 shows the *Freenet* routing model, where peer A is the initiator of a request and numbers from 1 to 12 show the routing trace. When finally a peer that holds the file is found, the file is sent back on the inverse route to the originator. Also, it gets cached on each of the peers it passes on the way to the initiator of the request, so for next queries not the whole route has to be taken again. Mentionable is also the aspect that no peer can know if the sender of a message is the originator or only a peer that forwards it, so anonymity of originating peers can be retained. A similar approach to reduce network load like in *Gnutella* is used. All messages are tagged with unique message IDs to prevent re-forwarding and a TTL is added to limit the hops a message can take. *Freenet* avoids any kind of central index like in *Napster* and also avoids a distribution of a message to all neighbouring peers like in *Gnutella*. Therefore, it might be less attackable and more scalable than those two approaches. Nevertheless, one could think about that it still might be vulnerable to man-in-the-middle (MitM) attacks or malware injection, which is also mentioned in [LCP+05].

Not only file-sharing applications have been implemented based on *Freenet*, the whole network is designed in an approach similar to the world wide web (WWW). The counterpart of a website in the *Freenet* is a *Freesite*[17]. On a *Freesite* there can be links to other *Freesites* or other data reachable by a data key. An email system[18] and an IM system[19] are examples of further implementations based on *Freenet*.

In the ongoing development of *Freenet*, two running modes have been implemented, namely *Opennet* and *Darknet* mode, like explained in [RSHS14]. In the *Opennet*, new joining nodes connect to the network through publicly known seed nodes that then forward their requests based on the location of the new node to establish a local neighbourhood. In the *Darknet* mode neighbours that want to connect have to be invited or granted access by joint nodes and have to be trusted explicitly. As a result of this design, it can be made difficult for attackers on the network, e.g. governmental institutions, to harvest addresses of nodes of the network. Also, if some *Darknet* nodes hold connections to

[17] `https://wiki.freenetproject.org/Freesite` accessed 09.2016
[18] `https://freenetproject.org/freemail.html` accessed 09.2016
[19] `https://freenetproject.org/frost.html` accessed 09.2016

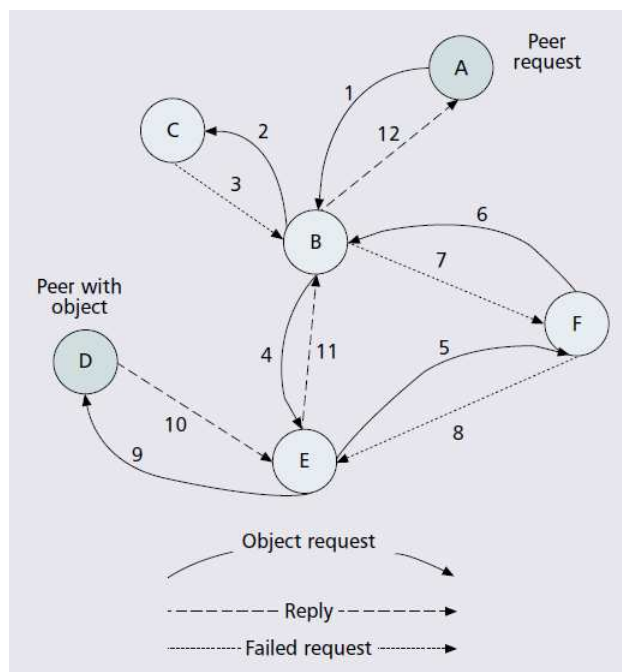


Figure 2.3: Freenet routing model [LCP+05]

Opennet nodes, all of their trusted *Darknet* neighbours can also access all the content of the *Opennet* but not vice versa. With this hybrid approach, benefits of both networks can be used. When installing a *Freenet* client there are usually two possibilities to connect to others, there exists a built-in, predefined search mechanism for stranger neighbours to connect to, or there exists the mentioned manual possibility to enter addresses of trusted friends to connect to.

The *Freenet* protocol is still in development and is available for multiple platforms. Even though it is written in Java and also available for Linux, during the research no successful efforts to port it to e.g. the *Android* platform could be identified. Presumably this is due to the design of the network, which prefers a stable network connection, a considerable amount of storage and possibly needs for firewall and port-forwarding configurations to function properly (see the *Freenet* help website[20] for more information on necessary configuration). So, for *Freenet* to work behind a router with NAT you have to manually forward ports on the router. *Freenet* can also forward the port automatically if Universal Plug and Play (UPnP) is enabled. Nevertheless, this is a problem for mobile devices, which are e.g. in a public wireless local area network (W-LAN) where the firewall cannot be configured. For more insights on *Freenet* see also [CSWH01] and [CMH+02].

[20] `https://freenetproject.org/help.html` accessed 09.2016

2.3.4 FastTrack

The proprietary *FastTrack* P2P network (see [JC10]) is based on the concept of *Gnutella*, but extends the network by an additional overlay. Each node that has enough storage, bandwidth and computing power can become a super-peer [SMR12]. Like Jin and Chan stated in [JC10], super-peers form a second overlay on the normal peers and maintain a meta-data index of available files stored on the normal peers, the network is therefore semi-centralized. When joining the network, a peer usually gets a predefined set of available super-peers from the client application and then uploads a list of its offered files to one super-peer. When searching for specific files, the query is propagated also to this super-peer like stated in [LCP+05]. Super-peers on the other hand operate in the same manner as peers in the *Gnutella* network. They query for the file by a broadcast search. The detailed protocol how super-peers communicate is not publicly known, because the source code of the system is encrypted. When the file is finally found in the meta-data list of a super-peer, a response is sent back to the sender with the address(es) of the peer(s) hosting the file and the actual data exchange happens via HTTP by a direct connection. Figure 2.4 shows the two-layered data upload and data lookup approach used in *FastTrack*. Here *Peer 1* requests a file *Object2* from its super-node and gets the address of *Peer 2* as response. Then it downloads *Object2* directly from *Peer 2*. Explained by Li in [Li08], the protocol implements the *UUHash* hash function so that users can download a file from multiple sources at the same time. This is possible because this hash function allows hashing of file parts, but therefore is also vulnerable to collision attacks, leading to *FastTrack* being vulnerable to pollution attacks, which was heavily used by the Recording Industry Association of America (RIAA) to distribute fake or corrupted files.

Liang et al. show in [LKR06] how *FastTrack* can circumvent firewalls and NATs. Newer client implementations use random port numbers to prevent static port blocking at the firewall. The randomly selected port is then published to the super-peer of the peer for others to retrieve. Furthermore, if a peer is behind a NAT router and a direct connection cannot be established, a requesting client instead contacts the super-peer, which in return contacts the destination peer to actively establish a connection with the requesting peer to send data. This concept is also used by other P2P protocols that include servers or higher level peers and is called connection reversal.

Obviously *FastTrack* suffers from comparable security issues like *Gnutella*, e.g. DDoS, because it also uses query flooding at super-peer level [LCP+05]. Also, fake content and malware can be distributed. Furthermore, popular *FastTrack* clients like *KaZaA* have been identified to contain spyware, like discussed in [BCJ04]. Liang et al. also discuss in [LNR06] how index poisoning can be done in *FastTrack*. Index poisoning means the insertion of fake records into the index of a central authority and thereby achieving a DoS. The paper also deals with how to harvest IP addresses and ports in the *FastTrack* network.

In the research no efforts to implement the *FastTrack* protocol on a current mobile device

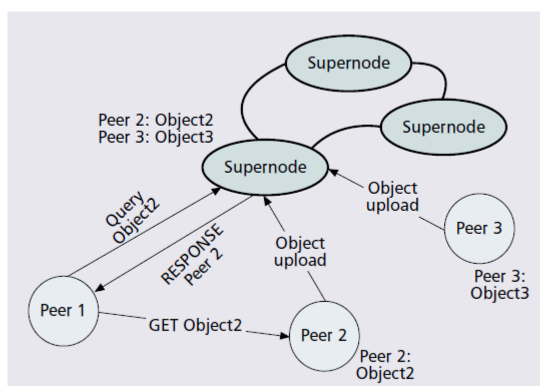


Figure 2.4: *FastTrack* two-layered P2P network [LCP+05]

could be identified. A reason therefore is probably that *FastTrack* is proprietary and many of the *FastTrack* clients are having issues because of illegal use for file-sharing or have even been shut down. The most popular application implementing the *FastTrack* protocol was *KaZaA*. The application was closely coupled with the protocol and developed by the same programmers. The last stable release was in 2006 and the *KaZaA* website[21] has been shut down. There exist further proprietary *FastTrack* clients that are not in further development or have been shut down due to copyright infringement like e.g. *Grokster*.

2.3.5 eDonkey

The *eDonkey* P2P protocol (see the protocol definition on the *jmule* website[22]) defines a centralized overlay network with *eDonkey* servers and client peers. Servers maintain lists of meta-data from files that client peers provide. At start-up, a client connects to one or more server and uploads the meta-data of its files providing to others. Usually *eDonkey* client software is shipped with some well known *eDonkey* servers to connect to, further servers can be added by getting their addresses from e.g. websites. The protocol differs from *Napster* by not using a single server or server farm. Instead dedicated servers are run by power users. Servers communicate with each other by exchanging server lists and their presence state via UDP. In contrast to *FastTrack's* super-peer structure, the servers do not exchange or cache file meta-data from other servers. Also, the server software is an independent application and is proprietary. After connecting to a server, the client peer can retrieve a list of further servers where it can also search for files. The actual search is done by a simple full text search on the file name and some additional file properties in the server meta-data list of files. The server then responds with a list of files that matched the query including the hash of the file content. After the user has chosen a file and initiated the download, the client application sends the hash of the file to the server and gets as response a list of addresses and ports of peers hosting the file

[21] `http://www.kazaa.com/` shut down

[22] `http://www.jmule.org/files/eDonkey-protocol-0.6.2.html#File%20info` accessed 10.2016

data. This happens repeatedly after some time to get new addresses and ports from peers that host the file. After that, a direct connection to some of the sources is opened and a simultaneous download of the file is started. Also here, the file gets split up in chunks so different parts can be downloaded from different sources.

Like already mentioned, the server software of the *eDonkey* protocol is proprietary. There have been some freeware implementations created by reverse engineering, but they have not been published as open-source. For peers behind a firewall, which cannot be contacted directly by another peer, a special mechanism exists. Like stated in [HB02], such peers receive a so-called *low-id*. When a peer wants to contact a *low-id* peer it has to send a request to the server to send a push notification to that peer over their existing server-peer connection. After that, the peer contacts the requesting peer and the data exchange can happen. Nevertheless, two *low-id* peers cannot receive data from each other, at least one of them must be reachable directly by design of the original *eDonkey* protocol. To be reachable from outside, port forwarding of the ports that an *eDonkey* client needs has to be done if the peer is behind a firewall. Therefore, usually the user has to have administrator permissions on the firewall or router.

Thommes and Coates describe in [TC06] how they could harvest IP addresses and ports of *eDonkey* users. Also, like the most other P2P file-sharing networks, it suffers from pollution and malware distribution. Also, because the structure of *eDonkey* is comparable with that of *FastTrack*, it is vulnerable to index poisoning which can similarly result in DoS.

There exist open-source multi-platform implementations of *eDonkey* clients like *JMule*[23] where the last release was in 2010. No ports of an *eDonkey* client to *Android* or *iOS* platform could be found, a reason could be that the protocol is basically designed for file-sharing, which is currently still not particularly established on mobile platforms because of mobile network limitations. That can be derived because there already exist e.g. some *Android* applications to manage an *eDonkey* client application that is running on a personal computer. One example for such an application is *MLAndroid*[24]. *JMule*, though, could be the most suitable client to try to port to e.g. the *Android* platform, because it is open-source and written in Java.

Because of increasing load on *eDonkey* servers, a successor protocol, *Overnet*, has been designed. It can operate completely without servers using the *Kademlia* distributed hash table (DHT) algorithm, for further explanations on *Kademlia* and DHTs see the work of Tillian [Til17]. No ongoing development on the protocol could be identified, server software is not open-source and there are no official statements on usage or versions. Research on clients supporting the *eDonkey* network showed that latest releases are years ago. You can read more about *eDonkey* in [HBMS04], [LCP+05] and [HKLF+06].

[23] `http://www.jmule.org/` accessed 09.2016

[24] `https://play.google.com/store/apps/details?id=com.mme.mlandroid` accessed 09.2016

2.3.6 BitTorrent

BitTorrent is a collaborative and centralized file-sharing protocol[25] that originated in 2001. The founder of the protocol, Bram Cohen, describes the concept of the protocol in a detailed way in [Coh03]. Content is distributed in the network by so-called *torrent* files with the file extension *.torrent* or *.tor*. These files are usually distributed on *torrent* file exchange platforms (usually websites) on the internet [LCP+05]. The torrent file contains the address of one or more trackers, which are responsible for the distribution of the data file or the bunch of related data files the peer wants to download. It also contains a list of checksums of segments of the data file(s). Now, when a peer wants to download data, it connects to the tracker and the tracker adds the peer to a list of collaborating peers, also called swarm, for the corresponding data file(s). The downloading peer now gets from the tracker a list of other peers, where it can download parts of the data file(s), usually in chunks of several kilobytes. Using this method, a peer can download parts of the file from different other peers of the swarm that already downloaded these parts and can check for integrity by using the checksums of the parts from the *torrent* file. When a part has been successfully downloaded and checked, the peer reports this to the tracker and now can also provide this part of the data file for other peers as a so-called seeder. In a nutshell this means that a peer acts as an uploader and downloader of the same data file simultaneously, like described by Saroliya et al. in [SMR12]. Figure 2.5 shows how different peers form a swarm around a tracker that is addressed by a *torrent* file. With this technique, the more peers already have downloaded parts of the file the more the distribution performance can be improved. Also, the protocol implements means to reward peers with good upload rates by providing them with good download rates. This principle of reciprocity has been driven forth by implementing functionality to ban peers from the network that only download data but do not contribute to the P2P principle by also providing the data to others. A metric to measure this was for example by providing bonus points to the peer for a specific time interval of uploads that then could be redeemed in download time. By design of the protocol, many disjoint *BitTorrent* networks may exist globally, by building swarms around many different trackers that provide specific files. There has been also an implementation of a DHT algorithm for the *BitTorrent* protocol, which then runs without any trackers. The function of the tracker in this case is done by the *BitTorrent* client application itself.

Because the *BitTorrent* specification is free to use, many open-source clients have been implemented for various operating systems. An example for an *Android* client is *aTorrent*[26]. For *iOS* it is not so easy to install a *BitTorrent* client, because *Apple* blocks them from their *AppStore*. Still, there are possibilities to download and install a *BitTorrent* client for *iOS* platform like *iTransmission*, downloadable from the *iemulators* website [27]. Also for the *Windows Phone* platform, a *BitTorrent* client (downloadable from

[25] `http://www.bittorrent.org/beps/bep_0003.html` accessed 09.2016

[26] `https://play.google.com/store/apps/details?id=com.mobilityflow.torrent` accessed 09.2016

[27] `http://iemulators.com/` accessed 09.2016

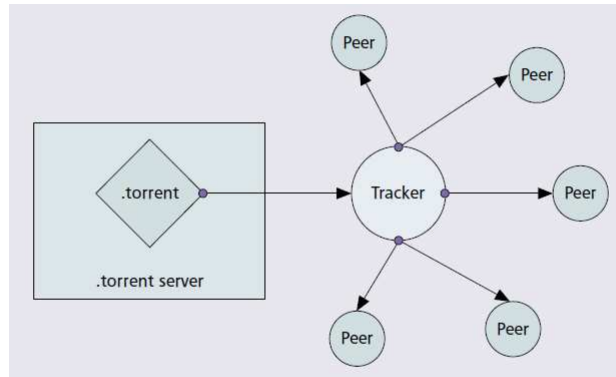


Figure 2.5: BitTorrent architecture with .torrent file, tracker and peers [LCP+05]

the *Microsoft Store*[28]) exists. Not only there exist many *BitTorrent* clients in ongoing development, also the open protocol itself is continuously enhanced. There exists an own website[29] where *BitTorrent Enhancement Proposals* to the protocol are managed, so the protocol is steadily extended.

In [KBM07] Konrath et al. show how a swarm can be attacked by malicious peers and hindered or taken out of service with modest amounts of resources, which can be seen as a DoS attack. Here, attacking peers lie about possession of data pieces and make them artificially rarer. Defrawy et al. show in [EDGM07] how *BitTorrent* can be used to launch DDoS attacks by using modified trackers. Santos et al. state in [SdCCGB10] how *BitTorrent* also suffers from heavy pollution and malware distribution.

Regarding firewalls and NAT bypassing, various *BitTorrent* clients implemented different techniques to accomplish that. For example *μTorrent*[30], like many others, suggests manual port forwarding or using UPnP or NAT port mapping protocol (NAT-PMP) for automatic port forwarding (see the *μTorrent* help center[31]). This client also implements the UDP based *μTP* protocol, explained in more detail in [TR11]. This enhancement to the *BitTorrent* protocol is trying to increase performance of data exchange by decreasing delay times. Compatibility to this protocol has also been added to other *BitTorrent* clients later. UDP hole punching is also supported in connection with *μTP* to bypass NATs (see the *BitTorrent* help center[32]).

[28] `https://www.microsoft.com/en-us/store/p/wptorrent/9wzdncrfj4cw` accessed 09.2016
[29] `http://www.bittorrent.org/beps/bep_0000.html` accessed 09.2016
[30] `http://www.utorrent.com/intl/en/` accessed 09.2016
[31] `http://help.utorrent.com/customer/portal/articles/1753715` accessed 09.2016
[32] `http://help.bittorrent.com/customer/portal/articles/163493-client-features` accessed 09.2016

2.3.7 Summary on unstructured P2P networks

In this section a compact summary on the analyzed P2P networks is done in form of Table 2.1. Additional to the main characteristics of the respective network, like e.g. structure and search algorithms, also features essential for later evaluation of usability as base technology for the MPM framework like e.g. publication, development state, NAT traversal and existing implementations on current mobile devices are compared.

	Napster	**Gnutella**	**Freenet**	**FastTrack**	**eDonkey**	**BitTorrent**
Centralization	centralized with servers	fully decentralized	fully decentralized	centralized with super-peers	centralized with servers	centralized with trackers
Structure	static, provider-hosted servers	flat and pure ad-hoc P2P network	semi-structured, flat and pure P2P network	two-layered with user-hosted super-peers	customized, user-hosted servers	dynamic, user-hosted trackers
Bootstrapping	predefined server list by client software, manual adding of servers possible	predefined set of very likely available *Gnutella* hosts by client software	predefined search by application or manual connect to trusted users in *Darknet* mode	predefined set of likely available super-peers by client application	predefined set of servers by client application, manual adding possible	retrieve tracker address from *.torrent* file
Search and retrieve data	text string search on server, direct download	query flooding and direct download	hashes of files with routing to best neighbour, inverse routing to get data	index on super-peer, super-peers communicate to locate file	search on server, retrieve further servers from server, direct download	get peers from tracker for direct download

	Napster	Gnutella	Freenet	FastTrack	eDonkey	BitTorrent
Publication	originally proprietary, open-source implementations exist	open-source	open-source	proprietary	server software is proprietary, open-source client software exists	open protocol, many open-source implementations exist
Development	no server development since years, some clients seem to be in progressive state	no development on protocol since years, some clients in ongoing development	in ongoing development	no ongoing development on protocol or client software found	no official development on reverse engineered servers, clients no release since years	ongoing extension of protocol and development of client software
Vulnerabilities	DoS attacks, censorship, technical failure, IP harvesting, polluted media files	DDoS attacks, malware / pollution distribution, IP harvesting	MitM attacks, malware / pollution distribution	DDoS attacks, malware / pollution distribution, spyware in clients, IP harvesting	DoS attacks, malware / pollution distribution, IP harvesting	DoS attacks, initiate DDoS attacks, malware / pollution distribution

	Napster	**Gnutella**	**Freenet**	**FastTrack**	**eDonkey**	**BitTorrent**
NAT traversal	connection reversal using server	inverse route push requests	manually by port forwarding or UPnP	connection reversal using super-peer	port forwarding or connection reversal using server	manually by port forwarding or UPnP/NAT-PMP, UDP hole punching
Ported to mobile	none found	existing for *Gnutella2* on *Android*	none found	none found	none found	existing for common platforms

Table 2.1: Comparison of popular unstructured P2P protocols, state at 09.2016

2.4 General purpose P2P protocols

From the former section it can be derived that use-cases for the existing unstructured P2P overlay networks are mainly file-sharing and content distribution, although e.g. pure P2P communication has already been implemented using the described protocols or could be realized. Still, there also exist protocols and frameworks that are more general and offer functionality for building P2P based applications. Besides the communication handling, some offer additional P2P services like peer discovery, forming of peer groups, and so on. After some research Peter Tillian and me selected the four most popular, widely-used and probably applicable representatives. Juxtapose (JXTA) and session initiation protocol (SIP) are discussed in this thesis, Java agent development framework (JADE) and the extensible messaging and presence protocol (XMPP) are addressed in the thesis of Peter [Til17].

2.4.1 Juxtapose - A general purpose P2P framework

The open-source P2P framework JXTA (see the protocol specification[33]) has gained some popularity in P2P approaches and research. The development was started in 2001 by Sun Microsystems Inc. and it was set up as a logical overlay of several extensible markup language (XML) based protocols over IP and non-IP networks, like Harjula et al. explain in [HYAK$^+$04]. The network is more generic in comparison to e.g. the *Gnutella* or *BitTorrent* P2P approaches because it is not dedicated to a specific purpose like in that case file-sharing.

Characteristics

The *JXTA* protocol consists of six XML based protocols. The *Peer Resolver Protocol* enables peers to send messages to one or more other peers and also receive one or multiple answers. The *Endpoint Routing Protocol* is responsible for the routing mechanism to deliver messages to a destination peer. The *Peer Discovery Protocol* is used to discover the services and resources that a peer advertises. The *Rendezvous Protocol* is responsible for propagating messages in a group of peers. The *Peer Information Protocol* is used to query state information of a peer and finally the *Pipe Binding Protocol* is used to establish connections, in *JXTA* named pipes, between two peers. These pipes can be seen as an abstraction layer over the *Endpoint Routing Protocol*, hiding the logic of routing the messages. The *Pipe Binding Protocol* can use a variety of underlying transport protocols such as HTTP, TCP/IP or bluetooth. All messages exchanged over such pipes are XML documents that can also contain binary codes [BX11].

The structure of the framework is comparable to *FastTrack* or *Gnutella2*, forming a two-layered network with so-called *rendezvous peers* and *relay peers*, which are super-peers, and normal *edge peers*, like stated in [LA10]. *Rendezvous peers* maintain lists of their *edge peers* and their offered resources and services. In contrast to *FastTrack* or *eDonkey*, *JXTA*

[33] `https://tools.ietf.org/html/draft-duigou-jxta-protocols-02` accessed 10.2016

uses a DHT technique for distribution of these indices to other *rendezvous peers*. Now, when an *edge peer* searches for a resource or service, a query is sent to the *rendezvous peer* and also a multicast to all the peers in the same subnet. If the *rendezvous peer* finds the resource in its local cache, it notifies the peer(s) hosting the resource and the peers will respond directly to the requester peer by establishing a pipe. If the resource cannot be found at the local rendezvous peer's index, the same DHT function is used to locate the *rendezvous peer* that stores the index for the resource. Then a pipe between the hosting *edge peer* and the requesting *edge peer* is established using the mentioned *Endpoint Routing Protocol*. NAT and firewall traversal is achieved by *relay peers*. So, whenever a network consists of more than one subnets with NAT, at least one *relay peer* is needed to connect *edge peers* of those networks. A *rendezvous peers* can at the same time also serve as a *relay peer*.

Figure 2.6 shows the trace of a resource lookup and data transfer within the JXTA routing model. *Edge peer A* initiates a query and sends it to its *rendezvous peer RP1*. Since *RP1* does not hold the index for the queried resource, it forwards the query to its own *rendezvous peer RP3*. *RP3* has the information that some of *rendezvous peer RP4s edge peers* hosts the resource and forwards the query to *RP4*, which has the corresponding index and forwards it to *edge peer B*. Because *B* is in a different subnet behind firewall and NAT, it establishes a pipe with *A* using *RP4* as a relay. As can easily be seen in this example, the DHT mechanism called shared resource distributed index (SRDI) used by these *rendezvous peers* can be hierarchical, allowing a *rendezvous peer* having its own *rendezvous peer* as super-peer [HYAK+04]. This model differs from other hybrid P2P models like *FastTrack* having a flat super-peer layer. Also, the network is semi-structured using a loosely coupled DHT technique at the super-peer level with a local routing table like implemented also by *Freenet*.

JXTA supports forming of peer groups providing a shared environment for participating peers, like Barolli et al. state in [BX11]. Here, own policies for membership can be applied, where peers can belong to more than one group at the same time.

Security

Arnedo-Moreno and Herrea-Joancomartí [AMHJ09] have done a fine-grained survey on security in *JXTA* applications. Several security mechanisms have been integrated in the framework, e.g. signed advertisements of resources and services on *edge peers* to avoid spoofing, MitM attacks or replay attacks. Moreover, they implemented an own version of transport layer security (TLS) on the transport layer to avoid eavesdropping of messages. Furthermore, it implements crypto-based JXTA transfer (CBJX), proposed by Bailly in [Bai02]. CBJX provides a lightweight and secure message source verification by enriching messages with the sending peer's digital signature ensuring data integrity and authentication. By that means, spoofing, MitM attacks and replay attacks can be prevented for the message exchange part of the framework. Nevertheless, this only is valid for unicast messages, multicast messages for groups of peers are not reliably and not secure, like Bailly also states in [Bai02]. [AMHJ09] also reveals that the core *JXTA*

protocol has several vulnerabilities in the basic peer operations. By traffic analysis it is possible to identify important peers, no masquerading mechanism exists. There exists no encryption mechanism for advertisements of resources and services by *edge peers*, the advertisements are transmitted in plain text and last but not least authentication is not enforced in the peer group joining mechanism. It is possible for malicious peers to join any group.

Implementations

JXSE[34] is the official Java implementation of the JXTA framework. The last stable release is currently version 2.7 from March 2011. According to a news entry on the official version control system (VCS) of Sun Microsystems Inc.[35] , Oracle, which acquired Sun Microsystems Inc., announced withdrawal from the *JXTA* projects including *JXSE*. Since Oracle refused to transfer the JXTA trade name and project to the Apache Software Foundation, like the community wanted to, there has been a voting for a new name: *Chaupal*[36]. According to the official *Chaupal* wiki[37], the *JXSE* code is still under the *JXTA* license which cannot be changed, extensions to *JXSE* are governed under the Apache License 2.0 and published in an own repository, where currently the last commit was in July of 2015.

There also exists a C/C++/C# implementation of the *JXTA* framework, namely *jxta-c*[38], but with the last release in 2007 it seems quite abandoned. This extends also to the Java Platform Micro Edition (J2ME) implementation called *JXME*[39], with the last contributions in 2009. These implementations might be insignificant because they are deprecated and seem in no further development. Also, according to the statistics on the *Netmarketshare* website[40], the market share of J2ME in mobile and tablet operating systems is below 1.5 percent and decreasing. *JXME* has been designed by the *JXTA* community as a light version of *JXTA*, because the protocol is quite heavyweight for thin peers [HYAK+04].

There has been a porting of the *JXME* implementation to the *Android* platform, named *peerdroid*, which is downloadable at the *Google* code archive[41]. The last commit to the source code was in July of 2010. Filbert shows in [Fil10] how he implemented a multi-purpose P2P chat application for the *Android* platform using *peerdroid*. In the *peerdroid* implementation a peer represents only an *edge peer*. A rendezvous peer must be implemented using *JXSE* and therefore be run on a desktop computer.

[34] https://java.net/projects/jxta-jxse/sources/svn/show accessed 10.2016
[35] https://kenai.com/projects/jxse/pages/LatestNews accessed 10.2016
[36] https://github.com/chaupal accessed 10.2016
[37] http://chaupal.github.io/wiki/contents.html accessed 10.2016
[38] https://java.net/projects/jxta-c/sources/svn/show accessed 10.2016
[39] https://java.net/projects/jxta-jxme/sources/svn/show accessed 10.2016
[40] https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1 evaluated 06.10.2016
[41] https://code.google.com/archive/p/peerdroid/ accessed 10.2016

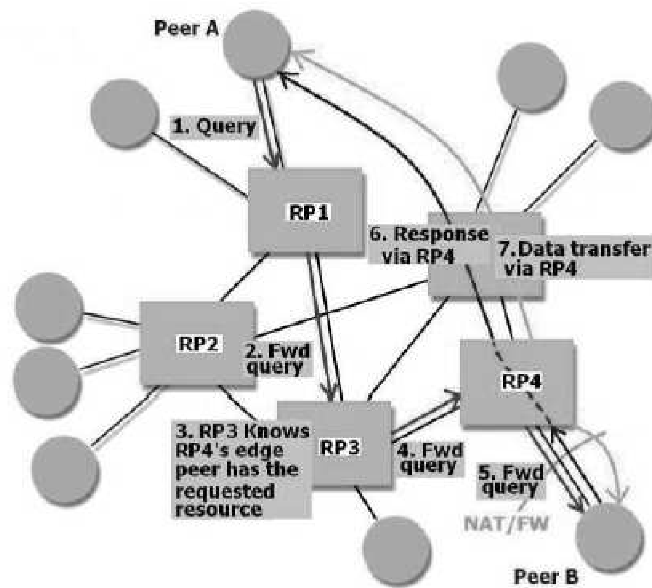


Figure 2.6: JXTA resource lookup [HYAK+04]

2.4.2 P2P communication using Session Initiation Protocol

SIP (see the RFC[42]), which is widely used as a communication protocol for multimedia data transmission, including internet telephony with audio and video calls, but also IM over IP networks, is presented in this section. The original version of this text-based, open protocol is client-server based with different types of servers that is discussed shortly in this section, but there are efforts of using the protocol in a P2P manner, defining it as peer-to-peer session initiation protocol (P2PSIP), which is developed by the Internet Engineering Task Force (IETF) in an own work group and is based on resource location and discovery (RELOAD) (see the RFC[43]) as the P2P overlay network protocol.

Characteristics

The SIP is used to establish sessions between two or more participants. It can traverse NATs and firewalls and can address users independently of their network IP address. The underlying transport protocol can be transmission control protocol (TCP) or UDP. For endpoint resolution it uses different, logical server roles, like Wierzbicki et al. explain in [WDŻR10]. The *proxy server* role is responsible for the address resolution, it also works as gateway to other networks or domains. Optionally, it also stores some information for clients like failed requests to establish a session by others (e.g. missed SIP calls) or reports about usage. The *registrar server* role contains the definite information for SIP address resolution, the IP address of the client. SIP clients must register to this server

[42] `https://www.ietf.org/rfc/rfc3261.txt` accessed 10.2016
[43] `https://tools.ietf.org/html/rfc6940.html` accessed 10.2016

first to be able to establish sessions. The protocol resembles the HTTP protocol, but in contrast it is possible for client users to act as a SIP sender that sends requests and also as server that sends responses at the same time. The users are identified by uniform resource identifiers (URIs) in the form of *sip:user@domain* [BAD06]. The *redirect server* role allows proxy servers to redirect session establishment invitations also to clients on an external domain. The *location server* role is responsible for translating the client SIP URIs to the possible location of a client. It therefore maintains a database of SIP-address/IP-address mappings. Figure 2.7 shows how a session is established between two SIP clients within different domains. The numbers 1 to 14 show the message flow. The initiating user agent contacts its *proxy server*, which furthermore contacts the *redirect server* to redirect the traffic to the *proxy server* of the destination client's domain. Then the *location server* is used to find the *proxy server* that is used to traverse the NAT of the destination client. When the destination client is available, the session between the clients can be established. In a physical layout, different server roles can also be integrated on one host.

SIP, which supports endpoint locating and also establishment, termination and modification (e.g. adding further participants) of sessions, is decoupled of the protocol used for the actual media transport. Often the real-time transport protocol (RTP) is used for this purpose. It is a streaming protocol based on UDP that supports streaming of multimedia datastreams like audio, video but also text. SIP has gained widespread acknowledgement and deployment in services like VoIP, IM, collaboration and for push-to-talk services [BAD06].

P2PSIP

The aim of P2PSIP is to provide the functionality of SIP without the use of a server infrastructure. The conceptual model of the system is that peers are coupled with SIP entities like *proxy* or *redirect servers* and operate in a P2P overlay network. When a peer wants to join the network, it must locate a P2PSIP peer that has already joined the network by using a cached list, multicast or public bootstrap nodes. After that, it has to authenticate itself and register itself in the overlay directly or contact a peer that serves as a proxy if that is not possible. The role of the *registrar server* in this concept is distributed in the overlay network, usually using a structured DHT-based P2P approach. When a peer discovers another peer over the network, initiation of sessions use the standard SIP functionality independent of the P2PSIP protocol, but possibly using proxy or gateway peers (e.g. for NAT traversal).

The IETF is developing a generic P2P protocol that can be used to achieve P2PSIP functionality and named it RELOAD. It represents an overlay network with a pluggable topology, that means that the overlay topology algorithm can be implemented as a plugin in the protocol, allowing different structured overlay implementations like *Chord* or *Kademlia* to be used with the protocol, having *Chord* as mandatory standard implementation. For more information on those two protocols see [Til17]. The plugin offers abstract messages to e.g. join and leave the overlay network or request the responsible

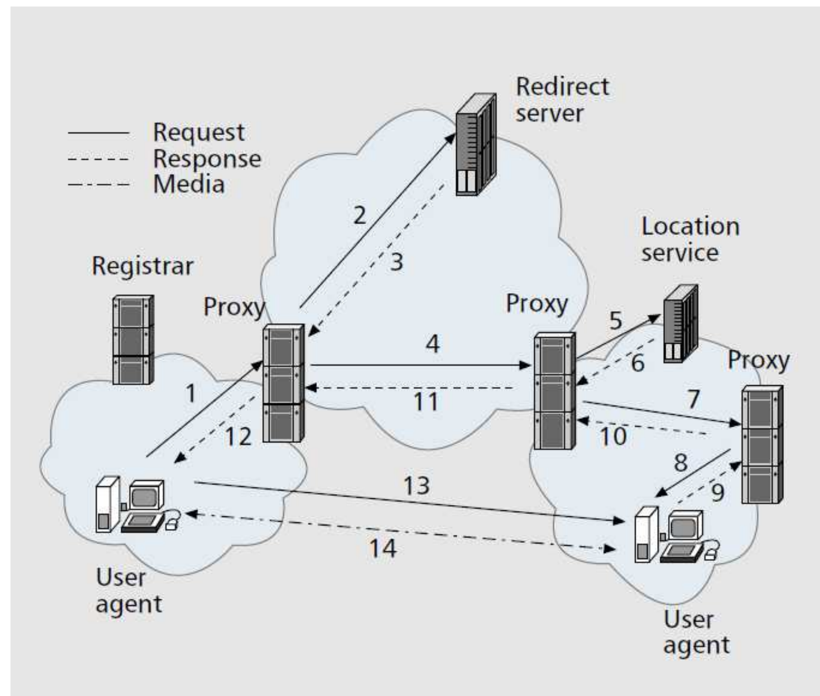


Figure 2.7: Session initiation over SIP [BAD06]

peer to route a message directly to a given destination. Like explained in detail by Roly in [Rol09], the protocol also provides a service to store and retrieve information in the structured overlay network using the topology plug-in. In the protocol, each participant node has a node ID. To replace the SIP *registrar server*, the plugin's storage service can be used to publish the mapping from SIP address to the node ID in the overlay network [TSIS12]. Then, when a client has identified the node ID of the user it wishes to contact, it uses the message routing system of RELOAD to set up a direct connection and exchange SIP messages. For this concept to work, the clients must have a RELOAD implementation running on their hosts. If this is not possible, they may also use a pure SIP client to connect to one peer that has the RELOAD implementation running, acting as a proxy/registrar and making the overlay transparent to the client. For further information on structured overlay networks and their possible usability for the MPM framework see the work of Peter Tillian [Til17].

The P2PSIP specification documents are still on standards-track, but the concept seems emerging and promising to become an internet standard any time soon, since the documents of the IETF seem to be work in progress (for more information see the P2PSIP status pages on the IETF website[44]).

[44]`https://tools.ietf.org/wg/p2psip/` accessed 10.2016

Security

P2PSIP in conjunction with RELOAD is designed for very large networks over the internet where security should be a big factor in system design. Therefore, connections between peers use TLS and datagram transport layer security (DTLS). Also, each message and object in the structured overlay network is signed by a certificate. The certificate can be found in the overlay configuration file, which can be downloaded from a secured server [Rol09].

Touceda et al. survey in [TSIS12] in detail the possible attacks and defenses on P2PSIP networks. These include e.g. possible attacks on the access control like ID mapping attacks, where an attacker in the overlay is responsible for storing the ID mapping information for other users and therefore for example could deny any attempt from a user to contact other users (DoS). Even if the responsibilities for storing the mapping information gets distributed in the network using the DHT or if users get assigned new IDs periodically, still attackers can use sybil attacks on the system. A sybil attack means that attackers try to control a lot of nodes in a P2P network to have an influence on decisions. In that case this approach could be used to take over a big portion of the system and disrupt resolution of IDs to addresses. Also, for a new node, when bootstrapping the system (joining for the first time), Touceda analyses possible attacks like the fake bootstrapping attack, where a new peer's first contact is a malicious peer, giving it a fake initial view on the system. For access control and bootstrapping attacks, the author mentions a central authority for identity and certificate management as a possible solution. For secure bootstrapping, even an external system mechanism like used by *BitTorrent* with the *.torrent* files is conceivable. The author also emphasizes approaches in the RELOAD protocol that prevent attacks on the overlay network like MitM, eavesdropping and message replay by using encryption, digital signatures on messages and also message counters. All communications shall use TLS or DTLS. This also extends to the media transmission protocol if used. For the storage service of the overlay network, the author recommends again the use of a centralized access control model and, to increase availability, the replication of user's contact information. Moreover, it is suggested to implement a functionality to create buddy lists. This would make it possible to create a sort of optional white-listing to prevent DoS and spamming. The conclusion of the work is that establishing security is not easy in a P2P system and usually more difficult and resource consuming than in a centralized approach. An extensive analysis of the developed system is important to be able to balance countermeasures to security threats and performance.

Implementations

The *Android* platform has integrated an SIP client implementation since version 2.3. A description of the application programming interface (API) can be found at the official

Android developer website[45]. For *iOS*, there is no integrated API but there exist open-source implementations of the protocol that can be compiled to work with *iOS* like *PJSIP* (see the official website [46]). *PJSIP* is also usable with *Android*. The variety of SIP clients is high, but in the client-server implementation also at least one dedicated SIP server is needed, or even more servers with different server roles. There exist several open-source implementations of SIP servers, e.g. *OpenSIPS* (see the official website [47]), for different platforms.

Some proof of concept implementations of the P2PSIP approach, like the *Columbia P2PP Project* (see the project website [48]), can be found. It is an open-source implementation of a VoIP and IM system following the P2PSIP principle, using an own implementation of *Kademlia* DHT as overlay topology. It has been implemented in *C++* and is available for *Windows* and *Linux* operating systems. Cohrs [Coh08] shows in his work another proof of concept implementation using the *Ruby* programming language. Roly [Rol09] did a partial implementation of the RELOAD protocol and showed some obstacles of the implementation in *Java*. Because SIP can be used with current mobile devices, running e.g. *Android*, the P2PSIP client protocol could be implemented on these devices to connect to fully functional peers as clients. No usable full peer implementations for current mobile platforms of a complete P2PSIP protocol together with an overlay plug-in (like in RELOAD) could be found.

2.5 Coordination frameworks and models

This section deals with coordination frameworks. Coordination in this context is understood as the controlling and directing of data flow and corresponding computation of data in a network of self-contained nodes. Nodes here mean physical hosts and not virtual nodes in an intra-process model. When conducting the research on related work, some existing approaches for coordination in distributed systems could be identified. There exist some modelling tools, which include modelling of coordination parts of a system, but the focus is on higher level models or even implemented frameworks, which could be applicable for coordination of nodes in a highly distributed P2P network.

2.5.1 DataSpaces

DataSpaces [DPK12] is one approach to be mentioned here. It is a distributed framework and consists of dynamic sets of nodes. The main purpose of the framework is to support coordination in distributed systems, where the nodes produce massive amounts of data (like large simulations on super-computers). The lookup of nodes is implemented by using a DHT. The communication layer is modular. Existing communication frameworks

[45] `https://developer.android.com/guide/topics/connectivity/sip.html` accessed 10.2016
[46] `http://www.pjsip.org/` accessed 10.2016
[47] `http://www.opensips.org/` accessed 10.2016
[48] `http://www1.cs.columbia.edu/~salman/peer/` accessed 10.2016

for distributed shared data like message passing or shared memory can be used. The communication layer together with the DHT form a virtual shared space at a higher abstraction level. In this space, data is specified as key-value pairs and stored at a virtual address. The actual addressing scheme of the DHT is specific to the application. The *DataSpaces* query engine supports and optimizes complex queries on the virtual shared space and also supports notification of data availability. Coordination between application components is realized using a model similar to *TupleSpaces* [DSMPR03], optimized for grid computing. In contrast to the centralized, globally shared space of *TupleSpaces*, *DataSpaces* shared space is decentralized at the data store layer and data can be inserted, retrieved and filtered in a decoupled and asynchronous way.

2.5.2 TuCSoN

TuCSoN [OZ99] is another example of a coordination concept that is based on tuple spaces, furthermore it is agent-based. The basis are distributed tuple spaces called *tuple centers*, which are constructed with enhanced *Linda* spaces [Gel85]. *Tuple centers* are distributed through the network and separated in specific domains. The meta-data of these domains is populated by a *tuple center* on a gateway node and the application data is hosted on connected *tuple center* nodes called *places*. *Gateway tuple centers* can keep track of which *tuple centers* reside in the neighbourhood. *Tuple centers* can be addressed in the internet uniquely by the identifier of the hosting node and the identifier of the *tuple center*. The actual coordination policies are applied depending on the currently contained tuples in the tuple spaces (the state of the space). These tuple spaces additionally to the abstraction also support modularity, information hiding and security. State transitions happen as reactions to communication events from actors in the system, therefore a reaction specification language is used, which links communication events from actors to reactions and possible state transitions of the *tuple centers*.

2.5.3 DTuples

Another agent-based approach that uses the concept of tuple spaces in a distributed manner is *DTuples* [JXJY06]. Like *DataSpaces*, it uses an underlying DHT as distributed storage. In the prototype implementation *FreePastry* is used as underlying P2P overlay network. Agents communicate through tuples in shared spaces. Sub-spaces (with own name or subject) can be created and bound to one or more agents so that private spaces with own DHT (and subject) can be hierarchically defined. Additionally, there exists a global shared space. Coordination is done by template matching on a string identifier and on some further tuple parameters. In the matching procedure also the information hiding by the sub-spaces (with own subjects and contributors) is considered.

2.5.4 Comet

A similar approach to *DTuples* is proposed by Li and Parashar in [LP05] and called *Comet*. It is also *Linda* [Gel85] based and uses a DHT. The prototype implementation

is based on the JXTA P2P framework and a DHT mechanism that is aware of context localities. That means that data is stored considering proximity, e.g. tuples that originate from the same geographical region. These context localities could be important for mobile applications. In *Comet*, so-called *transient spaces* can be defined, which support scope constraints (e.g. to be hosted in the same geographical region) with membership and authentication mechanisms for peers participating in a concrete space. Applications can switch at runtime between such transient spaces and a globally accessible space. Coordination is similar to *DTuples* with the additional benefit of content locality in comparison to the subject spaces of *DTuples*.

2.5.5 CArtAgO

Another approach, which is once again agent-based and possibly also makes use of tuple spaces, is presented by Ricci et al. in [RPV11]. Nevertheless, it differs from the presented approaches. It is based on a concept that is called *agents and artifacts* (*A&A*) and is implemented in a computational framework named *CArtAgO*. Artifacts can be used by agents and offer operations and observable properties. Events might be triggered by actions on artifacts and might be perceived by other agents. For coordination between agents so-called *coordination artifacts* are used in the environment. In the work it is proposed that *Linda*-like tuple spaces might be integrated using their space-operations for coordination artifacts. All artifacts can be dynamically instantiated and expose resources and tools to agents. The model is independent of the used agent programming platform. The authors show that the concepts of shared data objects, shared resources and also communication are implementable by using artifacts.

2.5.6 The Peer Model

Kühn, Craß, Joskowicz, Marek and Scheller proposed the PM in [KCJ+13] as a design tool for parallel and distributed applications. The model assumes an underlying space-based middleware. On top of that, in the model peers act as self-contained components, which interact by exchanging so-called entries that contain application and coordination information. Each peer stores received entries and outgoing entries in local containers (e.g. implemented as *Linda*-like spaces) and is globally addressable. The model, in contrast to the *agents and artifacts* approach from above, strictly separates application from coordination logic. In comparison to other coordination modeling tools, like *coloured petri nets* [JKW07] or *Reo* [Arb04], it is more high-level as it takes assumptions on the domain. This results in higher scalability and robustness of the model. At the moment, there is one finished C# [Rau14] implementation and some ongoing implementations of the model. More details on the PM are provided in Chapter 4.

CHAPTER 3

Requirements and selection of background technology

In this chapter the requirements on the proposed framework are defined. They are based on the problem statement and the aim of the work from the introductive chapter of this thesis and are also influenced by some insights obtained during the background technology research. In the second section of this chapter the evaluation and the selection of the researched background technology to be used in the proposed framework is done.

3.1 Requirements

Besides the functional requirements also the non-functional requirements (quality attributes) for the framework are included. All requirements have been stated in accordance with Peter Tillian (see the requirements chapter in [Til17]) and are listed in both works. Nevertheless, details on specific requirements that are regarding focus points of the works are referenced.

3.1.1 Functional requirements (FRs)

The functional requirements describe the functionality that the framework offers for developers and end-users. They define how the framework shall operate and which functions and tasks the framework shall provide.

FR1: Coordination

The framework shall abstract coordination logic of an application and offer an API to coordinate the data-flow and the execution of business logic in the system. Further details on the requirement can be found in [Til17].

FR2: Running in background

The framework shall offer an *Android* service that can run the MPM runtime in the background. That means that no graphical user interface has to be opened on the *Android* device. Because this is also in the scope of Peter's work see [Til17] for further requirement details.

FR3: Autonomous startup

Besides a manual start by a user an autonomous startup of the MPM runtime shall be possible. This shall be done when specific events, like an incoming message, occur at the device. The reason for this requirement is to save resources on the mobile device by making it possible to shut down the application and still react to incoming messages and by not maintaining a constant connection to the MPM network.

FR4: Decoupling from application

An application built on top of the framework should be decoupled from it. The implementation of the framework should be importable as an independent module using a build-tool or directly as a library. The network operations and system calls happening inside the framework logic shall not block or disrupt the outside application code. Nevertheless, there shall be possibilities for the developer to react to occurring exceptions or important events in the framework.

FR5: Connectivity with local and mobile carrier networks

Applications built on top of the framework shall be able to operate seamlessly in mobile W-LANs as well as networks of mobile carriers. Connection handover to another network shall be handled automatically and transparently by the framework. On loss of connection, re-establishment shall happen as soon as possible. No configuration on network firewalls or on a NAT box shall be necessary for the communication layer of the framework to work properly, as this is in most cases not possible in public wireless networks. Also, if a network is used where network configuration by the end-user is principally possible, it might be a hard task for an average home user without specific IT knowledge to do the needed configuration.

3.1.2 Non-functional requirements (NFRs)

The non-functional requirements describe how the framework shall operate in terms of quality attributes. In contrast to the functional requirements, which describe what the framework shall do, the non-functional requirements specify criteria that can be used to judge the operation of the framework.

NFR1: Licensing

The specification as well as the code of the reference implementation of the framework shall be published under a copyleft license and as open source software, making it unrestrictedly usable by the P2P community as long as no copyright restrictions are added on any changes or further developments on the framework. This requirement implies that all used external code, e.g. libraries, has to be under a compliant license too. The source code and documentation of the framework shall be publicly downloadable by everyone.

NFR2: Scalability

As mentioned in Section 2.2, a P2P network might not only consist of pure P2P components, which are scalable by design, but also of centralized components like super-peers or servers. There should be possibilities to scale such centralized components with acceptable effort, if they are needed in the system design.

NFR3: Security

As peers using the framework should be able to communicate over the internet, appropriate security measures shall be provided by the framework. In the communication layer possibilities for encrypting the network traffic shall be available as a countermeasure to eavesdropping. Signatures on encrypted messages shall prevent manipulation of data. Together with some kind of identity provider for peers, this shall also avoid spoofing. There shall be means to tackle spamming by blocking specific other peers or only allowing communication with specific peers, preventing also DoS to some amount. Optional end-to-end encryption shall offer additional security and privacy to end-users of applications built on the framework. If possible, functionality shall be offered by the framework to perform some kind of communication partner verification to circumvent false identity and MitM attacks.

NFR4: Simple API

The API of the framework shall be as simple and intuitive as possible, designed by exposing exchangeable, reusable and documented interfaces to the developer, hiding the inner logic in a black-box. The API shall also offer methods to propagate important events happening in an application onwards into the framework logic. Events that might happen in an outside application or operating system that are important for the framework are e.g. connectivity events like establishment of connection to a LAN or mobile network.

NFR5: Debugging and documentation

A developer shall be able to debug an application that is based on the framework, therefore the source code of the application shall be publicly available to be retrieved by the debugger of the integrated development environment (IDE). Moreover, there shall be

an option for a developer to configure a human readable form of serialization for network messages. Public interfaces shall be documented sufficiently in the source code and test cases shall be delivered with the source code as additional documentation and test basis for further extensions and changes to the framework.

NFR6: Exchangeability of components

The framework shall be designed so that important components are exchangeable by implementing interfaces. Possibilities to configure the used implementations shall be provided where appropriate. An example for such a component is the communication layer of the framework, which implements the used communication protocol and possibly also the identity management. The serialization component is another example. At least one implementation of all important components shall be delivered by the first version of the frameworks reference implementation. Decisions for the choice of the implementations shall be reasonable and described in this work.

NFR7: Design to work with a modeler

The API and configuration of the framework shall be designed in a way such that a model of a system based on the framework could be defined by a corresponding modeler. The scaffold of the application code shall be generated out of that model. The developer should only need to implement the application code, but cooperation or communication related logic shall be generated where possible. The implementation of the mentioned modeler is not within the scope of this work.

NFR8: Operability on popular mobile platforms

The framework shall be designed to be implementable on popular mobile platforms. Therefore, a reference implementation of the framework on *Android* shall be delivered as a part of this thesis. Operability on other mobile platforms shall be analyzed and considered.

NFR9: Benefit in comparison to own implementation

A developer shall have a significant benefit when implementing a mobile application for an appropriate use-case using the framework in comparison to implementing all functionality offered by the framework. This shall be achieved by abstracting the communication and coordination logic needed by a mobile application so that the developer can mainly concentrate on application logic and configuration of the framework. Appropriate security countermeasures shall also be abstracted by the framework and be configurable for the developer. Altogether, implementation of the application shall be faster and more efficient.

NFR10: Resource-efficient implementation

The MPM framework shall be optimized to save resources on a mobile device (storage, processing power and network bandwidth). No permanent connection to the MPM network shall be needed and the mobile operating system shall be able to suspend or close the MPM application when resources are needed. When this happens the application shall still be reactive to incoming messages, no data shall be lost and no resources shall be consumed in that state.

NFR11: Reliability

Whenever the MPM runtime is stopped (manually, by a failure or outside event) the data and state of the MPM runtime shall not be lost, shall be reconstructable and the processing shall correctly continue after a restart. For more details see the requirement in [Til17].

3.2 Evaluation and selection of background technology

The following section is about evaluation of the presented background technologies against the formulated requirements on the framework to be implemented. Based on the evaluation, useful technologies shall be selected to serve as basis for the design and prototype implementation of the framework. The process is done in correspondence with the work of Peter Tillian [Til17].

3.2.1 Selection of underlying P2P network technology

In Section 2.3 some important P2P overlay networks have been presented and analyzed, considering multiple factors. Meanwhile, Peter has analyzed prominent representatives of structured overlay networks. Furthermore, in this work JXTA has been presented as a general P2P programming framework. The possibilities of using SIP in the context of P2P networks have been presented in Section 2.4.2. Moreover, in [Til17] two more important, probably useful, technologies, namely JADE and XMPP have been discussed. JADE is, like JXTA, a general P2P framework, which in contrast is agent based. XMPP is a communication protocol that enables real-time P2P communication with authenticated partners. It uses an XMPP server for relaying messages and as an identity provider.

In the following sections, found background technologies from this thesis are evaluated against the proposed requirements on the framework. Peter evaluated the systems in his scope in [Til17].

Napster

The assessment begins with *Napster*. The network concentrates only on media file-sharing and also no open-source implementations of *Napster* servers that are in further development could be found. Also, no implementation of the protocol on a mobile

platform could be identified. Moreover, the centralized architecture with static servers makes the network highly vulnerable and not inherently scaleable. Therefore, *Napster* is excluded from the possible candidates for selection.

Gnutella

For the *Gnutella* protocol there exist open-source implementations and for *Gnutella2* there even exists one implementation for the *Android* platform. Also, scalability is given because of the pure P2P approach. Nevertheless, found implementations for *Gnutella* are not in further development and hard to port to current mobile platforms. The *Gnutella2* implementation for *Android* seems also to be abandoned by developers and not applicable on all devices. Security is hardly a topic in the network, e.g. malicious code distribution and eavesdropping are easily possible. Also, the protocol is trimmed on the purpose of file-sharing and not on general data exchange. Moreover, the query flooding approach of the protocol is not very resource efficient. Therefore, *Gnutella* and its successor protocol are excluded from further considerations.

Freenet

Freenet in contrast to *Gnutella* is highly focused on security and the *Darknet* mode even offers a concept of trusted peers. There exist open-source implementations, which are also still in development and could probably be ported to current mobile platforms with reasonable effort. The network is by design not only focused on file-sharing. Still, the main use-case is content distribution, where peers do not decide which content they store. This is not the intended use-case for the proposed framework. Additionally, participants have to contribute large amounts of local storage. The protocol needs firewall or router configurations for clients to work properly. Although the network might be a very interesting choice for content distribution over a static and stable connection and when the network and firewall can be configured, it can be excluded as possible background technology for the framework implementation.

FastTrack

The major issue with *FastTrack* is, that the protocol is proprietary. This also applies to the client and super-node implementations. Also, they are not in further development or even have been shut down because of legal issues with file-sharing, which is once again the main purpose of the protocol. Therefore, there have also been no efforts to support any mobile platforms. Because of these aspects, *FastTrack* is excluded from the list of possibly useful background technologies.

eDonkey

Similar to *FastTrack*, also the *eDonkey* protocol is proprietary. There have been some freeware implementations of *eDonkey* server software, but they are not open-source. Also,

there is no ongoing development on *eDonkey* clients since years. The same issues extend to the successor protocol *Overnet*. Configuration of the firewall is needed for two peers behind different NATs to be able to communicate. These protocols are therefore no choice for a possible useful technology for the proposed framework.

BitTorrent

BitTorrent in contrast is open-source and the protocol is in ongoing development. Furthermore, there exist open-source clients, even for mobile platforms, that are also in further development. There have been other use-cases for the protocol than just P2P file-sharing, e.g. for the *vuze BitTorrent* client there exists a chat plugin that works in a P2P manner on the current *BitTorrent* swarm (see the *vuze* website[1] for more details). The problem when considering the formulated requirements on the framework is that with *BitTorrent* it is not possible to bypass NAT in every constellation. Different clients use different techniques like port forwarding, UPnP or UDP hole punching to achieve that. Eventually, either a configuration at the router or firewall is needed or hole punching techniques have to be applied, which also do not work on every NAT setup, for more details why see [Til17]. Also the scalability is an issue of the protocol, because one swarm always belongs to one central tracker, which could probably become a bottleneck. Because of all these reasons, *BitTorrent* cannot be the technology of choice for communication in the framework implementation.

JXTA

JXTA seems to be a quite good candidate for selection as background technology. It is open-source and supports many P2P features additionally to communication. NAT traversal is possible with the framework by using peers as relays. Multiple different, underlying transport protocols can be used. Security measures have been implemented in the framework in an adequate manner, at least for unicast P2P messaging. Also, at least *edge peers* have already been ported to *Android*. Porting of other components to *Android* seems feasible. The contras of JXTA as a choice may be that it is quite heavy-weight and the state and licensing of the implementation seems obscure and chaotic at the moment. Also, a framework based on JXTA would need a complete re-implementation of all the features offered by JXTA on an *iOS* implementation. Nevertheless, JXTA is a candidate to base the proposed framework on.

P2PSIP

P2PSIP seems to be a quite promising concept and the ongoing work on the RELOAD implementation of it might bring this concept to wider popularity in the future, but it still seems too early to base another framework for mobile devices on it. There are still some important missing parts, e.g. the implementation of a fully working P2PSIP client on a mobile device. Additionally, RELOAD needs an underlying structured P2P

[1] `https://wiki.vuze.com/w/Chat_plugin` accessed 01.2017

	Napster	Gnutella	Freenet	FastTrack	eDonkey	BitTorrent	JXTA	P2PSIP
FR1 - Coordination	-	-	-	-	-	-	-	-
FR5 - Connectivity	+	+	-	+	+	-	+	+
NFR1 - Licensing	~	+	+	-	-	+	+	+
NFR2 - Scalability	~	+	+	+	~	~	+	+
NFR3 - Security	-	-	~	-	-	-	~	~
NFR8 - Operability	~	~	~	-	-	+	+	~
NFR10 - Resource-efficient	+	-	~	+	+	+	~	~

Table 3.1: Probable fulfilment of requirements by presented P2P technology

network as a plugin. Choosing this concept as background technology would mean having to implement the full protocol for a mobile platform, also including the plugin for the underlying structured network. This seems to be too much effort and too risky for a protocol that is not even fully defined. Moreover, using mobile devices solely as SIP clients, which have to connect to a super-peer running the full P2PSIP software, is not compliant with the intentions of the mobile peer being self-contained and a first-class component of the framework. This concept is therefore no candidate for selection.

Overview of technology and fulfilment of requirements

Table 3.1 shows which of the most important requirements from Chapter 3 regarding the P2P communication technology could be fulfilled (+), fulfilled partly or with additional effort (~) and not fulfilled (-) by which presented technology. The most important requirements have been selected considering which requirements are essential for the proposed framework and when not fulfilled by the background technology would cause big implementation efforts or would break the whole design. The requirement *FR1* could not be fulfilled by any of the described P2P protocols and frameworks, which made it necessary to additionally chose a coordination framework or implement a coordination model and integrate it with the P2P technology.

Coordination frameworks and models

In Section 2.5 several coordination frameworks and models have been presented, on which the proposed framework could be built on. Effectively, all of them are based on some sort of space-based technology or do suggest to use one for the data storage layer. *DataSpaces* is intended to be used in a scenario, where nodes produce massive amounts of data in a grid-computing environment. Furthermore, like also *DTuples* and *Comet*, the data storage in the formed network is distributed with a DHT, that means that nodes do not decide which data they store, this is managed by an overlying algorithm. *TuCSoN* and

Comet are additionally quite restrictive on the used underlying space-based technology, defining it as a concrete *Linda* based approach with specific functionality. Only CArtAgO and the PM are not restrictive on the underlying data storage technology and also do not use a distributed storage, which contradicts the intention for a peer in the framework to be self-contained. Therefore, *CArtAgO* and the PM seem to be the best candidates as coordination models to base the proposed framework on.

Selection

After a mutual discussion with Peter Tillian on the final selection of technology and evaluation of the possible candidates that emerged from both researches, the outcome was that XMPP is the best choice as underlying technology. The protocol offers P2P communication with an authenticated partner over an XMPP server or server cluster, so the centralized part is scalable. The server role includes relaying of messages and identity management. There exist open-source client implementations for *Android* and *iOS*, which are in further development. Moreover, there exist open-source XMPP relay server implementations. Security measures include optional traffic encryption and blocking of specific peers, as well as a central identity provider including a user presence protocol. NAT traversal is possible by design in any network constellation because of the relay server. For further details on the XMPP protocol see [Til17].

The reason XMPP has been preferred to JXTA is that also JXTA needs a relay peer (analogous to the XMPP server role) to bypass any NAT, but it is more heavy-weight. JXTA, besides just P2P communication and identity management, offers many features that would probably not be needed by the framework implementation and need more storage and computing power on a mobile device. XMPP fulfils the imposed requirements without making a client application too heavy-weight. Furthermore, the P2P communication layer can be implemented with low effort on *Android* and *iOS*, because of existing client libraries, whereas there are no implementations of JXTA for e.g. *iOS*. Also, the limbo state of the JXTA project at *Oracle* is not promotive.

The PM is chosed as coordination model. There are different reasons for this choice. Firstly, the advisor and the assistance advisor of this thesis are both among the authors of the PM at the TU Wien and besides their personal support also weekly meetings of the PM technical board can be used to discuss the implementation of the model. Secondly, without consideration of that, the PM also seems to be the most intuitive and pragmatic model amongst the found approaches compliant to the stated requirements. The model is quite easy to understand, is P2P based with self-contained peers and can be adapted to a mobile profile of the original full specification to be compliant to restrictions of mobile devices most effectively. Furthermore, there has already been a successful implementation of the model for the *.Net* platform [Rau14] and there are ongoing implementations for *Java* and *Go*. With help of the thesis advisor Prof. Dr. Eva Kühn also the support of the authors of these implementations can be used.

CHAPTER 4

The Peer Model

This chapter is dedicated to the PM, that has been proposed and determined by Kühn, Craß, Joskowicz, Marek and Scheller in [KCJ$^+$13]. The intention of this work is to design an optimized, compatible profile of the model for mobile devices and deliver an implementation of it for the *Android* platform. The MPM thus represents the mobile profile of the PM with its adaptations and optimizations for mobile devices. In this chapter the original and full PM is presented.

4.1 Characteristics

The PM defines a programming model inspired by tuple space communication and event- as well as data-driven architectures, especially targeted on heavily distributed applications. By design of the model, any business logic is treated as a black-box within so-called services. Only coordination and communication parts of the system are described. To bootstrap the model, an underlying tuple-space is assumed to be present to store the data and requests that flow from the input, through the internal logic to the output stage of a peer in the PM. A space as described in [CKS09] is used, which provides shared containers offering configurable coordination mechanisms, like explained in [KMKS09], and a flexible API. Similar space-based technologies may also be used in the model.

Entry

Entries are objects that represent data and requests in the PM. There are system properties (coordination data) and application properties attached to each entry. System entry properties are always existent, application properties can be defined and added by a developer modelling a specific application. Subsequently, the system properties of every entry shall be explained.

Every entry has an *entry-type* property that is queryable and used for selection of entries in the model. It differs from the type of the application data transported by the entry. Besides the *entry-type* (*type*), the *application-data-type* (*dataType*) and the *application-data-object* (*data*), there exist further system properties like the URI of the peer that originally created the entry (*origin*). Furthermore, if the entry has been received from another peer, the URI of that peer is stored in the *from* property. When sending an entry to another peer, the destination peer's URI is stored in the *dest* property. Moreover, a profile of the PM usually defines a time-to-start (TTS) (*tts*) and a TTL (*ttl*) on the entries to be able to define spans of validity. Entries will only be processed in the model if their TTS has been reached and their TTL has not yet been reached. The *dataType*, the *origin* and the *from* properties have not been defined in the original paper but are extensions proposed by the PM technical board.

Container

Containers are used in the PM to hold data and requests represented as entry objects and are part of the underlying space-based technology. Containers are referenced by a URI in the network and provide an API to *write* (put into), *read* (retrieve without remove) and *take* (retrieve and remove) entries. These operations are carried out in transactions and also support bulk processing by specifying a counter on the operations to apply to specific amounts of entries (exact number, a minimum or a maximum). The operation also always includes a type identifier to match a specific *entry-type*. Selection order of *read* and *take* operations is configurable, depending on the underlying space-based technology. The designers of the PM suggest usage of an underlying space-based middleware like proposed in [CKS09], which offers several predefined selectors like the *any-selector* (no order guaranteed) or the first in - first out (FIFO)-selector (first in - first out order). These selectors can be combined and besides others there exists also a *query-selector* where custom queries can be defined on entry properties.

Peer

The peer represents a node in the PM network and is addressable by a URI. Two containers are part of a peer. There is a peer-in-container (PIC), where entries are received and a peer-out-container (POC), where processed entries are stored. The processing is handled by services within one or more so-called wirings of the peer. Processed entries can then be delivered from the POC to the PIC of another peer (destination URI). Peers may also contain one or more sub-peers within their scope. All peers of a local site referred by a URI form a *peer space*, whose runtime environment is bootstrapped via a runtime peer (RTP).

Wiring

Wirings handle the transport of entries between the containers of a peer (and contained sub-peers) and a communication with a remote peer is possible using the *dest* property

of an entry. A wiring consists of guards, services and actions. Guards and actions consist of guard links and action links and define specific operations on those (the terminology of the original PM paper [KCJ+13] is used in this work). The link operations on a guard specify which types of and how many entries are taken or read from the PIC of a peer or the POC of a sub-peer using the internal API of the container, where at least one take operation must be specified to avoid easy creation of endless loops. All the guard links must be satisfied in sequential order for the wiring to be executed in an own transaction. If that is not possible, the wiring will not be executed. If all guard links can be satisfied, the entries are taken or read and stored in a temporary entry collection (EC) visible to this wiring, which has the same functionality as a space container. The EC is then handed over to the service(s) of the wiring, which are executed sequentially and have access to the EC for processing the entries and writing resulting entries to. After processing all services, the action links are executed in sequential order. They define which and how many entries in the resulting EC are distributed to either the PIC or the POC of the peer or the PIC of a sub-peer. Not all action links must succeed, they are just executed if they can be satisfied, otherwise they are skipped.

Figure 4.1 shows an example of a peer *P1* having one sub-peer *P2* in the graphical notation of the PM. The peer contains a single wiring *W1*. If the guard of the wiring can be satisfied, it takes one entry of type *T1* and reads all available entries of type *T2* from the PIC. The entries are collected in the EC and handed over to the two services *S1* and *S2* of the wiring. After execution of those two services, one entry of type *T4* and all entries of type *T3* are taken from the EC by the action links (if these entries are available) and are placed in the POC of the peer to be distributed to other peers. One entry of type *T1* is handed over to the PIC of sub-peer *P2*, if the entry is available in the EC after processing. As you can see, entries with type *T3* and *T4* might be created by the services, whereas *T1* and *T2* might be provided by another peer or by the RTP at start-up.

The processing code of the services is defined by the developer and contains logic that is not part of the PM. Nevertheless, within the service the entries that have been handed over to it in the EC can be accessed and the *dest*, *data*, *TTL* and *TTS* properties can be set or new entries can be created. Wirings connect peers and services in a data-driven way that provides high decoupling. All the wirings of a peer can run concurrently and represent the only active part of the PM.

Flows

To model global tasks, so-called *flow identifiers* are attached to entries that belong to one flow, corresponding to a workflow in an enterprise system. Each wiring only fetches entries in its guard links that belong to the same flow, identified by this unique identifier. With that concept multiple different workflow instances can be processed in parallel, e.g. data from different users.

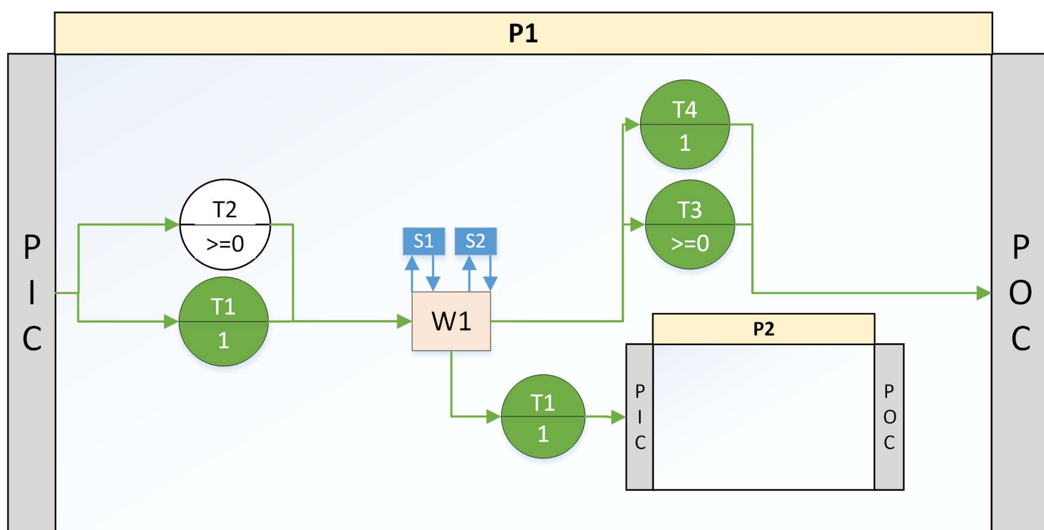


Figure 4.1: Example: Graphical notation of a peer with one sub-peer, one wiring and two services

CHAPTER 5

Design

The aim of this work is to implement a mobile P2P communication and coordination framework. After background technologies have been researched, evaluated and selected in Chapter 2 and 3 and the PM, on which the framework shall be based on, has been introduced in Chapter 4, the integral parts of the software engineering process regarding the conceptual design of the framework shall be described in this chapter.

The engineering process has started with the assessment of the requirements on the MPM in Chapter 3. In the following sections, important details on the design of the framework are provided. The two engineering processes are done in cooperation and synchronization with the work of Peter Tillian [Til17]. Detailed insights on who has done which parts and tasks are provided.

Both scalability and security are main focus points of this work and are emphasized in the design phase.

5.1 Mobile profile of the PM

In Chapter 4 an overview of the full formal specification of the PM with some important features has been provided. In several meetings agreements with Eva Kühn and Stefan Craß, who developed the PM, have been made to establish an adjusted profile of the model, suitable for mobile devices and use cases. The outcome was a reasonably trimmed model, having only needed and supportable features for mobile platforms. Nevertheless, the mobile profile and the reference implementation of it are just an initial version and might be extended by additional features in the future. In this section, the features of the MPM in contrast to the full PM shall be presented.

5.1.1 Runtime peer (RTP)

There is a single RTP on every node participating in an MPM network. The RTP is not a usual peer in the MPM. It offers interfaces to create peers, contains all local peers forming a peer space on the mobile host and can be started and stopped. Local peers are uniquely addressable by a peer name in the namespace of the RTP. Moreover, the RTP exposes an interface to communicate with an outer system or user interface (UI). The concept of the RTP has already been introduced by Kühn et al. in [KCJ+13].

5.1.2 Entry

In contrast to the entries of the full PM, presented in Section 4.1, the MPM does not support additional, queryable application specific properties on entries that can be defined by a developer. All system properties of an entry are the same as in the full specification, including the data type, the data object, TTS and TTL.

5.1.3 Container

The underlying space-based technology is implemented by the reference implementation, no existing implementation of such a model is used. Reasons are that underlying space containers can be defined and optimized for the concrete purpose of the MPM and do not use unnecessary storage and computational resources in the optimized and lightweight mobile framework, like an implementation of a sophisticated space-based middleware in the manner of the proposed approach from [CKS09] would do. Nonetheless, the design allows the replacement of the underlying container technology by an implementation that supports the mentioned operations below by easily disposable interfaces.

Containers, like in the PM (see Section 4.1), provide an interface to put (*write*), non-destructively retrieve (*read*) and destructively retrieve (*take*) entries. Retrieving operations offer a string type parameter on the operation to match entry types (an empty string will match any entry type). Bulk operations are supported on the retrieving operations by applying a *count* parameter. The parameter specifies the selection of a specific amount (exactly n, where $n > 0$) of entries greater than zero or an amount greater than or equal to a specific number ($>= n$, where $n >= 0$). The $>= n$ *count* operator would select as many entries as possible but at least n to be satisfied, otherwise the operation would not be successful. Additionally, there exists the possibility to specify a maximum amount of entries for a bulk operation ($<= n$, where $n > 0$). If more entries than the specified maximum and type would be present in the container, the operation would still only be performed with the specified maximum number of entries but would be successful.

A wiring specifying link operations with a *count* $>= 0$ or a *count* $<= n$ have to specify at least also one *take* guard link with *count* $>= n$ where $n > 0$ or *count* $= n$ where $n > 0$ to avoid easy creation of endless loops in the execution of the wiring. This is necessary because such guards would always be satisfied and the wiring execution would loop forever.

5.1.4 Peer

A peer in the MPM is, like in the PM (see Section 4.1), a self-contained part of a local *peer space* contained in a locally hosted RTP. The peer holds two containers, PIC and POC, like in the full PM specification. Nevertheless, the POC in the implementation of the MPM is only a virtual container. Internally action links will forward respective entries to the PIC of a system-defined input/output (IO) peer, which then transports the entries to the container of the designated peer or internally to the PIC of the same peer. For better understandability and encapsulation the POC will still be shown in the following graphical notations of the MPM like in the full PM notation. Wirings with guards, services and actions are defined analogously and are using the operations offered by containers of the MPM. Inter-peer communication is possible by sending entries to local peers or to external peers hosted in a remote RTP.

The peer URI of the MPM, by which a peer is uniquely addressable in an internet scale network, is a combination of the host name of the device hosting the local RTP and the local name of the addressed peer, both represented as strings. The structure of the host name depends on the used communication protocol and possibly used identity provider. Therefore, also the used communication protocol shall be encoded in the peer URI. Ultimately, the peer URI shall be encodable as a string *<protocol>://<host-name>/<peer-name>*.

In contrast to the full PM specification, the MPM does not support sub-peers in the first version. The only peer that contains other peers is the special RTP.

In the MPM there are statically defined system peers, which are provisioned automatically in every RTP and are bootstrapped like usual local peers, but have special tasks. These system peers include IO peers for sending and receiving entries to and from external peers, as well as an exception peer, which is responsible for handling exceptions that might occur within the framework.

5.1.5 Wiring

Wirings are defined analogously to the wirings from the PM (see Section 4.1). The difference regarding wirings is that no sub-peers can exist, so wirings cannot deliver entries to the PIC of a sub-peer or retrieve wirings from a sub-peer's POC. Furthermore, only one service is part of a wiring, so multiple services within one wiring, like in the full PM, have to be combined in one.

Flows (see Section 4.1) are not supported by the MPM in the first version, so guards in wirings do not consider flow IDs when retrieving entries from the container. Entries cannot contain a flow ID. Therefore, in the prototype reference implementation workflows with different peers would have to be handled by the application logic (data object of the entry). Nevertheless, the support of flows has the highest priority of planned extensions to the framework.

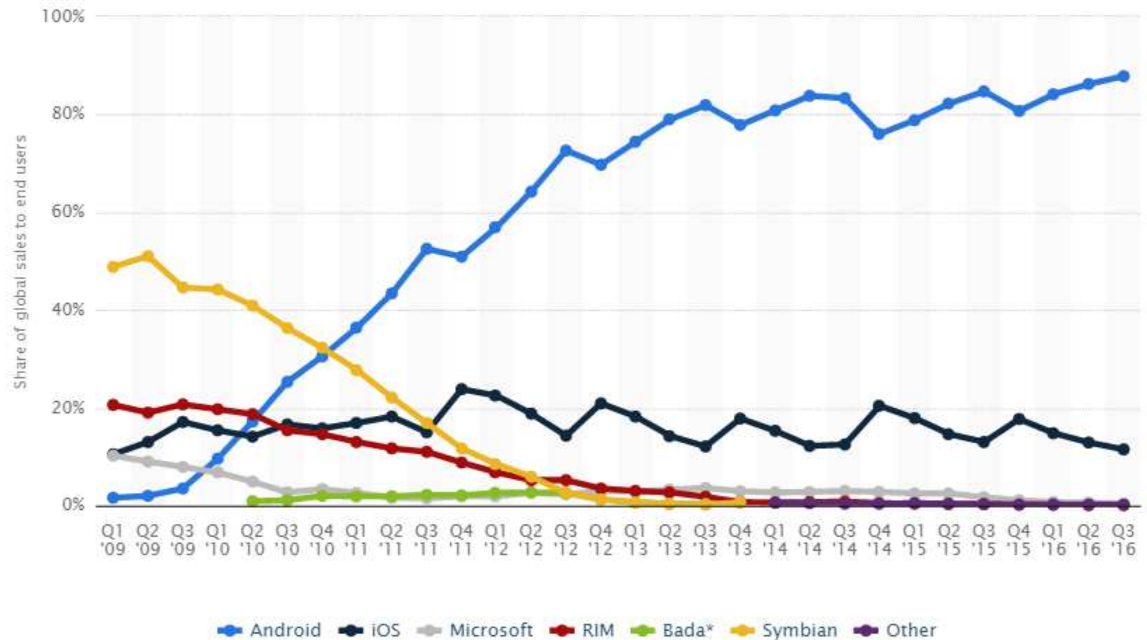


Figure 5.1: Global market share of operating systems for mobile devices[1]

5.2 Architectural overview and separation of engineering tasks

This section provides an architectural overview of the system by means of simplified figures and explanations. Moreover, information is given on which components and architectural layers of the framework have been implemented by whom, because the software engineering process, like mentioned, is a cooperation work.

5.2.1 Decisions on background technology

In Section 3.2.1 detailed information and reasoning have been provided why XMPP has been chosen as background technology for communication and identity management for the reference implementation and why the framework implements a mobile version of the PM. The operating system for the host of a RTP is *Android* in the reference implementation, because firstly *Android* at the moment has the highest market share of mobile operating systems (87.8% at Q3 of 2016), like shown in Figure 5.1. Secondly, the implementation of the core MPM framework is then in Java, which also can be run on any machine that supports the installation of a Java virtual machine (JVM), which is possible on nearly every desktop and server operating system.

[1] https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/accessed01.2016

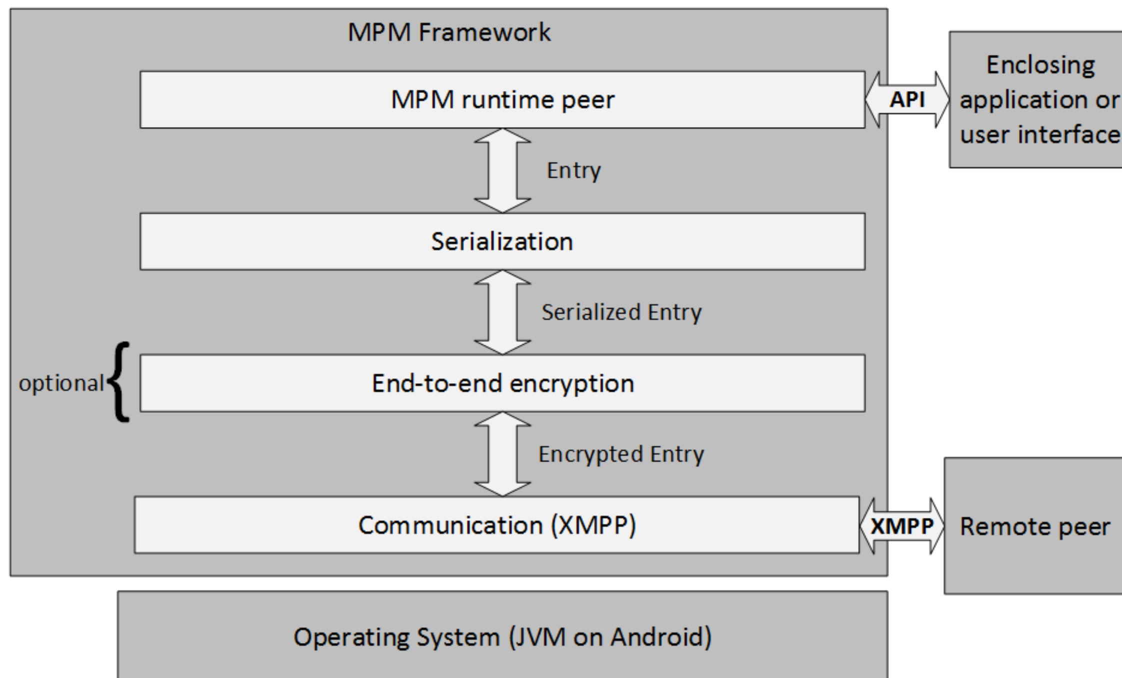


Figure 5.2: Components of the MPM framework including communication with enclosing application and remote nodes

5.2.2 Architecture of a node

Figure 5.2 shows the architecture of a node in the reference implementation of the MPM. The framework runs on top of the underlying JVM, either on an *Android* device or on any other machine that can run a JVM. The core framework itself consists of four components, which build on one another. The communication component is implemented using XMPP. It encapsulates the whole remote communication functionality, the exchange of entries with remote peers over XMPP. For security reasons, an encryption component is placed on top of the communication component. The component offers optional E2E encryption of entries. To be able to send entries to remote peers over the network, the entries have to be serialized at the sender and de-serialized at the receiver. Therefore, the next component on the stack is the serialization component, which takes care of serializing and de-serializing entries. The actual RTP is then placed on top of the serialization component. Like explained in Section 4.1, it contains special IO peers that receive entries from the bottom to the top in the component stack and send entries from the top to the bottom in their implementations. Also, the RTP offers an API to interact with an enclosing application, e.g. an *Android* service or a UI.

5.2.3 Distribution of work

Design and implementation have been separated between the work of Peter Tillian [Til17] and me in the following way. Peter Tillian was responsible for the design and implementation of the MPM RTP with all its contained components as well as optimization of the framework regarding restrictions of mobile devices in comparison to desktop or server machines. Also, Peter was responsible for the RTP to be able to be encapsulated in an *Android* service and to be able to interact with the UI. One of his tasks was also the delivery of a ready-to-use *Android* service that encapsulates the MPM RTP and offers all necessary interfaces to be able to interact with an outside *Android* application using the service.

My task in the work was the design and implementation of the communication, the E2E-encryption and the serialization components, as well as designing the system to be scalable. This includes research, evaluation, selection and integration of appropriate protocols for each of the components. Furthermore, all considerations about the security aspects in the framework were my responsibility. Although, engineering processes have been separated, we kept in contact to support each other with tasks and discuss important decisions.

5.2.4 Architecture of the RTP

Although detailed design and implementation decisions are in the scope of Peters work, some broad outlines on design decisions have been made in mutual agreement. The coarse overview of the design of the RTP with its important components shall be presented also in this work to be able to understand the following chapters.

Figure 5.3 shows the architecture of the RTP with its components and also the flow of entries into, inside and out of the RTP. On the bottom of the graphics you can see the RTP object, where all the components explained above are contained. The object holds a list of local peers as well as an interface for an external system (in this case called user interface). It offers functionality for peer creation and for connectivity event propagation from an outside system. The RTP can receive entries from remote peers via the receiver peer, which has an interface to the underlying communication protocol. The receiver peer reads the destination property (*dest*) of the incoming entry, looks up the addressed local peer in the peer lookup list of the RTP and delivers the entry to the PIC of the right local peer. The externally received and sent entries are marked with an E in the figure. The receiver peer is, like the sender peer and the exception peer, a local system peer, which is automatically provisioned for every runtime. The RTP can furthermore contain several local peers that are addressable by name in the name-space of the RTP, forming a local peer space. These local peers have to be created by an application developer using a provided peer creation interface on the RTP at design time. All of them contain a PIC and can contain several wirings with guard, service and action, like explained in Section 5.1.5. Entries flowing through wirings can be received externally (E), from another local peer (I) or be newly created inside a service or by the user

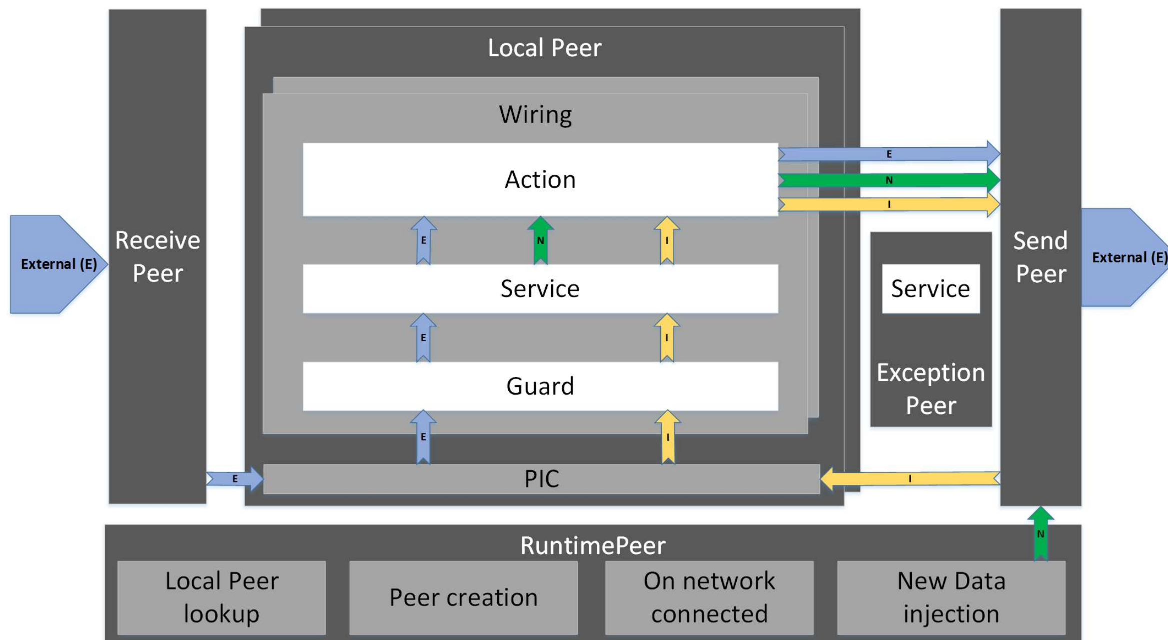


Figure 5.3: Architecture of the MPM RTP including important components and flow of entries

interface (N). When an action fires, the entries get delivered to the sender system peer, which investigates the destination property of the entry ($dest$). If the destination is a remote peer, it forwards the entry to the underlying communication protocol for network delivery. If the destination is a local peer, it directly writes the entry to the PIC of this peer. Last but not least, there exists the exception system peer, which is like the other two system peers bootstrapped like a common local peer, but already has predefined wirings. If an exception happens inside a service of a wiring, the application developer might want to handle that exception. Therefore, there exists the possibility to wrap the entry on which the exception occurred into an exception entry, which then gets delivered to the exception system peer. This can happen in every MPM peer except the exception peer (exception entry flow is not shown in the figure). In the service of the exception peer, it is up to the application developer to handle those exceptions. An exception that can occur already in the scope of the framework is the expiration of the TTL of an entry. The framework automatically wraps such entries into exception-entries and propagates them to the exception peer to be handled by the developer in the exception peer's service. How exceptions can be handled by the developer in the exception peer and what happens when an uncaught exception is thrown inside of a service is described in detail by Peter [Til17].

5.3 Architecture of communication and identity management

This section presents the design of the communication between nodes of the MPM framework. Because the requirements state that communication should work in an internet-scale network and from behind any NAT, the task is quite sophisticated. Fortunately, XMPP offers quite a large set of features that help fulfil a set of requirements. The communication is relayed over an XMPP server and therefore NAT traversal is guaranteed. Also, it serves as an identity provider and offers scalability by clustering of the server and security by encryption over TLS. Details on how an identity is provided for a joining node and how communication is set up in the proposed framework are provided in the following section.

5.3.1 Joining the network and communication with other peers

In Figure 5.4 a detailed overview of the communication model is given. There are three different peers involved, two client peers (*Client Peer A* and *Client Peer B*) that want to communicate with each other, as well as a *notification peer*. Furthermore, three different servers are part of the communication model, a *registration server*, the *XMPP server* and the *Google Firebase cloud messaging (FCM) server*.

The numbering from *(1)* to *(14)* indicates the information flow from registration of *Client Peer A* in the MPM network until a successful communication with *Client Peer B* is done by sending an entry. The starting situation of the shown graph is that *Client Peer A* has not yet registered and joined the network, but wants to send an entry to *Client Peer B*, which has already joined.

5.3.2 Registration

The first action that has to be performed is the registration and creation of an identity for *Client Peer A* at the XMPP server. Because *Client A* does not yet have an XMPP identity, it cannot communicate in the network by sending entries over XMPP, therefore an external registration has to be done, which is the purpose of the registration server role and is not in the scope of the network. The registration server uses the user creation API provided by the XMPP server and exposes it in an adequate and secure manner to a client peer. In combination with the core MPM framework, a registration server implementation in Java is delivered, which offers all needed registration functionality over HTTP endpoints. Different implementations of the registration server that use the user creation API of the XMPP server may easily be implemented. In the registration process, a client sends a registration request *(1)* to the registration server, which then creates an identity in the XMPP database *(2)*. This database is used as identity database by the XMPP server. After successful registration, an acknowledgement message (not shown in the graph) is sent back to the new client by the registration server. Now the client has a valid identity to join the network. Of course, the registration server must be

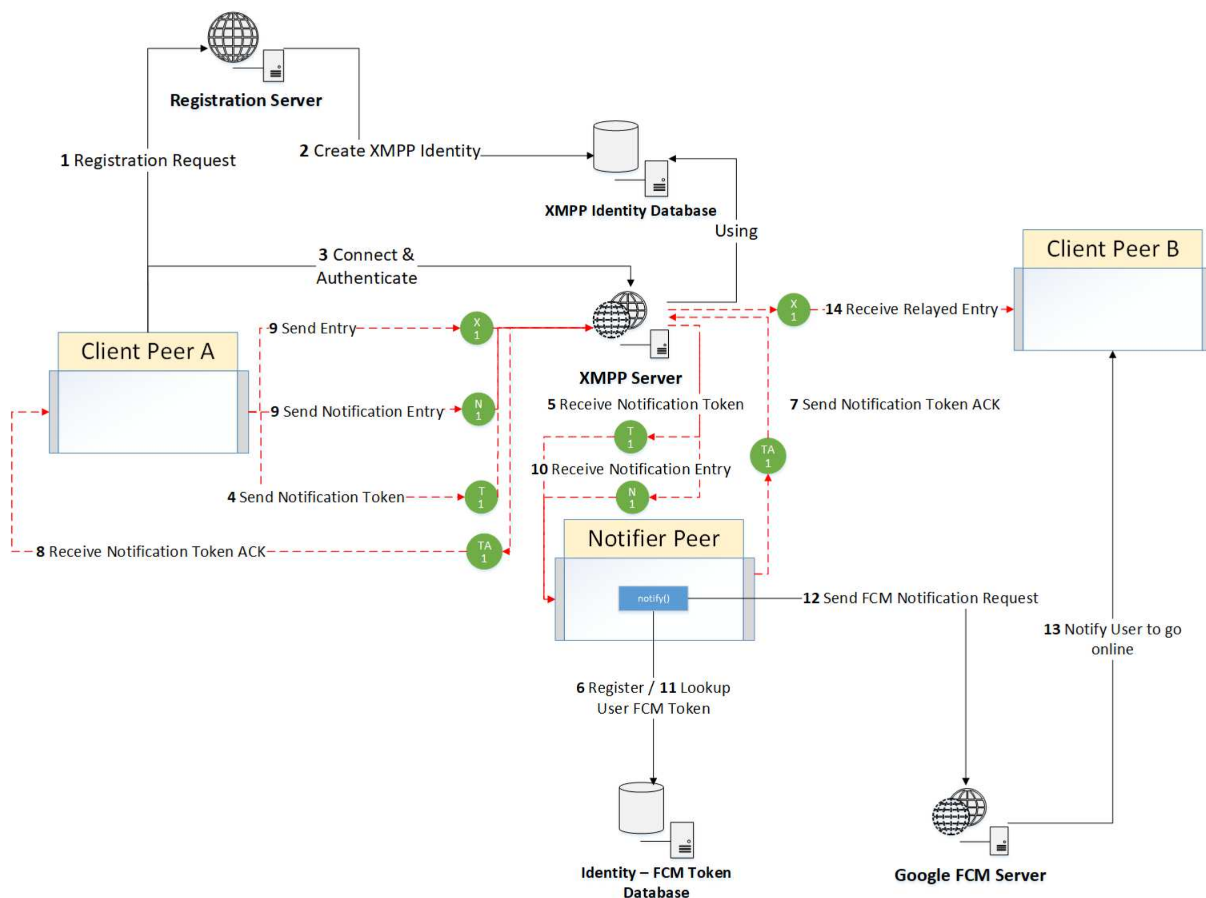


Figure 5.4: Architecture of the access control and communication in the MPM network

trusted and communication with the client must be secured by encryption and verified by a trustable server certificate. Also, the registration server role can be hosted at the same physical machine as the XMPP server and be validated by the same cryptographic server certificate. Figure 5.5 shows a sequence diagram of the registration process a new node has to go through.

After successful registration in the network, the user can join the network by connecting and authenticating to the XMPP server *(3)*. This is done by credential based authentication (username and password). After the connection to the relay server has been established, the client is able to send data, represented as entries, in the scope of the MPM network.

5.3.3 Notifications

The next important communication component that shall be explained here is the role of the notifier peer. This special peer is bootstrapped as a normal MPM peer and has a

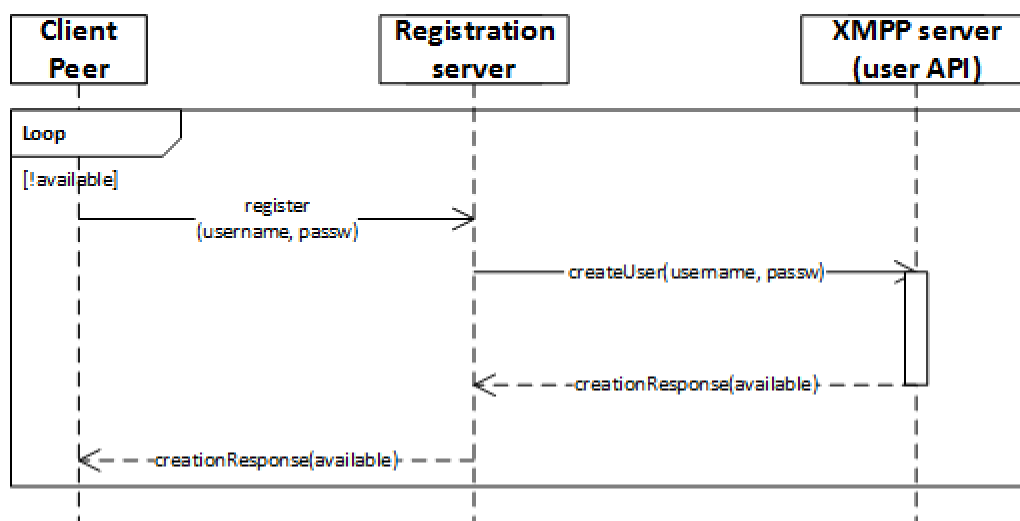


Figure 5.5: Sequence diagram of a new node registering at the registration server

well-known, configurable XMPP identity so that it can be addressed by all nodes. The task of the notifier peer is to notify users about availability of an entry at the XMPP relay server. The notifier peer can be hosted on any server machine that can run a JVM and is a peer in the network like any other peer but contains predefined wirings and services to provide the notification functionality for the network. Client peers shall not be required to maintain a constant TCP connection to the relay server. The concept of notifications has been elaborated in synchronization with the research of Peter Tillian, who analyzed issues with network connectivity of mobile devices. Mobile peers understandably cannot achieve static network connectivity, because the device might be offline for various reasons, like connection handover, no phone coverage or the termination of the mobile MPM based application by the user or by the mobile operating system. For example, the current *Android* version might easily terminate a service to save battery and to free memory.

To enable the possibility to awaken a mobile application or to notify the user and the application about new incoming data, the *Android* platform maintains a constant connection with the Google FCM network[2]. Over this connection the device might get notifications as soon as network connectivity is established. After that, the programming model of an *Android* application allows handling of such notifications and binding them to a specific application on the device by using a unique token, which is called notification token in this work. By binding the incoming notification from the constantly established FCM connection to an application, the *Android* operating system might start the application or might just display a notification for the user to open it. This concept is also used in the MPM framework. Here, the developer can configure if a notification shall be displayed or not. By default, the RTP is started, but no notification is displayed. Then, whenever the MPM based application is opened, the RTP automatically connects

[2]`https://firebase.google.com/docs/cloud-messaging/` accessed 08.2017

to the XMPP server and receives the newly available entries. The FCM network uses also XMPP or optionally HTTP. The concept enables *Android* devices to maintain only one open TCP connection and still offers the opportunity to notify and start applications when incoming data arrives.

At first start of an application that is based on the MPM framework, a globally unique FCM notification token is generated using the functionality of the FCM libraries provided by *Google* and sent to the notifier peer, wrapped in a notification token entry *T (4)*. This peer manages a database with a mapping between identities of peers and their FCM tokens. Whenever the cache of the *Android* application gets cleaned, the application is re-installed or the FCM service renews the token (FCM server sends a notification about that to the device), an FCM token update with a new notification token is sent to the notifier peer. The notifier peer receives *(5)* the notification token entry *T*, stores the identity - notification token binding in the database *(6)* and sends a notification token acknowledge entry *TA* back to the client peer *(7)*. After the client has received the acknowledgement entry *TA (8)*, it can be sure that other peers might send notifications to it using the functionality of the notifier peer and the FCM network. As the reference implementation of the core MPM framework is runnable on any operating system that can host a JVM, the notifier peer might be hosted e.g. on a cloud virtual machine (VM) or cloud container service.

5.3.4 Sending and receiving entries

When *Client Peer A* sends an entry *X (9)* to *Client Peer B* this automatically triggers an additional notification entry *N*, containing the identity of the destination peer *B*, to be sent to the notifier peer *(9)*. Both entries get cached at the relay server. *Client Peer B* might not be online at the moment, but at least one notifier peer should always be available, which receives the notification entry *N* containing the identity of *Client Peer B* that shall be notified. The notifier peer looks up the FCM token of *Client Peer B* in the database *(11)* and sends an FCM notification request to the Google FCM server (12). The destination peer's connection to this server and the binding of the token to the device of the destination peer is handled by the destination peer's *Android* platform, which has registered the token also at the FCM server. Therefore, the FCM server can lookup the connection bound to the token and forward the notification request to *Client Peer B*. As soon as peer *B* receives the notification request from its FCM connection, the *Android* system looks up to which application the incoming token is bound and calls a callback function that opens the RTP, which then connects to the XMPP relay server and receives the designated entry *X (14)* and all other entries that have been received at the relay server from *Client Peer A* or other participating peers. Logically, notifications over the FCM connection can be ignored if the RTP is already running and connected to the relay server as it then will instantly receive entries.

The sequence diagram in Figure 5.6 shows the delivery process of an entry from one peer to another. First, the entry and the additionally generated notification entry are sent to the XMPP server in parallel. Then, two different sequences can follow, depending

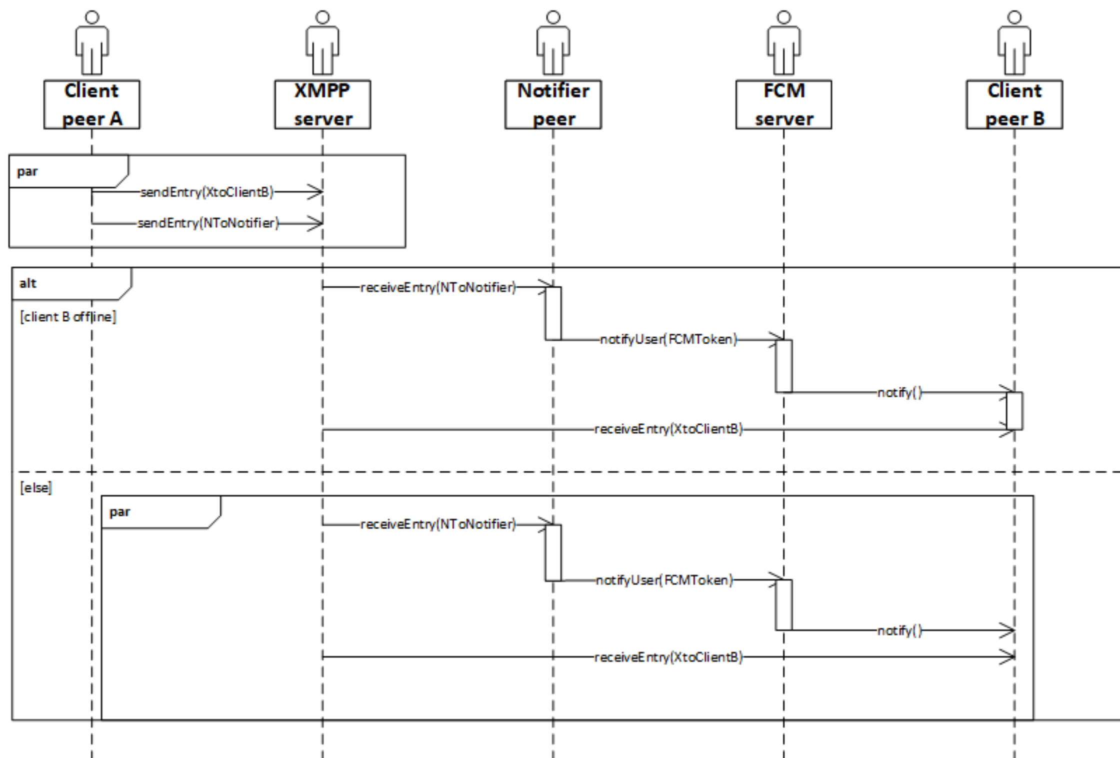


Figure 5.6: Sequence diagram of a client peer sending an entry to another client peer

on whether the destination peer is online or not. The sending peer can not know if the destination peer is online when sending the entry, therefore a notification entry has to be sent to ensure that the destination peer gets notified. If the destination peer is online, it can just ignore the incoming notification entry. If it is not online, the notifier peer receives the notification entry containing the destination peer's identity, looks up the destination's FCM token (message exchange with the database is not shown in the diagram) and then notifies the destination peer over the FCM network to go online and receive the entry from the relay server. This sequence of messages happens in this strict order. If the destination peer is already online, the entry might even be received before the notification entry arrives (second, parallel sequence in the diagram). The messages that are automatically sent when a peer is connecting to the relay server are not shown in the sequence diagram (e.g. when the destination peer gets notified and connects to the XMPP server). These messages are specific to the communication protocol and are handled by the XMPP implementation in the framework (see [Til17] for further details on XMPP).

5.4 Architecture of serialization

This section gives information how entries are delivered in the network. To be able to transport entries to remote peers by means of the communication component of the framework, they have to be serialized at the sending peer and de-serialized at the receiving peer. In the reference implementation, at least one widely used and very efficient option for serialization and additionally one human readable form of serialization shall be provided. The human readable serialization is mainly intended for debugging purposes. During research on serialization protocols that work well on mobile devices, it became clear that human readable formats are usually based either on JavaScript object notation (JSON) or XML in American Standard Code for Information Interchange (ASCII) encoding. Then, there exist protocols that provide binary formats. Usually, the serialization and de-serialization performance of ASCII-based formats is much lower than of binary-based formats and the used storage size for ASCII-encoded formats is much higher. The question may arise why the framework implementation does not just use programming language provided serialization, like e.g. Java object serialization. The reason is simply that it should be possible to send data to heterogeneous peers, which might be implemented in another programming language. Therefore, there has to be an agreement on the data format that both can interpret.

5.4.1 Research

Sumaray and Makki did a comparison on efficiency of serialization formats on a mobile platform (*Android*) in [SM12]. Another one has been done by Maeda in [Mae12]. There, besides XML and JSON, I identified *Google protocol buffers* (protobuf) and *Apache Thrift* as prominent and widely used candidates. They are also very often included in publicly available open-source performance benchmark tests like this one on *GitHub* (see the project website[3]). Another possible candidate that has been identified is *Apache Avro*. All three are acknowledged examples of efficient binary serialization formats, but as *Apache Avro* does not support Objective-C or Swift (programming languages for *iOS*) (see the *Apache* documentation website[4]), it gets excluded, because then no serializer for *Avro* would be available for a mobile peer on an *iOS* device.

5.4.2 Selection

As the provided references and the benchmark comparison for the JVM show, JSON serializers are almost always faster than XML serializers. So JSON is the format of choice for the human readable implementation of the serialization component for the MPM framework. When doing research on *Android* forums and blog sites, two candidates seemed to be widely used and approved JSON serializers for the platform, namely

[3] `https://github.com/eishay/jvm-serializers/wiki` accessed 01.2017

[4] `https://cwiki.apache.org/confluence/display/AVRO/Supported+Languages` accessed 01.2017

Google Gson (see the project website on *GitHub*[5]) and *Jackson* (see the project website on *GitHub*[6]). They are both open-source, in continuous development and seem to be the most complete libraries for Java to JSON conversion, supporting deep inheritance structures and Java generics etc. Both are licensed under the Apache license 2.0 (see the license specification on the Apache website[7]), which fulfils the requirements on the licensing. Although *Jackson* seems to perform insignificantly faster on large files, the API of *Gson* looks more simple, having only a *toJson(instance)* and *fromJson(String, class)* interface for the serialization and de-serialization. Also, the project size is more lightweight. As the JSON serializer is easily implementable and for plain Java classes does not need any additional implementation, *Google Gson* has been chosen for the framework implementation, which could be extended and exchanged with *Jackson* with very low effort.

Apache Thrift and *Google protobuf* both support Java, Objective-C, C, C++, Go and further important programming languages and are both open-source. *Google protobuf* is licensed under the 3-Clause BSD license (see the license specification on the *opensource.org* website[8]) and *Apache Thrift* is licensed under the Apache license 2.0. Therefore, both licenses meet the requirements. Both frameworks do not produce self-describing serialized objects, which means that for de-serialization descriptive schema files are needed on the system. In the concrete case of the MPM framework this means that the type information (schema) must be present on the sending and receiving peer for a serialized entry and also the contained serialized data object to be de-serializable. *Protobuf* stores that information in text files with extension *.proto*, *Thrift* does the same with extension *.thrift*. Both frameworks offer an own interface description language (IDL) for the type specifications in these files. The IDLs are slightly different, but in both systems a code generator for those files is used to produce code for the various target languages. For example, on an implementation of the MPM in Java for *Android*, code for serialization of objects could be generated by means of the IDL specification files, whereas on an Objective-C implementation for *iOS*, the code for serialization could be generated with the same files. Necessarily, the same specification files have to be known to both platforms to be able to interpret the exchanged data within the entries.

For more information on the *Thrift* IDL see the *Thrift* documentation at the the *Apache Thrift* website[9] and for information on how code can be generated out of that files for specific languages see the documentation about that on the same website[10]. Information on the *protobuf* IDL can be found in the tutorial at the *Google* developers website[11] and for information on how code can be generated out of that files for specific languages see

[5] `https://github.com/google/gson` accessed 01.2017
[6] `https://github.com/FasterXML/jackson` accessed 01.2017
[7] `https://www.apache.org/licenses/LICENSE-2.0` accessed 01.2017
[8] `https://opensource.org/licenses/BSD-3-Clause` accessed 01.2017
[9] `https://thrift.apache.org/docs/idl` accessed 01.2017
[10] `https://thrift.apache.org/` accessed 01.2017
[11] `https://developers.google.com/protocol-buffers/docs/proto3` accessed 01.2017

the documentation for different languages at the same website[12].

The main differences between *Apache Thrift* and *Google protobuf* are e.g. that *Thrift* supports exception definitions and *protobuf* not, whereas *protobuf* can handle extended types better. That means, if an incoming type has additional fields not known to the local *protobuf* schema, they are just ignored. So, extensions to types can be made easily without losing backwards compatibility. Moreover, definitions of remote procedure call (RPC) services for automatic generation of server and client skeletons are integrated into the *Apache Thrift* framework, whereas in *protobuf* they have to be added by a plugin. The most prominent *protobuf* compatible RPC implementation is *gRPC* (see the project website on *GitHub*[13]). For the proposed framework no RPC code generation and also no exception definitions are needed. Only entries and protocol-specific messages of the communication and encryption components are sent over the network. Because *Google protobuf* has a more extensive online documentation, performs slightly better than *Thrift* in the benchmarks and provides all needed functionality for the requirements, *protobuf* is chosen for the reference implementation. However, the system shall be designed so that this component is easily exchangeable by another framework like *Apache Thrift*.

5.4.3 Serialization process and needed registries

For non self-descriptive serialization formats like *Google protobuf*, schema files have to be present on every host that wants to serialize and de-serialize the respective data. Therefore, an application developer has to define all needed schema files at design time. Code has to be generated for the files on every platform supported by the application and the generated files have to be shipped together with the application. *Protobuf* needs type adapters to convert the de-serialized *protobuf Messages* to first-class Java objects that can be used in the system. To be able to find the right *protobuf* adapter for a specific Java object, a *protobuf* type adapter registry is needed. Additionally, to be able to lookup which *dataType* string from an entry (see Section 4.1) is bound to which Java class in the system, a *dataType* registry is needed. This registry is also necessary for the *Gson* implementation of the serialization component. For the *protobuf* serialization to work, all descriptive schema files have to be present and compiled to *protobuf Message* classes on each participating peer, all *protobuf* type adapters have to be present and registered in the system and all types of data objects that can be received by a peer have to be registered in the *dataType* registry. For further details on the serialization process with *protobuf* and on the needed registries see Section 6.3 in the implementation chapter.

Figure 5.7 depicts the activity diagram of a de-serialization process of an incoming entry byte stream at a receiving peer. One can clearly see the purpose of the two needed registries, the *dataType* registry that is needed by all implementations of serialization and the *protobuf* type adapter registry to convert from *protobuf Messages* to first-class objects, which is needed especially by the *protobuf* implementation. The serialization

[12] `https://developers.google.com/protocol-buffers/docs/tutorials` accessed 01.2017
[13] `https://github.com/grpc` accessed 01.2017

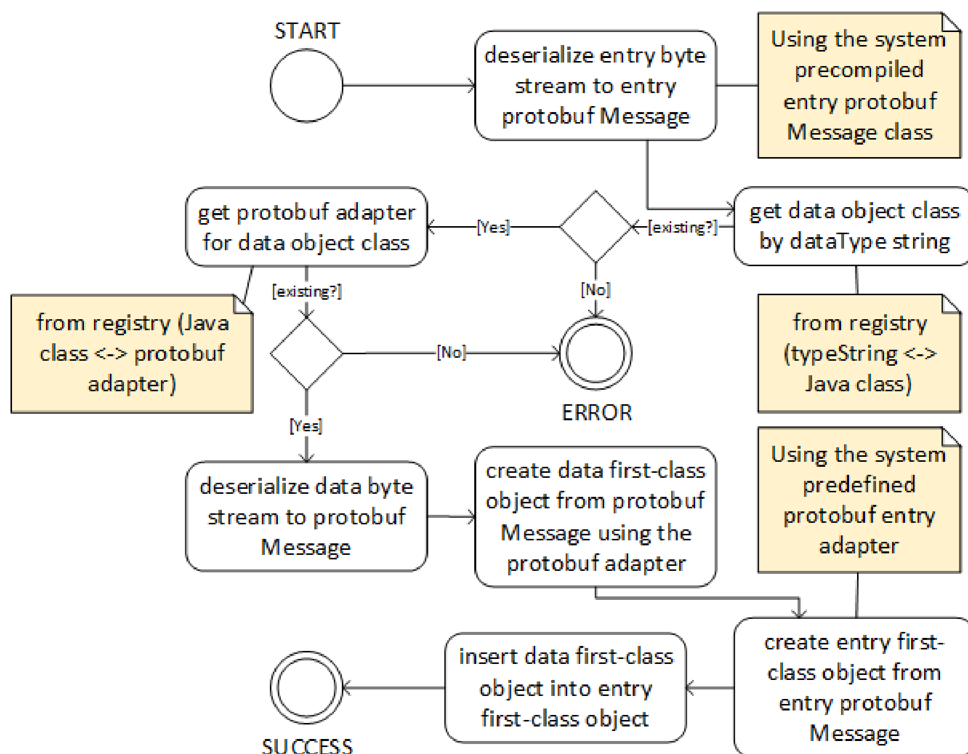


Figure 5.7: Activity diagram of de-serialization of entry and contained data object with *protobuf*

process works analogously, but in the other direction. To understand the depicted process it is additionally important to know that when deserializing the entry byte stream to a *protobuf* entry *Message* the object contains the byte stream of the entry data object in a field. Therefore the deserialization of the entry data byte stream has to happen in an own process. First the data byte stream has to be deserialized to a *protobuf* data *Message* and then converted to a data first-class object and is injected into the entry first-class object, which is deserialized separately.

5.4.4 Separation of entry serializer and data serializer

In the example of Figure 5.7, the serialization and de-serialization has been done with *protobuf* for the entry as well as for the contained data object. The framework is designed in a way that for an application it can be configured which implementation for serialization shall be used for entries and which one for the contained data objects. This separation of serializers allows that e.g. an application could serialize entries with their system properties using a JSON format and serialize the contained data object using *Google protobuf*. Participating remote peers have to be configured the same way or have to get information about the used serialization strategy to be compatible when communicating.

5.5 Security concept

In this section the security concept for the framework shall be discussed. As the framework is intended for internet-scale P2P collaboration and peers might be connected to the internet over public LANs, security is an essential aspect in the design of the framework. Based on the discussed security threats from the research done in Chapter 2, a concept of security countermeasures has been developed. It mainly consists of four aspects, an identity provider for peers joining the network, encryption between the relay server and peers, countermeasures against DoS attacks and optional E2E encryption between peers.

5.5.1 Identity concept

For MPM peers to be able to address each other in a WAN, they need to have an identity. In the reference implementation of the communication layer, the XMPP server acts as an identity provider in the network in addition to the relaying of messages. How peers can obtain an identity and join an MPM network has already been discussed in Section 5.3.2. XMPP as an identity provider is already a good countermeasure against e.g. fake bootstrapping attacks, like discussed in Section 2.4.2. Of course, the XMPP identity server role must be trusted to be effective as a secure identity provider.

5.5.2 Encryption on the transport layer

To avoid eavesdropping of exchanged data, encryption is used between a peer and the relay server. Out-of-the-box, the XMPP server offers encryption of network traffic using TLS. The option to use encryption on the XMPP communication layer can be switched on and off as a property of an XMPP connection. As TLS uses a public-key-infrastructure (PKI), the public key of the relay server has to be verifiable by the client peer. Therefore, the relay server should be equipped with a certificate signed by a certificate authority (CA). If the XMPP server holds a valid trusted certificate, peers in the network that are relaying messages over the server can be sure that the traffic is not exposed to anyone else than the server that is dedicated in the certificate.

5.5.3 Countermeasures against DoS and spamming

As can be derived from the presented research on security in Chapter 2, it is generally very hard or impossible to completely avoid DoS or DDoS attacks. Nevertheless, some means shall be provided to application developers and end-users of the MPM framework to take counteractions against possible DoS or spamming. These means are effective in the viewpoint of a peer in an MPM network. If a peer receives large amounts of unwanted messages from another peer, there shall be the possibility to block incoming traffic from that peer. Also, a white-list of peers shall be definable, which enumerates the peers that are allowed to contact a specific peer. Incoming messages from peers not contained in the list get blocked. Details on the implementation of these features are presented in Chapter 6. Although peers might block other peers or even only allow a specific set of

peers to contact them, the relay server might still be target of a DDoS attack. This is a general vulnerability of all publicly addressable hosts, like the relay server, and cannot be avoided easily.

5.5.4 End-to-end encryption

It is obvious that if all peers encrypt their traffic by using TLS with a certified XMPP relay server, still the server is able to see the transferred data in plain-text. In general, users of an application that operates in that way would have to trust the owner of the relay server to treat the transferred data confidentially and to not expose it to third parties. Also, further implementations of the communication layer in the MPM framework might not support encryption out-of-the-box. The framework therefore offers an interface to optionally encrypt the data between two communicating peers (end-to-end). If transferred data is E2E encrypted, also the relay server is not able to look into any data, it only relays the encrypted messages. There exist several protocols for E2E encryption. The reference implementation of the MPM framework implements one of these protocols, but is easily extendible by others.

Research on existing protocols

When doing research about existing protocols usable to achieve E2E encryption, several popular examples have been identified including pretty good privacy (PGP), with its open-source alternative specification named *OpenPGP* (see the official website[14]), and secure/multi-purpose internet mail extensions (S/MIME) (see the RFC[15]), both especially designed for email encryption. In contrast, off-the-record messaging (OTR) (see the official protocol specification[16]) and *Signal* (see the specification on the WhisperSystems website[17]) are popular protocols specifically for encrypting instant messages. Ermoshina et al. have done an overview of current E2E encryption protocols in [EMH16] dealing with all of them. There is an open-source implementation of the *GnuPG* suite, which supports PGP and S/MIME for the *Android* platform (see the project website on *GitHub*[18]). Also for *iOS* there exists a PGP implementation (see the project website on *GitHub*[19]) and it natively implements support for S/MIME messages in the development API (see the developer API reference on the *Apple* website[20]). Open-source *Signal* implementations for both *Android* and *iOS* can be found on *GitHub* (see the projects website[21]). One open-source implementation of the OTR protocol for Java and *Android* is also available

[14] `http://openpgp.org/` accessed 02.2017
[15] `https://tools.ietf.org/html/rfc5751` accessed 02.2017
[16] `https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html` accessed 02.2017
[17] `https://whispersystems.org/docs/` accessed 02.2017
[18] `https://github.com/guardianproject/gnupg-for-android` accessed 02.2017
[19] `https://github.com/krzyzanowskim/ObjectivePGP` accessed 02.2017
[20] `https://developer.apple.com/reference/security/cryptographic_message_syntax_services` accessed 02.2017
[21] `https://github.com/WhisperSystems` accessed 02.2017

on *GitHub* (see the project website[22]) and is called *Otr4j*. For *iOS* one can be found in the *GitHub* repository at the project website[23].

Encryption properties

There are some security aspects the framework can provide by using E2E encryption. The first one is confidentiality, that assures that no one else but the designated destination can read a sender's messages and furthermore authentication, which assures that the identity of the sender is really the originator of the message in terms of the user account. Moreover, there is the property of verification, which shall assure that the identity of the sender is really the person or service that it claims to be. In contrast to authentication, verification means to verify that the authenticated logical entity is the real-world entity that it claims to be. Last but not least, there is message integrity, which means more precisely checking whether a message has not been altered since it was created and if it has actually been sent by the declared sender.

To understand the following sections, some knowledge about asymmetric and symmetric encryption with the corresponding encryption primitives is assumed. Moreover, a reader should have some insights on the basics of the PKI and hash functions. A good source to get some detailed overview of cryptographic concepts and algorithms is [BF11].

PGP and S/MIME

Both PGP (see the RFC[24]) and S/MIME (see the RFC[25]) offer the property of confidentiality by encrypting the traffic with asymmetric or hybrid encryption mechanisms. In both concepts communicating partners each hold a public and a private key, where messages encrypted with the public key can only be decrypted with the private key of the designated participator. Public keys, like the name already says, are public to everyone and private keys are always confidential to the holder and shall never be exposed. The actual authentication is done implicitly by the asymmetric encryption system, because when encrypting with the public key of the destination, only the destination can read the contained data, because only the destination can decrypt with the private key. Additionally, cryptographic certificates are used in both protocols for verification that the public key is really owned by the claimed identity. These certificates can be signed by a chain of trusted CAs or following the web-of-trust concept, where the trust originates from many different participants signing the public key and thereby confirming the key-identity mapping. The web-of-trust concept is not supported by the S/MIME concept. Message authentication and integrity checking happens by digital signatures. Therefore, the sender calculates a unique hash-code of the plain text of the message and signs the hash with its private key and appends it to the message content. When the receiver decrypts the message with his private key, it can check integrity and authenticity

[22] `https://github.com/jitsi/otr4j` accessed 02.2017
[23] `https://github.com/ChatSecure/OTRKit` accessed 02.2017
[24] `https://tools.ietf.org/html/rfc4880` accessed 08.2017
[25] `https://tools.ietf.org/html/rfc5751` accessed 08.2017

by decrypting the hash-code with the sender's public key, calculate the hash-code on the decrypted plain text and compare the two hashes. These roughly explained concepts are supported by both PGP and S/MIME.

OTR

When Borisov, Goldberg and Brewer proposed the OTR protocol in 2004 [BGB04], they discussed the advantages of the protocol in comparison with e.g. PGP and S/MIME, especially for instant messaging. They highlight that those systems use long-lived encryption keys, which when compromised can be used to decrypt all future and also past traffic, if it has been stored. Furthermore, the digital signatures are a strong, jurisdictional proof of authorship of messages, which might not always be wanted, especially in social communications. Additionally to the properties offered by the described cryptographic systems, OTR has further interesting features. *Perfect forward secrecy* ensures that also if the private key of one participant gets compromised, past messages cannot be recovered, because every message is encrypted with a different short-term key that is derived from former keys. Message signatures in contrast to the other protocols have the additional property that they prove the authorship of the message to the designated partner, but not to any third party. More precisely, in contrast to the other protocols, OTR uses a different encryption scheme that provides confidentiality and authenticity for the communication partners internally, but lets a third party theoretically forge a message very easily to look like it has been sent by one of the partners. The concept is called malleable encryption and shall guarantee deniability of communication. The feature of deniability is offered in the first version of the framework in contrast to accountable communication for reasons of privacy. Developers might want to guarantee this to users in social applications where transferred data has to be absolutely confidential and communication has to be deniable (e.g. in countries with strict governmental cencorship). Nevertheless, another E2E encryption protocol that offers accountability of communication can be added to the framework by implementing the encryption module.

Signal

Signal (former *TextSecure*, see [CGCD+17] for more details), developed by *Open Whisper Systems*, is a relatively new protocol that can be seen as a successor of OTR. It is used by the famous *WhatsApp* chat application to E2E encrypt messages. It basically offers the same properties as the OTR protocol, but additionally also backward secrecy and asynchronous establishment of secure sessions, which means, that also future messages cannot be decrypted when a short-term secret key is compromised and for establishment of a secure channel not both communication partners have to be on-line. Nevertheless, although there are some additional features in the *Signal* protocol, an own additional server role is needed for the protocol to work. For detailed descriptions and comparisons of OTR and *Signal* see [FMB+16].

Protocol selection

Although a relay server role is already needed for communication over XMPP, there might be implementations of different communication protocols within the MPM framework, which do not need a server but still want to encrypt E2E. In the conceptual design of the framework's reference implementation it is not a requirement that peers can be off-line when establishing a secure channel. Therefore, for the reference implementation the OTR protocol is preferred over *Signal*. Also, in the latest version of the OTR protocol there exists an option to use the socialist millionaire problem (SMP) to verify that an identity is really the person or service it claims to be, like explained in more detail in [AG07]. *Signal* does not support this natively and when verifying public keys they have to be compared manually (like possible with *WhatsApp* by comparing bar-codes). Generally, the rigid concept of certificates for verification like in PGP and S/MIME is not applicable to dynamic, mobile networks. The additional features, like forward secrecy and deniable communication, that OTR and *Signal* offer, also seem to make sense for mobile and possibly social applications. Ultimately, because it does not need a server role and supports verification using the SMP without having to compare public keys manually, OTR is chosen as the first implementation of E2E encryption in the MPM. Still, the framework is designed that this component is easily exchangeable by another implementation like *Signal*.

Establishment of secure sessions with OTR

Figure 5.8 shows in detail how the cryptographic handshake of two peers (called *Bob* and *Alice*) looks like in OTR. It is based on the sign and mac (SIGMA) protocol. What happens first is a common Diffie-Hellman key exchange procedure. Both SIGMA and Diffie-Hellman are explained in further detail in [Kra03]. The advanced encryption standard (AES) is used to encrypt the traffic. From the shared secret created by the Diffie-Hellman exchange further shared secrets are created later on. By exchanging a message authentication code (MAC) and the long-term public keys, both clients can then authenticate each other.

Secure data exchange using OTR

If the secure OTR session has been established between the two authenticated, communicating peers, the confidential data exchange can happen with data messages from Alice and Bob. Figure 5.9 shows the components of such a data message. In every AES-encrypted message a new Diffie-Hellman key exchange is performed so that always a new shared secret can be created for future messages. Every message is signed with a MAC to guarantee integrity. The MAC key of the previous message is sent to the partner as well in plain text. This technique offers perfect forward secrecy, because for every message a different shared secret is generated and also deniable communication, because when publishing the previous key, a third party could forge a message and sign it by

[26] `http://matthewdgreen.tumblr.com/` accessed 02.2017 original from [BM06]

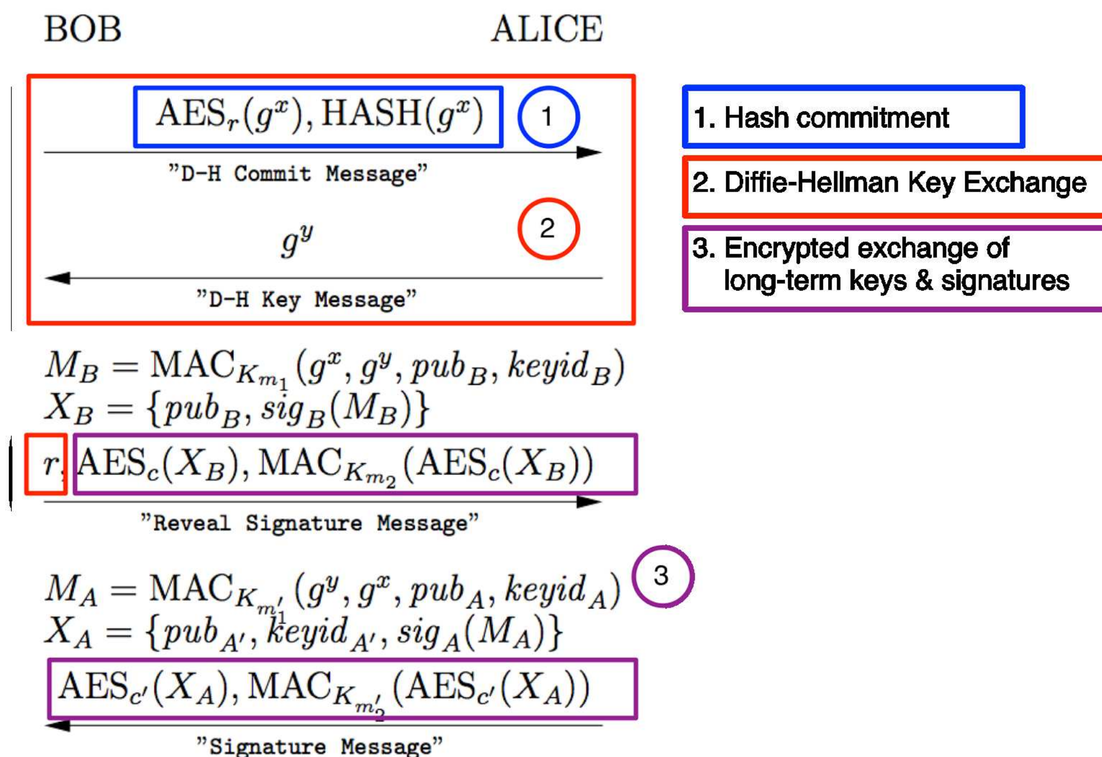


Figure 5.8: Sequence diagram of OTR handshake between Bob and Alice (see the post of *matthewdgreen* on *tumblr*[26])

creating a MAC with that key. The special streaming encryption-cypher used in this variant of AES allows changes to a specific character in the encrypted text resulting in only a specific change in the plain text and without scrambling the whole message, so the encrypted cypher text is actually malleable.

Verification of the secure session

An OTR session authenticates the communication partner and also authenticates every message for the receiver, but makes a prove of authorship to third parties impossible. Nevertheless, one can think of the situation where a communication partner holds a key and fakes an identity from the beginning of a conversation. Authentication in that case is no problem, because the key exchange with the attacker can happen legitimately. A solution to that is a comparison of a secret on another channel, which is possibly an insecure channel. Of course, one solution is to meet physically and compare the public key by hand, which is supported by e.g. *WhatsApp*. Nevertheless, if two partners already have a shared secret, the SMP offers a possibility for verification also on an

[27] `http://matthewdgreen.tumblr.com/` accessed 02.2017 original from [BM06]

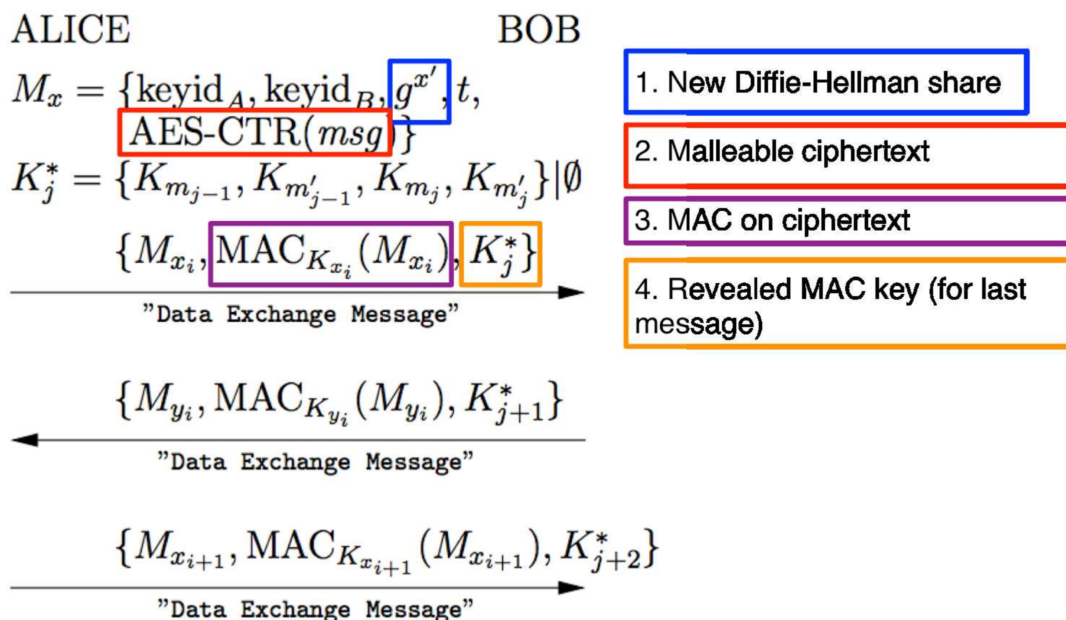


Figure 5.9: Sequence diagram of OTR data exchange and key publication between *Bob* and *Alice* (see the post of *matthewdgreen* on *tumblr*[27])

insecure channel and without having to meet personally. Basically, the protocol offers the opportunity to compare if two secrets x and y are equal ($x = y$) on an insecure channel without a third party being able to learn one of the secrets. The actual secret (answer) is never transmitted on the channel. Formal details on how this can be achieved are presented in [BST01]. The big advantage of the current OTR version 3 is that it supports the SMP for verification. By that means, e.g. a question "Where did we meet first?" could be asked to a partner and the answers (secrets) are compared by using SMP. Each side can then verify the secure session and now has a proof that the communication partner really has the identity that he or she claims to have.

5.6 Scalability concept

The following section is about the concept how scaling could work in an application that is based on the MPM framework. Basically, a pure P2P approach is scalable by design, but because communication shall be possible also from behind any NAT, a relay server role has been introduced to the communication layer reference implementation. Also, a registration server role is included to obtain an identity and a notification peer is used to send notifications to peers over the FCM framework (see Figure 5.4). Because these system defined components are necessary for operability of every participating peer, they need to be scalable if the number of peers in the network or the amount of network traffic

increases to ensure an appropriate performance of the application.

5.6.1 Scaling the relay server role

When choosing the implementation of the XMPP server, a main factor for the decision was that functionality to scale the server should be available. The *Openfire*[28] implementation of the XMPP server has the possibility for horizontal scaling by clustering the relay and identity server role. To achieve this, a special plug-in can be installed on the server named *Hazelcast Clustering Plug-in* (see the *Hazelcast* documentation on the *IgniteRealtime* website[29]). When the plug-in is installed on several XMPP servers, they can be joined together to a cluster by listing their addresses in a configuration file of the plug-in. The plug-in will then take care that all servers in the cluster will communicate with each other, e.g. to deliver messages to users connected to other servers within the cluster.

In the proposed architecture, a load balancer is needed to connect a user to one of the XMPP servers defined in the cluster. For that purpose, any load balancer that is able to do TCP load balancing can be used, but at least it should also support TLS, because this is necessary for encryption between peers and the relay server. For the scalability tests (see Section 8.1) the *HAProxy* high performance TCP/HTTP load balancer (see the *HAProxy* website[30]) is used. This load balancer needs to be configured with all the addresses from the single server instances in the cluster and also with the balancing algorithm to use for instance selection, e.g. *round robin* or *least connection* algorithms (see the *"<algorithm>"* argument in configuration documentation at the *HAProxy* website[31] for further details). When TLS is enabled, *HAProxy* e.g. offers TLS passthrough, that means it just opens a TCP tunnel to the chosen server and lets the client and server negotiate and handle the TLS traffic. Figure 5.10 shows the design of a scaled XMPP server role with 3 relay servers. When clients want to connect to the network, they have to connect to the load balancer, which will establish a connection with one of the servers in the relay cluster by using the specified balancing algorithm. After that, as long as connected, the peer will relay all communication within the MPM network over that XMPP server. The clustering software will manage to deliver messages to peers that are connected to different server instances. Also, *HAProxy* performs health checks on the clustered instances and does not establish connections to instances that are down at the moment, but only when they are on-line or become available again. When hosting an XMPP cluster the instances have to be located within the same LAN-based network infrastructure. That means they have to be within one locally switched network or in a virtual LAN, e.g. within a private or public cloud, like offered by *Amazon Virtual Private Cloud* (see the virtual private cloud (VPC) description on the *Amazon* website[32]).

[28] `https://www.igniterealtime.org/projects/openfire/documentation.jsp` accessed 08.2017
[29] `https://www.igniterealtime.org/projects/openfire/plugins/hazelcast/readme.html` accessed 02.2017
[30] `http://www.haproxy.org/` accessed 02.2017
[31] `http://www.haproxy.org/download/1.7/doc/configuration.txt` accessed 02.2017
[32] `https://aws.amazon.com/vpc/details/` accessed 02.2017

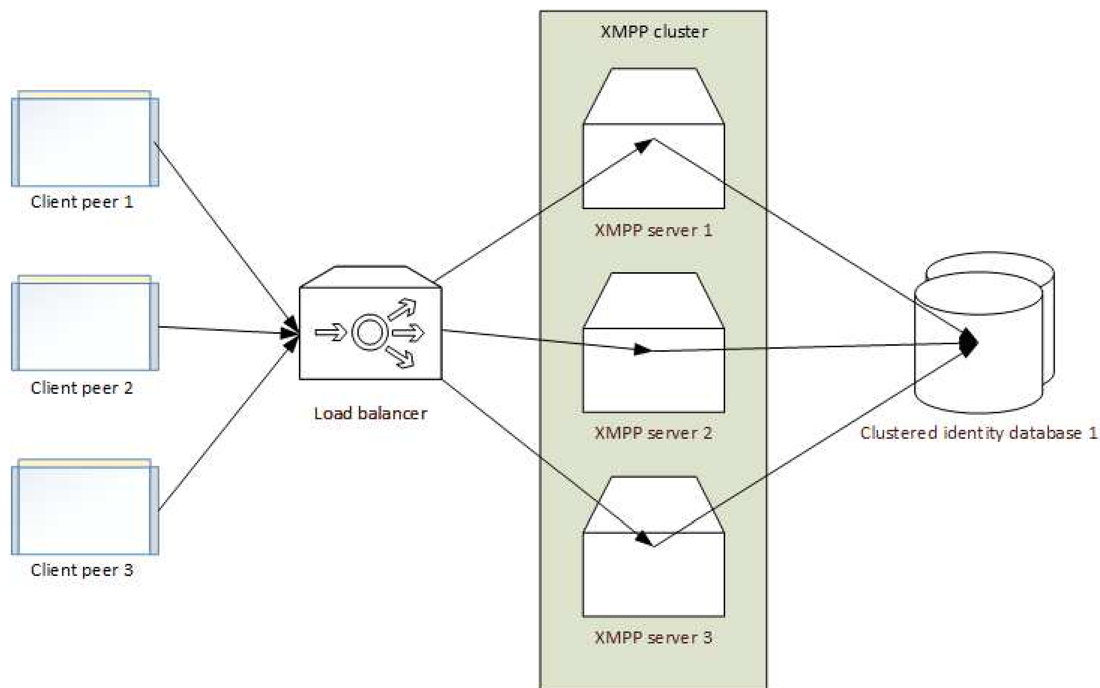


Figure 5.10: Architecture diagram of XMPP clustering for scalability of the MPM framework

All instances have to use the same identity database. The amount of database operations is negligible in comparison to the data exchange operations in the MPM network, e.g. database operations are necessary when a user registers or logs in to a server. Therefore, there is no particular focus on scaling the XMPP identity databases in the tests. Nevertheless, although scaling the database is probably not a big factor for performance, it is important for availability of the whole cluster. Therefore, the identity database should also be replicated in a database cluster, like shown in Figure 5.10. For example, *Amazon* offers cloud database clusters with *Amazon Aurora DB* (see the *Amazon Aurora DB* documentation on the *Amazon* website[33]). Clustering the database with e.g. *Amazon Aurora* provides good scaling for read operations used by the XMPP cluster and also increases availability of the database. *Amazon Aurora* distinguishes one primary instance, which is mirrored in milliseconds to the secondary instances. Access to the database cluster is completely transparent and available over standard client database connectors, looking like connecting to a single instance.

[33] `http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_GettingStarted.CreatingConnecting.Aurora.html` accessed 02.2017

5.6.2 Scaling the registration server

Peer registrations happens sparsely in comparison to the joining of and communication between existing peers. In the reference implementation of the MPM the registration server role is a web API that offers functionality to create a new identity in the application scope, like shown in Figure 5.5 via HTTP endpoints. This system component, as much as the identity database, does not play a role in the scalability tests. Nevertheless, the registration server has to be available to guarantee a new peer to be able to join the network. To guarantee availability and also some additional amount of performance, the registration server could be clustered as multiple web API instances. A good option to accomplish that is e.g. to use a clustered container service like *Amazon EC2 Container Service* (see the *Amazon EC2* developer guide at the *Amazon* website[34]). These containers act like light-weight virtual machines that can perform specific tasks like hosting the registration server API. *Amazon* offers out-of-the-box load balancers on such container clusters (see the blog post about load balancing on EC2 instances on the *Amazon* website[35]).

5.6.3 Scaling the notifier peer

Since the notifier is an important system peer in the scope of an MPM framework based application, needed by every application that wants to notify users about incoming entries, this centralized peer shall be scalable as well. The notifier peer, depending on the overall network traffic, might become a bottleneck, because a notification entry is sent to and processed by it for every other sent entry on the system. Therefore, depending on the application, it might be desirable to add more notifiers to the network, so that network and computational load can be distributed amongst them. The concept here is depending on the implementation of the communication layer. In the reference implementation with XMPP, multiple notifiers can just join the network with the same login name. With XMPP it is possible to set a globally unique identifier (GUID) for every connecting user. It is also possible to have multiple connections of a user at the same time, by specifying an XMPP resource (the mentioned GUID) (e.g. when connecting from different devices). In the MPM such a GUID is set on every different connection to the network. Now, when multiple notifier peers join the network, all other peers can request all the different joint notifier peers by getting all different resource GUIDs of them. This functionality is offered by the XMPP server. A main advantage is that new notifier peers can join the network dynamically and their resource GUID is delivered to peers that want to send notifications. To achieve load balancing, connecting peers select one currently available notifier peer at random.

Figure 5.11 shows a client peer logging in to an MPM network where two notifier peers are available, requesting their resource addressed from the XMPP server (1). After the

[34] `http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html` accessed 02.2017

[35] `https://aws.amazon.com/de/blogs/compute/microservice-delivery-with-amazon-ecs-and-application-load-balancers/` accessed 02.2017

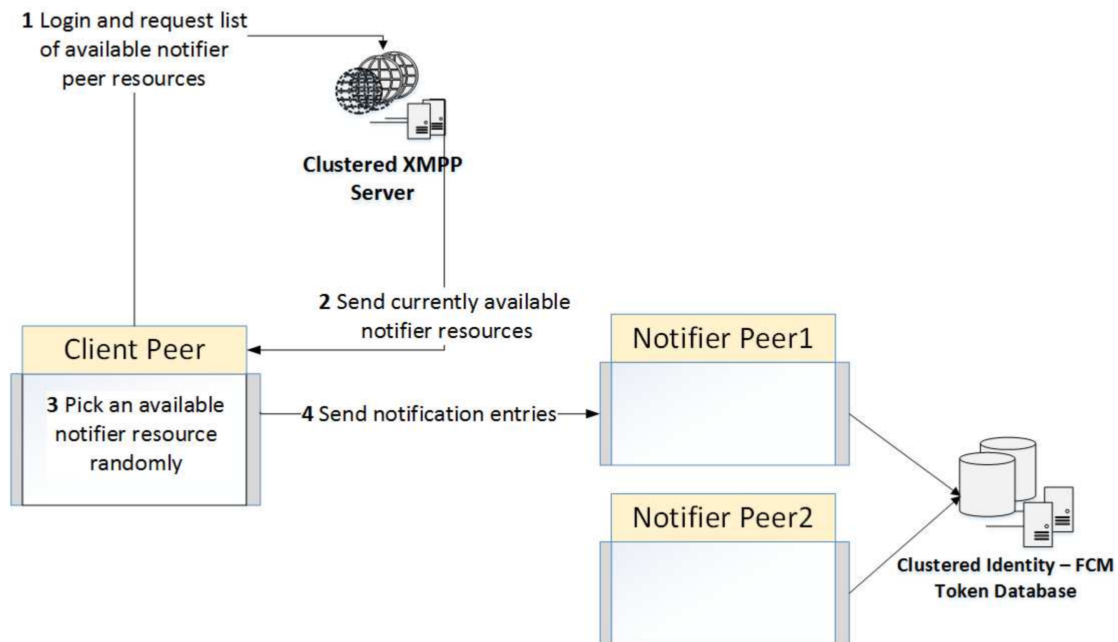


Figure 5.11: Diagram of a client peer randomly choosing one notifier amongst two available ones

server delivers the list of available notifiers (2), the client chooses one notifier at random (3). Notification entries are sent to this notifier peer in this session of the client peer (4). The graphic also shows that all notifier peers in an MPM use the same database for the mapping between identities and their FCM tokens. This database could be clustered analogously to the XMPP identity database to ensure availability of the notification service. Still, for scalability tests of the MPM this database is single instanced. Results of calls to this database are cached in the notifier peer's memory because when notifications are enabled there are many database queries.

CHAPTER 6

Implementation

After evaluating the background technology, establishing the requirements, clarifying the distribution of work and developing a system design for communication, serialization and security, this chapter shall expose some relevant details on the programming phase of the software engineering process. As the source code of the framework will be open, publicly available and documented, not so many code listings of the core model are presented in the paper. Important instructions on the configuration of system components and details on how to set up needed server roles, as well as insights on the most important interfaces of the exchangeable core components are discussed. As it has been described in Section 5.2.3, Peter Tillian focuses on the implementation details of the RTP with all its components, on the persistence and also on the integration with the *Android* platform. This section deals with my responsibilities in the framework development, which are the communication layer, serialization, scalability and security. Details on used external libraries and systems are provided.

6.1 Class overview

In this section a class overview of the implementation is given. Information is provided as unified modelling language (UML) class diagrams, only showing the most important packages with their included interfaces and classes. Also their inheritance structure and dependencies on each other are shown.

6.1.1 Communication and encryption components

The diagram in Figure 6.1 displays the most important classes and interfaces of the communication and encryption packages in the MPM reference implementation (package *at.ac.tuwien.mobilepeermodel*). The *IConnection* interface represents the exchangeable communication layer used in the MPM implementation. The *XmppConnection* class is

the provided implementation of this interface for the first version of the MPM framework. It is using the *XMPPTCPConnection* from the *Smack* XMPP client library for Java (see the Smack documentation website[1]), which is the default implementation in this package for connections to an XMPP server. *Smack* is an open-source XMPP client library that is in ongoing development and is provided by *Ignite Realtime*, who also publish the *Openfire* XMPP server.

The encryption package is integrated with the communication layer, as it is extending the *IConnection* interface to a *ISecureConnection* interface, which additionally to the communication functionality offers all the functionality to perform the E2E encryption. The *OtrSecureConnection* class is the delivered implementation of this interface for the first version of the framework. It is dependent on the *OtrEngineHostImpl*, which implements the *OtrEngineHost* interface of the *Otr4j* library (see Section 5.5.4). This class is responsible for managing all OTR encrypted sessions to other peers in the network over a secure connection. That means, when sending an entry to another peer over an *OtrSecureConnection*, a secure session gets instantiated on the *OtrEngineHost* with the peer in the *dest* property of the entry. Moreover, the *OtrSecureConnection* is using an underlying *IConnection* to send the encrypted data, e.g. an *XmppConnection* or another implementation. The actual implementation of the *Session* class, which represents such a secure OTR session, is not shown in the diagram and done by the *Otr4J* library. Furthermore, to store all needed keys used by the OTR protocol (see Section 5.5.4), the *OtrKeyManagerStore* interface of the *Otr4j* library is implemented by the encryption package of the MPM framework (*OtrKeyManagerStoreImpl* class) and used by the *OtrEngineHostImpl* class.

6.1.2 Serialization component

As has been stated in Section 5.4.4, serialization shall be configurable separately for entries and the data object probably contained in an entry. Also, a reference implementation for both entry serializer and data object serializer shall be provided with *Google Gson* and *Google protocol buffers* (see Section 5.4.2). Figure 6.2 shows the most important interfaces and classes of the serialization packages for data object serialization (package *at.ac.tuwien.mobilepeermodel.data.serialization*) as well as for entry serialization (package *at.ac.tuwien.mobilepeermodel.entry.serialization*).

All implementations of a data object serializer have to implement the interface *IDataSerializer*. In the reference implementation, a *GsonDataSerializer* and a *ProtobufDataSerializer* that implement this interface are provided. As has been discussed in Section 5.4.3, every implementation of an *IDataSerializer* is dependent on a data type registry (class *DataTypeRegistry*) with mappings from the entry *type* property to objects of the MPM framework.

[1] `https://www.igniterealtime.org/projects/smack/documentation.jsp` accessed 07.2017

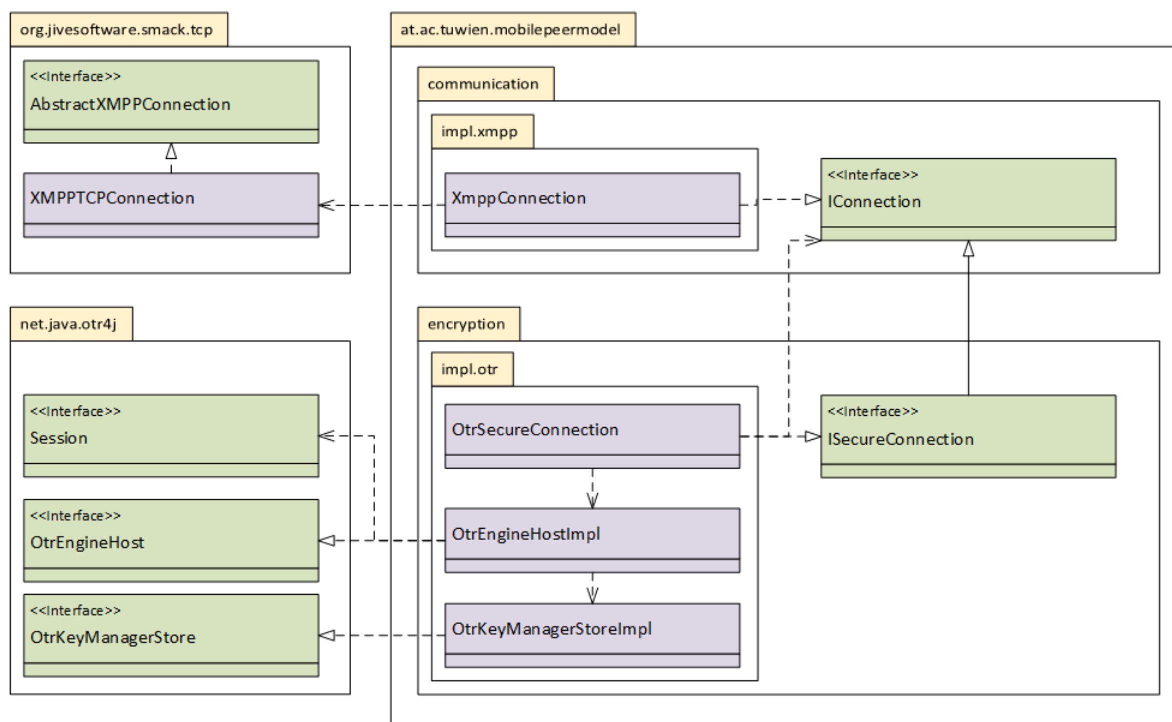


Figure 6.1: Main classes of communication and encryption of the MPM framework with their dependencies (packages yellow, interfaces green, classes gray)

Analogously to the data serializer, every implementation of an entry serializer has to implement the interface *IEntrySerializer*. Also here, a *GsonEntrySerializer* and a *ProtobufEntrySerializer* are provided. As has also already been described, a registry for *protobuf* type adapters has to be implemented (class *ProtobufTypeAdapterRegistry*), which holds all *protobuf* type adapters (package *adapters*) defined by the application developer. The *protobuf* data serializer needs this registry in order to convert *protobuf* messages (imported from *com.google.protobuf.Message*) to first-class data objects of the MPM framework. While *protobuf* adapters for data objects have to be defined by the application developer, the *ProtobufEntryTypeAdapter*, used to serialize entries with *protobuf*, is already implemented in the system and used by the *ProtobufEntrySerializer* class.

Google Gson does not need adapters to convert from objects to JSON strings by default, because it uses matching of field names in the objects to property names in the JSON string. As you can see in the diagram, the only thing needed by the *Gson* data and entry serializer classes is the implementation of the *Google Gson* serializer from the *com.google.gson* package.

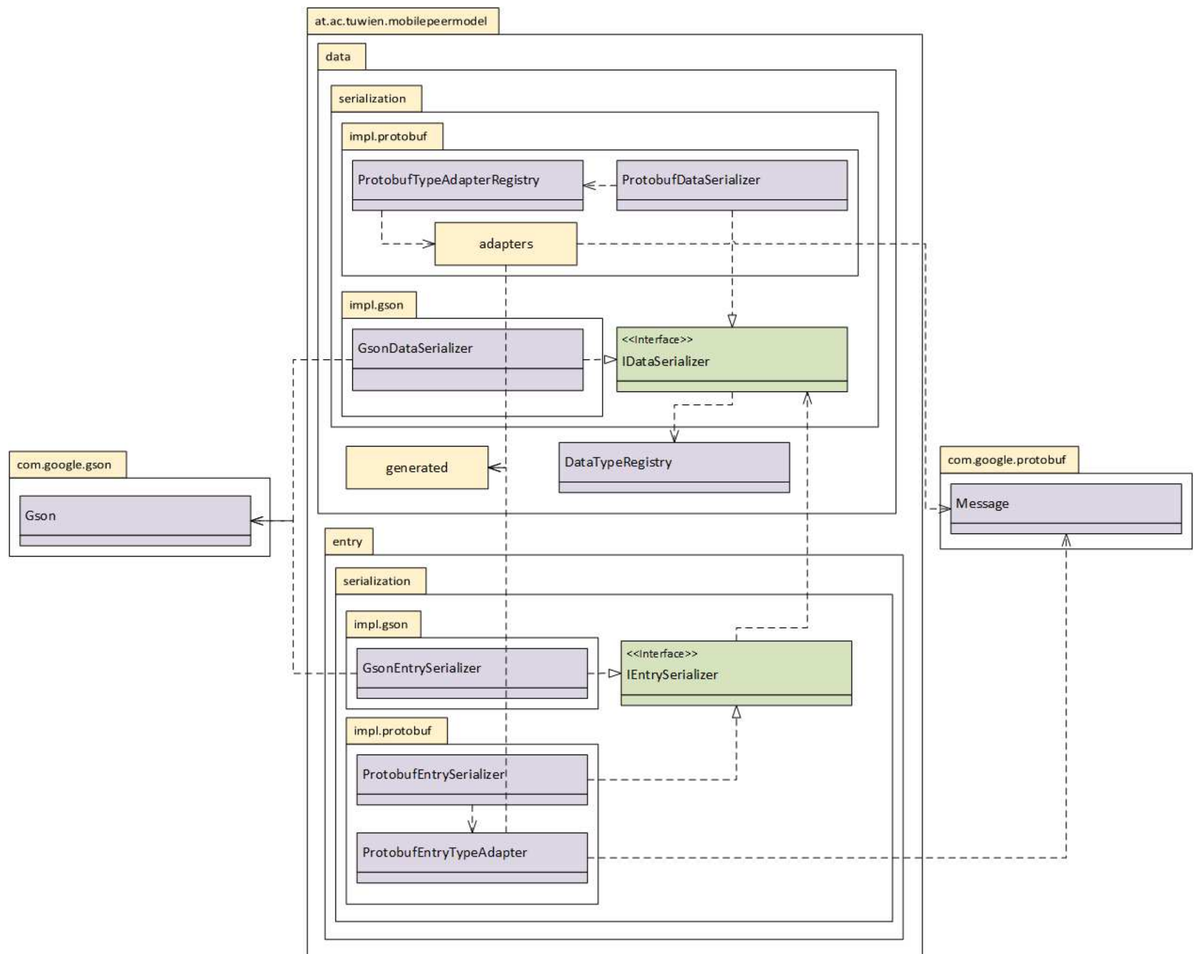


Figure 6.2: Main classes of the serialization component of the MPM framework with dependencies (packages yellow, interfaces green, classes gray)

6.1.3 Other components

There are further important components of the core MPM framework. These are the *at.ac.tuwien.mobilepeermodel.peer* package, containing the implementation of the RTP with its local peers and system peers, as well as the *at.ac.tuwien.mobilepeermodel.persistence* package, which manages persistence of entries and data-loss prevention. The overview of these parts and also the design of the *Android* service integrating an MPM RTP, which is delivered together with the core framework, is done by Peter Tillian [Til17], because these packages are within his responsibilities.

6.2 Communication and identity management

This section explains the important components of the communication layer. The first one is the *IConnection* interface, which is used by the RTP to connect to the network and has to be implemented by an MPM communication component. In the reference implementation this is the *XmppConnection*. Here the usage of the *Smack* XMPP library is shown. Furthermore, some installation and configuration instructions on how to set-up and configure the used XMPP server *Openfire* are presented. Also, some insights on the implementation of the notifier peer are given.

6.2.1 *IConnection* interface with its *XmppConnection* implementation

The communication component of the MPM framework is an important and exchangeable part. It offers all functionality to connect peers to an MPM network and enables sending entries to other peers. Figure 6.3 shows the *IConnection* interface, exposing all methods needed to connect, disconnect, send and receive entries to and from other peers. The method *addMessageListener* is used to register an *EntryMessageProcessor* on the connection, which has a single method *processEntry(IEntry)* that is called when a new entry arrives from a remote peer. There exists a method *setAllowedSenders*, where a list of peer URIs can be provided to the connection. The *EntryMessageProcessor* then only is called on incoming entries from that specific list of remote peers. By that means, some sort of white-listing of peers allowed to send entries to the local RTP can be achieved, which is a countermeasure to e.g. spamming. Moreover, the method *blockUnblockIdentity* can be used to block or unblock a specific remote peer, identified by a provided *IIdentity* object. In this case this is the whole MPM RTP identified by an XMPP host-name. It is also a countermeasure against spamming and DoS attacks, like discussed in Section 5.5.3. Last but not least, there is also the possibility to send not only entries (method *send(IEntry)*), but also *String* messages to other peers (method *send(String, IIdentity)*). This is needed by overlying E2E encrypted connections that are using this connection. An overlying *ISecureConnection* can be registered with the method *setSecuredConnection* in the underlying *IConnection*. This is necessary because when traffic is encrypted the *IConnection* that receives an incoming encrypted entry or an encryption control message has to hand it over to the *ISecureConnection* for further processing. A secure connection might need some kind of preceding handshake or conversation like Diffie-Hellman key exchange, where *String* messages have to be exchanged. These are used to establish a secure session and no coordination logic is needed for those messages, so it is more efficient to send them as plain strings.

The presented interface has been designed to be as general as possible, implementable also by other protocols than XMPP over TCP, which is used in the reference implementation. E.g., a pure socket based implementation for a LAN based MPM network without additional identity management or authentication might as well be realized as a bluetooth-based network. In the first implementation the *XMPPTCPConnection*

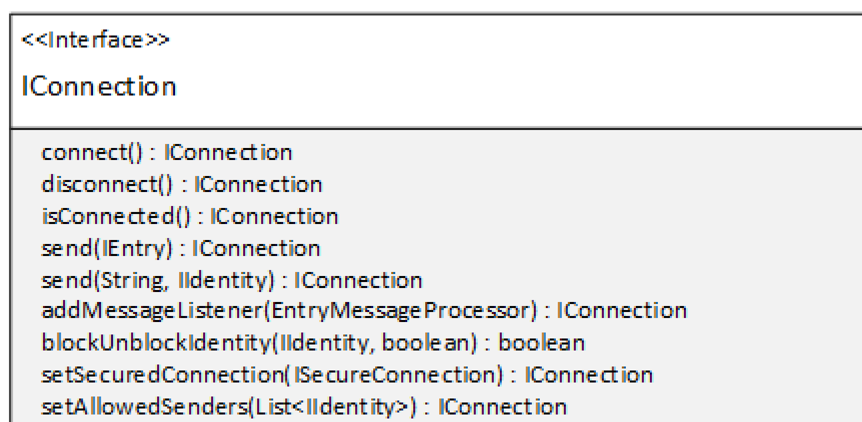


Figure 6.3: The *IConnection* interface

(see the *Smack* documentation [2]) of the open-source XMPP library *Smack* is used. Besides other functionality, this class exposes some important methods to be used by the *XmppConnection*, which is implementing the *IConnection* interface:

- *connect()* - used to establish a connection with the XMPP server
- *disconnect()* - used to disconnect from the XMPP server
- *login()* - used to authenticate the client on the XMPP server
- *sendStanza(Stanza)* - used to send XMPP stanzas (stanzas are XMPP messages, see [Til17] for further details)
- *addAsyncStanzaListener(StanzaListener)* - used to register a listener that processes incoming stanzas

6.2.2 Configuration of the XMPP communication component

Basically, the methods from the *Smack* implementation, which have been enumerated above, provide the main functionality used by the implementation of the *IConnection* interface. Nevertheless, for the establishment of an XMPP connection with a server, further properties are important. These include the address where the XMPP server or server cluster is reachable, the TCP port where the XMPP server role is running on that host, as well as the service name for the application on that XMPP server, representing the name-space where user-names have to be unique. These services are represented as domain specifications (<domain>.<top-level-domain>) and can also be used for resolution with the Domain Name System (DNS). For further details on the XMPP service definition see [Til17]. Furthermore, it can be configured whether the

[2] `http://download.igniterealtime.org/smack/docs/latest/javadoc/` accessed 02.2017

traffic with the server shall be encrypted using TLS, if additional compression shall be done on the XMPP messages (provided by the XMPP protocol) and if log messages shall be printed for debugging purposes. All these properties have default values specified in the file *xmpp-default.properties* in the resource folder of the MPM core framework. An application developer can override them by specifying them in the resource folder of the developed application project in the file *xmpp.properties* with the same names for the properties. The MPM framework automatically detects that and use the overwritten values.

6.2.3 Installing and configuring the XMPP server

Like already mentioned, the *Openfire* XMPP server implementation and the *Smack* XMPP client library are used. Both are licensed under the open-source *Apache 2.0* license and thus are compatible with the licensing requirements. While the usage of the *Smack* library has already been explained, this section gives some information on how to set-up and configure the *Openfire* server. Insights on how clustering of the server can be done is provided in the following chapter.

The *Openfire* server can be downloaded for *Windows*, *Linux* and *Mac OS* from the *Ignite Realtime* website[3], the source code is also available from there. Detailed installation instructions are available on the website's installation guide[4]. The server also needs a database for storing the XMPP identities. Details on how to set up the database can be found at the database installation guide[5]. After the installation, the server can be configured in a web-based administrator console. There also the used ports are configured, which also have to be activated on the server's firewall.

6.2.4 Setup of the notification infrastructure

The concept of notifications, like proposed in Section 5.3.3, specifies the role of a notifier peer in the system, which pushes notifications to other peers. The notifier peer is implemented as a self-contained peer in the system and is bootstrapped as a usual MPM peer. The only difference is that this special peer has a fixed address in any MPM network. This is defined in a configuration file and compiled for every peer in an MPM application. The implementation, including the sources, of the notifier is delivered accompanying the MPM core model framework in an own open-source project.

Basically, the notifier peer runs in an MPM RTP, like every other peer in an MPM network, as shown in Figure 5.3. This predefined peer has only one local peer with exactly two wirings containing two services. The two wirings exactly represent the two responsibilities of the notifier peer. These are, like explained in Figure 5.4, the storing of users' FCM

[3] `https://www.igniterealtime.org/downloads/index.jsp#openfire` accessed 02.2017

[4] `http://download.igniterealtime.org/openfire/docs/latest/documentation/install-guide.html#database` accessed 02.2017

[5] `http://download.igniterealtime.org/openfire/docs/latest/documentation/database.html` accessed 02.2017

tokens and the sending of notifications to peers. These wirings are triggered when either an entry with the type *TOKENUPDATE_TYPE* or *NOTIFICATION_TYPE*, both system-predefined, are sent to the notifier peer. The entries, accordingly, must contain a data object, which is simply a string and is then either the FCM token or the host-name of the peer that shall be notified.

After arrival of a token update entry, the token update service of the notifier peer stores the new token of the user, which is represented in the *from* property of the entry, in a database. This mapping-database has to be set up by the infrastructure provider. Then, a *TOKENUPDATE_ACK* entry is sent back to the sender to notify it about the successful storing of the mapping.

When a notification entry has arrived, the notification service of the wiring looks up the FCM token of the peer to be notified in the database and sends a request to the Google FCM servers containing that token, which then notify the remote peer.

The notifier is implemented as a *Spring Boot* (see the *Spring Boot* on-line documentation[6]) application with a dependency to *Hibernate* (see the *Hibernate* website[7]) for object-relational mapping of Java objects to the database.

There are only a few steps that an application developer has to perform to set up the notification infrastructure. First, the notifier code, provided as an own project beside of the core MPM framework, has to be downloaded. In the *application.properties* file of the project (resource folder), the database settings have to be configured (e.g. host and port of the database and the database user credentials). *Hibernate* creates the database schema automatically when the notifier is started. Secondly, on the XMPP server, a peer identity (username and password) has to be created for the notifier peer (e.g. over the *Openfire* administrator console). The default host and peer name of the notifier are *notifier_peer* and *notifier*. They are defined in the *general_default.properties* file in the resource folder of the MPM core framework and can be overridden in a *general.properties* file within an application that is dependent on the MPM framework. This file has to be created by the application developer in the project if default properties shall be overwritten. The password for the notifier to join the network is configured in the *mobilepeermodel.properties* file of the notifier project and has to be set to the chosen password. The last step of the setup is to create an FCM project on the Google FCM servers and configure an FCM server key and server URL in the *firebase.properties* file of the notifier project. As FCM in the reference implementation works directly together with the *Android* platform, it is in the scope of the work of Peter Tillian [Til17]. Details on how to create an FCM project and get the server key and URL are explained there.

When the explained steps have been taken, the notifier application can be executed on a host by just starting the *Spring Boot* application. This host does not have to be publicly available, because an MPM peer using XMPP, like the notifier, can be contacted from everywhere if it is connected to the same XMPP network. Still, the

[6] `https://projects.spring.io/spring-boot/` accessed 03.2017
[7] `http://hibernate.org/orm/` accessed 03.2017

host should have a stable internet connection. By default, a notification entry is sent to the notifier peer whenever a peer sends an entry to another peer. To switch off this functionality, an application developer can override the *general_default.properties* file in the file *general.properties* in the sending peer and set the value of *sendNotifications* to *false* (default is *true*).

Avoiding the massive overhead of keeping a secure session with every participating peer, all sessions with the notifier peer are not E2E encrypted. Anyway, the only data that is sent to the notifier are host-names of peers and FCM tokens, so no payload could be eavesdropped. Moreover, it is recommended to have several redundant notifiers running on different hosts to guarantee some amount of performance and availability in a productive application. See Section 5.6.3 on how to scale the notifier peer.

6.3 Serialization

A concept for the serialization of entries and data objects within the MPM framework has been developed in Section 5.4. In this section implementation specific details on the serialization process with *protobuf*, which needs schema files and type adapters, are provided. Also, important interfaces of the serialization component are described for framework developers to be able to develop new implementations of this component besides the provided *Google protobuf* and *Google Gson* implementation. Furthermore, it is shown how an application developer can configure the serialization component.

6.3.1 Details on the serialization process with *protobuf*

In the framework's reference implementation, *protobuf* generates Java *protobuf Message* classes, which are immutable and can be instantiated by using generated *Builder* classes. Listing 6.1 shows an example of the *protobuf* IDL definition of a *Person* type with a string property *name*. After running the Java code generator for *protobuf*, a *Person.java* class is generated containing a *Person.Builder* subclass, partly shown in Listing 6.2. The *Builder* subclass contains getter and setter methods for the *Person*. The *Builder* then also exposes an interface Builder.build() (not shown in the listing), which instantiates the object and returns the immutable *Person* object shown partly in Listing 6.3. It only contains getter methods.

```
1 //proto IDL type definition for a Person (person.proto)
2 message Person {
3  string name = 1;
4 }
```

Listing 6.1: Example of a *protobuf* definition of a *Person* object (*.proto* file)

```
(...)
//part of the generated code inside the Person.Builder class for
    Java (Person.java)
public boolean hasName();
public java.lang.String getName();
public Builder setName(String value);
public Builder clearName();
```

Listing 6.2: Example of a protobuf generated Builder subclass of a Person object (.java file)

```
(...)
//part of the generated code inside the Person class for Java
    (Person.java)
public boolean hasName();
public String getName();
```

Listing 6.3: Example of a protobuf generated Message class of a Person object (.java file)

Those generated Java files can get quite complex and are immutable. Therefore, they are not suitable as first-class objects to treat in the runtime of a peer. Furthermore, in that case the whole framework would depend on the *protobuf* framework and the serialization component would not be exchangeable. Suitable representations of those objects are needed and have to be written by the application developer. For instance, in this case the developer has to write the *Person.java* definition again as a usual Java class and has to provide an adapter that can convert from a *protobuf Message* to that Java first-class instance. Entry objects always have the same properties and therefore the adapter for entry objects is predefined in the framework. The developer has to provide only the type adapters for the contained data objects.

The serialization component uses a type adapter registry to look up the right type adapter of a *Message* to be converted to the first-class object and vice versa. The *protobuf Message* class exposes an interface to convert to and from a byte stream that can be sent over the network. The receiving peer knows that an incoming *Message* is an entry *protobuf Message* so it can always use the predefined entry protobuf type adapter to de-serialize the received byte stream to an entry. Every entry has a *dataType* string property that defines the type of the contained data object. So, the second registry besides the *protobuf* adapter registry is the *dataType* registry that maps *dataType* strings to first-class objects. By looking up the Java class for a *dataType* string in the *dataType* registry, the de-serializer can now also lookup the right *protobuf* adapter for that class and de-serialize also the contained custom data object. A precondition for this whole process to work is that the developer has implemented and registered the Java classes for the data objects and has implemented and registered a *protobuf* type adapter for each of them. All peers that want to participate must have these types and adapters registered.

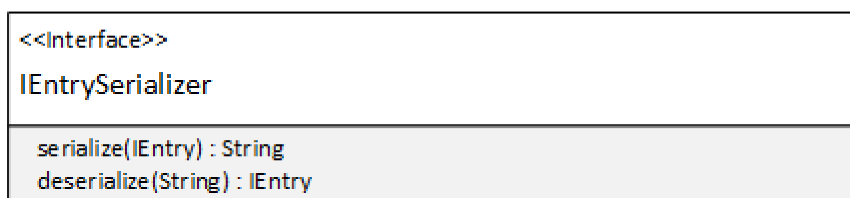


Figure 6.4: The *IEntrySerializer* interface

6.3.2 IEntrySerializer interface with its implementations

Figure 6.4 shows the very compact interface of an entry serializer. It only exposes two methods *serialize(IEntry)*, which converts a provided entry to a *String*, and *deserialize(String)*, which converts a *String* to an entry. An implementation of the interface has to take care that all the properties defined for an entry in the MPM framework are serialized and deserialized properly (see Section 5.1.2). Entries are intended to be serialized to *base64* encoded *Strings*, so that they can be sent properly over the internet. Both the *Gson* and *protobuf* implementations of the *IEntrySerializer* interface finally encode the outcome of serialization to the *base64* format.

As the intention of the system design is to instantiate only one entry serializer for the whole system, there exists a class *EntrySerializer* in the framework, which follows the *Singleton* programming pattern and implements the *IEntrySerializer* interface. It instantiates the entry serializer implementation that has been configured by the application programmer in the *general.properties* file of the application projects resource folder. Therefore, it also follows the *Proxy* programming pattern. If no file has been created or the value of the *entrySerializerImplementation* property has not been provided as a full valid class name of an entry serializer implementation, the default implementation *Google Gson* is used to serialize entries.

6.3.3 IDataSerializer interface with its implementations

The *IDataSerializer*, shown in Figure 6.5, is used by an implementation of the *IEntrySerializer* and is necessary to serialize the possibly contained data object of the entry, which can be any serializable object. The method *serialize(Serializable)* is used to serialize these first-class objects to be embedded in a serialized entry object by the entry serializer. In case of the *protobuf* implementation, a byte array is returned by the method to be further processed by the entry serializer when called on the data object, when using *Gson* a *String* is returned. Data objects are not *base64* encoded, because this is done by the entry serializer in the end. The *deserialize* method of the interface represents the inverse function. The important thing is that besides the serialized representation of the object, also the instance class type of the object has to be provided to the data serializer to know which class should be instantiated. This is necessary because *Gson* and also *protobuf* do not serialize to fully self-describing formats, so no type information is directly included. The class type is provided as a parameter to the *deserialize* method, when called by the

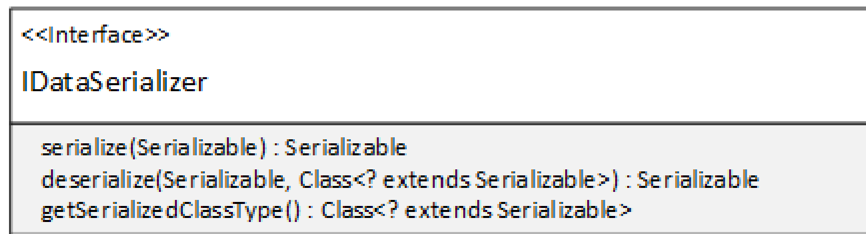


Figure 6.5: The *IDataSerializer* interface

deserialize method of the entry serializer on the data object. It is requested using the *dataType* property of the entry, which is a string and can be mapped to the instance class using the *dataType* registry (see Section 5.4.3). Finally, the entry serializer also needs to know which type of object the data serializer returns when serializing. For now, the possibilities are only *Strings* (*Gson*) and byte arrays (*protobuf*), but to be flexible for future implementations, the *getSerializedClassType* method shall provide this information when called. The entry serializer can then act depending on what is delivered by the data serializer.

Analogously to the entry serializer, also the data serializer is a *Singleton Proxy* class, which instantiates the configured full class name of the *IDataSerializer* implementation in the *general.properties* file. The default is again the *Gson* implementation.

6.3.4 Protobuf type adapters

In Section 5.4.3 it has been explained that *protobuf* serialization needs type adapters to convert *protobuf Messages* to first-class instance objects and the other way around. This means that for every data object that shall be sent within an entry to another peer, a *protobuf* type adapter has to be defined by the application developer. The class diagram in Figure 6.6 shows the abstract class *ProtobufTypeAdapter*, which has to be extended by the specific type adapters of the application developer. As can be seen, the abstract class has type parameters for the serializable first-class data type (S) and the *protobuf Message* class type (M), which has been generated using the *.proto* file defined by the application developer, like explained in Section 5.4.3. The methods *toProtobufMessage(S)* and *fromProtobufMessage(M)* have to be implemented by the concrete type adapters and use the *protobuf Builder* and *Messages* classes, like also explained in Section 5.4.3, to convert from instance objects to *protobuf Messages* and vice versa. There exist some predefined type adapters in the core framework to e.g. convert simple string data objects.

6.3.5 DataType and protobuf adapter registries

In Section 5.4.3 it has been shown that two different registries are needed in the system. One general *dataType* registry to map from *dataType* entry properties to Java instance classes and one *protobuf* type adapter registry to map from *protobuf Messages* to Java instance classes, needed only if *protobuf* is configured for data object serialization. Besides

```
<<abstract>>
ProtobufTypeAdapter<S extends Serializable, M extends Message>
--------------------------------------------------------------
ProtobufTypeAdapter(Class<S extends Serializable>, Class<M extends Message>)
<<abstract>> toProtobufMessage(S) : M
<<abstract>> fromProtobufMessage(M) : S
getInstanceClass() : Class<S>
getProtoMessageClass : Class<M>
```

Figure 6.6: The abstract class *ProtobufTypeAdapter*

some system defined data objects and *protobuf* adapters, it is the responsibility of the application developer to register all needed mappings in the *dataType* registry and if *protobuf* is configured to implement the needed adapters and register them.

The dataType registry (class *DataTypeRegistry*) is a simple class that exposes one important static method *putMapping(String, Class<? extends Serializable>)*, which adds a mapping of a *dataType* property of an entry to the corresponding Java instance class. So, e.g., if a *String* is sent over the network inside the *data* property of an entry and the definition for the data object type is e.g. the name "string", then there has to be a registration in the *dataType* registry with *putMapping("string", String.class)*. Some primitive *dataTypes* like *boolean*, *String*, *Long*, etc. have already been registered, also the system defined classes like *Entry.class* and *ExceptionEntry.class*, as well as *byte* arrays, have been added. The *dataType* registry is used by the implementations of the *IDataSerializer* interface to get information to which class a data object inside an incoming entry shall be deserialized to. One could argue that it would probably be a good idea to just use the Java class name for the *dataType* property, de-serialize to an object of the class with that full name (including packages) and thereby get rid of this registry. Nevertheless, such an approach would not be inter-operable with e.g. an *iOS* implementation and also always the full name of a Java class would have to be transferred on the network with every entry (instead of a short name). Therefore, the decision has been made to introduce this registry.

The *protobuf* type adapter registry (class *ProtobufTypeAdapterRegistry*) is used to register type adapters to convert from *protobuf Messages* to Java instance classes. The class exposes one important static method *registerAdapter(Class<? extends ProtobufTypeAdapter>)*, which can be used to register type adapters extending the abstract *ProtobufTypeAdapter* class in the system. This is necessary to be used by the *protobuf* implementation of the *IDataSerializer* interface. As implementations of such adapters have type parameters for the *protobuf Message* class and the Java instance class, the mapping between them can be automatically done by the *protobuf* type adapter registry, when registering an adapter.

6.4 Security

The security concept for the framework, like proposed in Section 5.5, is based on four important aspects:

- Identity concept - realized with XMPP identities in the reference implementation
- Encryption on the transport layer - realized with XMPP using TLS functionality
- Countermeasures against DoS and spamming - realized with white-listing and blocking of identities
- E2E encryption - realized with the OTR protocol implementation in the library *Otr4j*

This section provides insights in the implementation details that might be relevant for application or framework developers of the MPM framework.

6.4.1 Identity concept

Identities in the reference implementation of the MPM framework use XMPP as identity provider (see Section 5.5.1). Identities are obtainable for joining peers by using a trusted registration server role, like shown in Figure 5.4. Communication with this server can be encrypted using TLS and certified using a PKI certificate. As the XMPP server acts as an identity provider, the only action that an application developer has to perform here is to set up the registration server role and obtain a certificate for the server if wanted. The source code of an example registration server is delivered as an own project accompanying the MPM core framework code and is a *Spring Boot* (see the *Spring Boot* project website[8]) application that offers a web API to perform user registrations on the XMPP server. Details on the registration server role implementation and on how to install the server are delivered by Peter Tillian [Til17].

6.4.2 Encryption on the transport layer

XMPP, as the chosen communication protocol, offers client to server encryption out-of-the-box by making it possible to enable TLS on a connection. By default, TLS is turned off in the system, but can be enabled by the application developer by setting the property *security=enabled* in the file *xmpp.properties* in the resource folder of the project. Additionally, in the file a *TrustManager* and a *HostnameVerifier* can be configured.

The *TrustManager* is an implementation of the *javax.net.ssl.X509TrustManager* interface and is responsible to check if the certificate published by the server is valid and has not yet expired. This is done by the *isServerTrusted* method of the interface.

[8] `https://projects.spring.io/spring-boot/` accessed 03.2017

The *HostnameVerifier* is responsible to verify the host name of the server provided in the context of the TLS session with the configured XMPP service domain. It is an implementation of the *javax.net.ssl.HostnameVerifier* interface and exposes one method *verify*. The method is responsible to check if the host name of the server is matching the server's authentication scheme in the TLS session and also matches the XMPP servers defined service domain.

The properties *trustManagerImplementation* and *hostnameVerifierImplementation* in the *xmpp.properties* file specify which implementation of those interfaces to use for securing the connection with the XMPP server over TLS (full Java class name). There are two predefined implementations *AcceptAllCertificatesTrustManager* and *AcceptAll-HostnameVerifier* in the MPM core framework, which are implemented to accept all kinds of PKI certificates and host names of a server. If an application developer wants to implement a trustworthy TLS communication, a valid certificate from a CA has to obtained and installed on the server. Then the *TrustManager* and *HostnameVerifier* have to be implemented in the application as described and have to be configured in the *xmpp.properties* file.

6.4.3 Countermeasures against DoS and spamming

As has been shown in Section 6.2.1, there exists the functionality to block and unblock specific other peers in the network or even allow only a list of peers to be able to communicate with a peer. This functionality is offered by the *IConnection* interface of the communication layer in the MPM framework. Still, it would be possible to flood the XMPP server with requests.

Every peer in the MPM framework has exactly one RTP. Every RTP, when communicating over XMPP, has exactly one connection to the XMPP server. This connection can be accessed by the application developer from the RTP and the two methods to white-list peers or to block and unblock peers can be called. The white-list has to be provided to the RTP before it is started. Blocking and unblocking of specific peers can be done at runtime. These are features of the XMPP protocol and the configuration is stored on the server. To be more precise, the server stores a privacy list with this configuration and checks the list on every arrival of a message for the peer.

6.4.4 End-to-end encryption

The reasoning about introducing optional E2E encryption in the system and the selection of the OTR protocol as choice for the reference implementation has been discussed in Section 5.5.4. Also, the functional principle of establishing secure sessions and verifying them with SMP has been shown in detail. In the MPM framework, the *IConnection* interface (see Section 6.2.1) has been extended to an *ISecureConnection* interface to offer additional functionality to establish an E2E encrypted channel.

In Figure 6.7 the interface is depicted. The methods from the *IConnection* interface are not shown again in the diagram. An implementation of the *ISecureConnection* manages

an underlying *IConnection*. With this concept the used *IConnection* can be exchanged by another implementation without affecting the encryption and the functionality of the communication has not to be implemented again by the *ISecureConnection*. Here, the secure connection just passes method calls on to the underlying connection to offer the same functionality as the *IConnection* interface but with additional E2E encryption. The only exception is the method *send(IEntry)*, here the entry has to be encrypted first. Then the *send(String, IIdentity)* method has to be called with the encrypted cypher-text. The diagram shows the methods introduced by the *ISecureConnection* interface. Before being able to send E2E encrypted messages, a secure session with a specific remote IIdentity has to be established (method *establishSecureChannel(IIdentity)*). In the reference implementation this can happen manually by a call to the method or automatically when the *send(IEntry)* method is called. When an encrypted cypher-text arrives at the underlying insecure connection of a destination peer, the insecure connection passes on the cypher-text to the overlying secure connection and calls the *handleIncomingEncryptedMessage(String, IIdentity)* method. The encrypted text might be part of some handshake or key exchange of the E2E protocol or might be an encrypted entry. If it is an entry, it gets decrypted and handed over to the *EntryMessageProcessor* of the underlying connection to be processed in the RTP. The *verifySecureChannel(IIdentity)* method is intended to offer functionality to verify the identity of both partners of an established secure channel. Usually some kind of common secret among those is necessary to do this. Therefore, a *IRequireSecretProcessor* can be registered on the connection with *registerRequestSecretProcessor(IRequestSecretProcessor)*. An implementation of the *IRequestSecretprocessor* implements two methods *requireQuestion(IIdentity)* and *requireSecret(String, IIdentity)*. In the implementation of the *verifySecureChannel(IIdentity)* method a developer might call those methods to get a specific question to ask to a specific communication partner from the application user and as a counterpart might ask a user for the answer (secret) to a specific question. The *verifySecureChannel* method is then responsible to verify if those secrets are the same or not. Furthermore, it is possible to use and configure an application-wide secret to use for verification if wanted. All other methods of the interface are self-explaining.

In the *OtrSecureConnection* implementation of the *ISecureConnection* interface that is delivered with the first version of the MPM framework, the *establishSecureChannel(IIdentity)* is implemented like described in detail in Section 5.5.4. When the OTR handshake has been successfully performed, encrypted entries can be sent with *send(IEntry)* following the secure data exchange using OTR, like explained in Section 5.5.4. The functionality of this secure session is offered by the *SessionImpl* class of the *Otr4j* library.

All the secure sessions an MPM RTP has established with other peers are managed within the *OtrEngineHostImpl* class, which is an implementation of the *Otr4j OtrEngineHost* interface. This host keeps track of all sessions and receives events on all of them (e.g. when a session is successfully established or when verification succeeded or failed etc.). It is also responsible for storing the exchanged keys of the OTR protocol for later reuse, if another secure session shall be established between two peers. Therefore,

Figure 6.7: The *ISecureConnection* interface

an implementation *OtrKeyManagerStoreImpl* has been done in the MPM framework according to the *OtrKeyManagerStore* interface of *Otr4j*. This implementation offers methods to store private and public encryption keys and also boolean properties in password secured files. The verification of sessions (*verifySecureChannel* method) in the *OtrSecureConnection* is implemented using the SMP protocol, which is offered out-of-the-box by the OTR protocol version 3 and the *Otr4j* library (see Section 5.5.4 for further details on how the verification process works).

The most important methods of the *SessionImpl* class of the *Otr4j* library are:

- *startSession()* - initiates the establishment of a secure OTR session
- *transformSending(String)* - encrypts a *String* according to the session keys
- *transformReceiving(String)* - decrypts a *String* according to the session keys
- *injectMessage(AbstractMessage)* - sends a message
- *initSmp(String, String)* - initiates the SMP protocol with a question and secret
- *respondSmp(String, String)* - answers a SMP request with a question and secret

All the necessary configuration for the E2E encryption within an MPM framework based application can be configured in an *encryption.properties* file in the resource folder of an application.

Performing the configuration in that file, an application developer can easily set up E2E encryption with OTR. The only additional action to do is to register the *IRequestSecretProcessor* when using verification of sessions with SMP. When a session calls the processor, it can be used to e.g. print a dialog box to the user of the application to enter

the secret for a question of a sender. Additional implementations of the E2E encryption are possible for a framework developer by implementing the *ISecureConnection* interface and configuring the implementation in the *encryption.properties* file.

6.5 Scalability

The proposed scalability concept from Section 5.6 stated that central parts of the network shall be scalable. These are the XMPP server and the notifier peer. Furthermore, also the identity database of the XMPP server as well as the database of the notifier and the registration server are central parts. Nevertheless, as has been discussed in Section 5.6, they are not crucial to the performance of the system, but for the availability. Both the clustered XMPP server as well as all notifier peer instances are each designed to use one common database connection. It has been explained how these two databases can be clustered, e.g. in the cloud, to increase availability. In a large, productive application that is based on the MPM framework it would be essential to do so. Moreover, also the registration server role could be clustered to ensure availability of initial accession of peers to a network. It has also been explained in Section 5.6 how the registration server, which in its reference implementation is basically a web API offering registration functionality running as a Spring Boot application, can be clustered in the cloud.

This section deals with the scalability of the relay server and the notifier peer, which are the components that are essential for performance of an MPM framework based application. In Section 8.1 the outcome of some scalability tests is shown and evaluated to examine the system behavior under load.

6.5.1 Setup of the XMPP cluster

The detailed concept of how the relay server can be scaled has been introduced in Section 5.6.1. Like stated there, a clustering plug-in can be installed on the *Openfire* XMPP server very easily. After installing that plug-in, it has to be configured in a configuration file. Furthermore, a load-balancer has to be installed on a server, if the cluster is not already hosted in a cloud service, where a load-balancer can be switched on for the cluster usually by just a few clicks. For the tests on the local infrastructure an own load-balancer has been installed and configured.

The clustering plug-in for the *Openfire* server is called *Hazelcast*, which is licensed under the open-source *Apache License 2.0*. Details on the possible configuration are provided in the *Hazelcast* documentation on the *Ignite Realtime*website [9]. Basically it is enough to enumerate all member addresses of the cluster in this file, these can be IP addresses or host names. Listing 6.4 shows an example of a cluster configuration with two members provided as public routable host names running the XMPP server at default XMPP port 5222. Additionally, very important for the cluster to work is that the server-to-server

[9] `https://www.igniterealtime.org/projects/openfire/plugins/hazelcast/readme.html` accessed 03.2017

communication port, which has been configured in the *Openfire* administrator console (default is 5269), is enabled at every server's firewall.

```
...
<join>
   <multicast enabled="false"/>
   <tcp-ip enabled="true">
     <member>server1.domain.at:5222</member>
     <member>server2.domain.at:5222</member>
   </tcp-ip>
   <aws enabled="false"/>
</join>
...
```

Listing 6.4: Example of a cluster member listing in hazelcast-cache-config.xml

After the cluster has been set up successfully, connections have to be distributed amongst the server instances. This means that a load balancer has to be installed, like depicted in Figure 5.10. An application developer might use a cloud clustering service to host the XMPP cluster, where a load balancer can usually just be switched on. Nevertheless, some details on how a load balancer can be installed and configured to work with an *Openfire* cluster shall be described. An open-source load balancer, namely *HAProxy* (see the *HAProxy* homepage[10]), which is licensed under the *GPL v2* license (see the license specification on the *gnu.org* website[11]), is used. Details on how *HAProxy* works have been provided in Section 5.6.1. To configure the load balancer for the XMPP cluster it is necessary to bind every incoming TCP connection on the used XMPP port (default 5222) to one instance of an XMPP server. Furthermore, the algorithm can be specified according to which the connections shall be distributed to the back-end servers. An example of a configuration can look like this (part of the *haproxy.cfg* file):

```
...
frontend xmpp
  bind *:5222
  mode tcp
  default_backend xmpp-servers

backend xmpp-servers
  balance roundrobin
  server server1 server1.domain.at:5222 check
  server server2 server2.domain.at:5222 check
```

Listing 6.5: Example *HAProxy* configuration for an XMPP cluster with two servers

As can be seen in the listing above, every incoming (frontend) TCP connection on port 5222 is bound by a round-robin (one after another) algorithm to servers specified in the

[10] http://www.haproxy.org/ accessed 03.2017
[11] https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html accessed 03.2017

back-end *xmpp-servers*. For further details on other available balancing algorithms see the documentation of *HAProxy*. Also, the balancer performs health checks on each of the back-end instances and only distributes to servers that are available.

6.5.2 Scaling the notifier peer

In Section 5.6.3 it has been explained how the notifier peer can be scaled in an MPM framework based network. As every instance of a notifier gets a unique resource GUID when communicating over XMPP and every RTP randomly selects one of the available notifiers by requesting all those notifier resources at start-up, the notifier peer is scalable by just starting different instances of it on different hosts. Load is automatically be distributed amongst them, additional notifiers can be added dynamically. When implementing another communication protocol for the network, the lookup for notifier peers might be implemented in a different way.

XMPP offers presence information about users via the *Roster* class (see *Smack* documentation on the *Ignite Realtime* website[12]). In this class there exists a method *getPresences(String user)*, which delivers all presences of a user. In this way, e.g. by getting the presences of *notifier-peer@domain.at*, all different notifiers with their resources would be delivered, e.g.:

- *notifier-peer@domain.at/c0eb483f-946e-4c28-b7b1-d6745a71c2d6*
- *notifier-peer@domain.at/277aab13-4d07-477b-8ce6-41b93a7377ed*

The resource GUID is always appended after the full XMPP name. When a peer is connecting to the network, after retrieval of all the notifiers and randomly selecting one, it sends all notification requests to this specific notifier resource. In the first version of the framework implementation there is no automatic fail-over when a notifier resource fails. Peers that use a notifier that fails will be able to send notifications again when restarting the RTP and selecting a new notifier resource.

6.6 Integration with partner work

This section shall briefly describe how the different components in the range of responsibility of this work and the work [Til17] integrate to the first version of the MPM framework. In Section 5.2.3 it has been defined how responsibilities for the engineering process are distributed.

6.6.1 Integration of communication and encryption

The first integration point of the MPM framework, which has been designed by Peter Tillian, is that every RTP instantiates exactly one *Connection* or *SecureConnection*

[12] http://download.igniterealtime.org/smack/docs/latest/javadoc/ accessed 06.2017

instance (*Singleton Proxy* instance that instantiates the concrete class), depending on the E2E encryption configuration. Depending on which Java classes have been configured in the *general.properties* file for an unencrypted connection and in the *encryption.properties* for an encrypted connection, an implementation is chosen and instantiated, defaults are *XmppConnection* and *OtrSecureConnection*. This instantiation happens at RTP start-up and afterwards the IO peers (receiver and sender peer) use the exposed methods of this *IConnection* to receive and send entries. Encryption is completely transparent to the RTP and happens automatically when sending over an instance of *SecureConnection*. Also, serialization of entries to be sent on the network happens transparently. An instance of *IEntrySerializer* accordingly is used by the implementations of the *IConnection* interface. Furthermore, the RTP exposes a method to get the used *IConnection* instance (*getConnection()*). Thereby, e.g. it is possibly for an application developer to use the functionality of blocking and unblocking other peers, setting up a white-list or using SMP to verify secure channels by calling the exposed methods of the *IConnection* or *ISecureConnection*.

6.6.2 Integration of serialization and persistence

Secondly, there is another integration point, the implementation of persistence in the system is using the serialization part to store entries efficiently. Like all other components, the persistence management of the MPM framework is represented by an exchangeable interface *IPersistenceManager*, which offers all functionality to persist entries on a host to lose no data when a RTP is shut down or has a failure. Peter Tillian has implemented a reference implementation of the interface for *Android*, which makes use of the *IEntrySerializer* interface to serialize the entries efficiently (e.g. with *protobuf*) and store them in a database.

CHAPTER 7

Proof-of-concept implementation of mobile application

This chapter gives an overview of a proof-of-concept implementation of a mobile application for *Android* that is based on and leverages the functionality of the MPM framework. The intention of this implementation is to show that the framework is usable for productive development and shall give an example for a possible use case. Also, it shall serve as an incentive for possible usage of the framework. The code of the application is delivered as an own project together with the code of the MPM framework.

7.1 Background

The reference application is an instance of the master-slave pattern. In this pattern one or more masters partition a task into sub-tasks and send them to a list of slaves to process the sub-tasks and send the results back to the master. By that means, computation power of many slaves can be used to solve a computationally intensive task collectively and in parallel. A precondition for such a model is that the task is dividable into sub-tasks.

The computation time of the fitness function of individuals in genetic algorithms (GAs) can be computationally complex. In [SP94] the concept of GAs is explained. Basically, a GA is a technique to search for and optimize a solution to a hard problem, which is usually not solvable in polynomial time in reference to the size of the problem. The GA generates a set (called population) of random solutions (called individuals) to the problem. Then, some or all individuals are selected for the so-called crossover. Here, usually two individuals are selected to produce one or more child-solutions that inherit a partial solution of their parents. After that, all the new individuals usually are mutated a little bit, that means that the solution gets only slightly changed to gain more diversity. By crossover and mutation a new generation of individuals is created out of the old

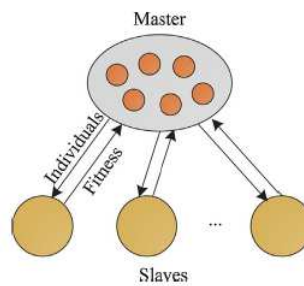


Figure 7.1: The master-slave model of a GA [GCZ+15]

population. Usually a selection for the replacement of the old population is done by evaluating the fitness of the new individuals, which is the quality of the solution in terms of optimizing the problem, while favoring better solutions to be in the next generation. This approach is called *genetic*, because that is basically what happens with the genes of creatures in the nature. The steps of crossover, mutation and selection are clarified in the section of the implementation.

Like mentioned, the evaluation of the fitness of individuals is important for the selection part of a GA. Depending on the problem, this evaluation might be the most computationally intensive part of the algorithm. Because the fitness evaluation is done for every solution separately, there exists the possibility to distribute the evaluation of the whole set of newly created individuals to workers who do the evaluation and send the result back to the master(s). In [GCZ+15] Gong et al. have done a survey on different models of such distributed evolutionary algorithms. Besides other models also the master-slave GA is explained. Figure 7.1 shows the master-slave model for a distributed GA. The master takes the steps of initial creation, selection of parents, crossover to create children, mutation of children and selection for the new generation. Before selection happens, the individuals are sent to the slaves for evaluation of their fitness.

7.2 Design

The application uses the reference implementation of the MPM framework for *Android* embedded in an *Android* service. It is a simplified example on how the framework can be used to implement a distributed GA on mobile devices. For the purpose of understandability, a well-known and quite easy non-polynomial solvable problem is used, namely the travelling salesman problem (TSP) [LKM+99]. The problem is about finding the shortest round trip in a set of given cities (or generally locations), where every city has to be visited once. One can think about the fitness evaluation of one of these tours, which is just the summation of the distances from one city to the other in the tour. Although this evaluation is not very complex in computation, this problem has been chosen because it is easier to understand, implement and evaluate than most other non-polynomial optimization problems. Also, the application-specific code is easily exchangeable for other

problems by still using the same MPM, because the model strictly separates application from coordination code. The selection of the problem therefore does not lead to a loss of generality.

In the system there are a master and slave peers as well as individuals (solutions to the problem) to be evaluated. Individuals are exchanged between the peers within the data objects of entries. The master maintains a list of all known slave workers and distributes the solutions to them by using a balancing algorithm, e.g. round robin. Figure 7.2 shows the communication flow between the UI, the master and a worker *WorkerX*, which represents the one or more worker peer(s) in the system. The user starts the master by sending a start command to the master service by using the UI interface of the RTP. The wanted population size is passed as a parameter. More parameters for the GA, like e.g. type of selection, could be added for configuration here, but for the demonstration the population size is enough. While the master gets started and initiates the population, workers may send a join request to the master to be selected for fitness evaluation. In the diagram this means that *WorkerX* gets registered if a join request is received by the master from this worker. This also might occur while a GA is already running. After the master has received this request by the worker it will start distributing solutions for evaluation to the joined worker. The actual GA is running as a loop. It starts with the crossover and mutation of the population at the master. Then, all currently joined workers, selected by a balancing algorithm, get the created solutions for evaluation in an inner loop. When all solutions have been evaluated by the workers and have been sent back to the master, the selection and replacement steps are performed at the master and the current best solution within the population is propagated to the UI. The whole process is running until it gets interrupted by a stop command from the UI. The UI of the peers is not shown in the diagram, it is only used to log in to the system with the peer's credentials. The master in this reference implementation is distinguished from the workers when logging in, because of a preconfigured master peer name. The application code is the same for worker and master peers, but different MPM services are executed at them.

7.3 Implementation

Some important details on the implementation of the distributed GA mobile application that is based on the MPM framework are discussed in this section. For any further remarks, please refer to the provided source-code of the application including documentation.

7.3.1 Master peer

The master peer is one of the two different roles that the RTP of the distributed GA application can take, depending on if a master user-name is used to log in or not. In this simplified scenario, the master peer contains three different wirings. The scaffold of the *Android* service, which contains the MPM framework, is implemented by Peter Tillian in his work [Til17]. When extending this *Android* service, there exists a method

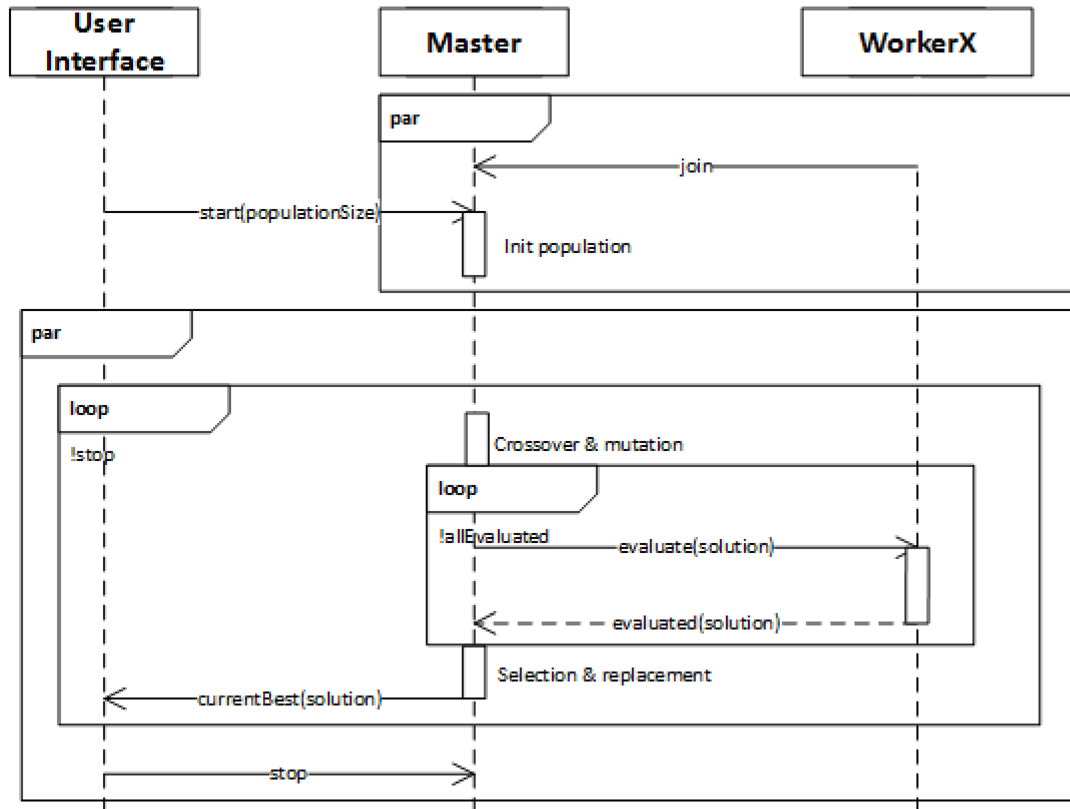


Figure 7.2: Sequence diagram of a master and worker communication of the distributed GA

initializePeers() where the structure of the RTP (local peers and wirings) can be defined by a programmer or generated by a modeler.

Listing 7.1 shows the content of the *initializePeers()* method that is relevant for the master peer in this application. First, the local peer *masterPeer* gets created. Then the first wiring for the peer initialization is added to the peer. The guard of the wiring needs an *INIT_MASTER* entry that comes from the UI (user presses start) and at least one *JOIN* entry from a worker to start the *InitDistributedTSPMasterService* MPM service. The details of this service are not explained, but basically it just generates a specified amount of random *Tours* for the TSP, wraps them in entries of type *EVALUATE* and addresses each to a worker that comes from the worker registry at the master peer, selected in a round-robin manner. The action of this wiring then sends all created entries of type *EVALUATE* to the designated worker peers.

The second wiring specifies what happens when an entry of type *JOIN* arrives at the master peer from a worker. The guard takes the entry and starts the *joinWorkerService*, which does nothing else than adding the worker that sent the entry to the worker registry

at the master.

After all the individual solutions have been evaluated in terms of fitness by the workers and have been sent back to the master, the guard of the third wiring continuously takes entries of type *EVALUATED* in the specified amount of the population size and calls the service *TSPDistributedGeneticAlgorithmService*, which performs the steps of crossover and mutation. The new individuals again get wrapped into entries of type *EVALUATE* and represent the new generation of the population, which again get sent to the workers for evaluation. Besides the action to send the entries to evaluate, there exists an additional action on this wiring, which writes possible entries of type *EVALUATED* back to the PIC of the master peer. This is done because by configuration the continuous GA service might keep the fittest individual(s) always also in the next generation.

```
1  (...)
2  IPeer masterPeer = runtimePeer.createPeer(Constants.masterPeerName);
3
4  //MASTER INITIALIZATION
5  IGuard initMasterGuard = new Guard();
6  initMasterGuard.addLink(LinkOperation.TAKE,
       EntryCount.largerEquals(1), EntryTypes.INIT_MASTER);
7  initMasterGuard.addLink(LinkOperation.TAKE,
       EntryCount.largerEquals(1), EntryTypes.JOIN);
8  IService initMasterService = new
       InitDistributedTSPMasterService(this);
9  IAction sendToursToEvaluateToWorkers = new Action();
10 sendToursToEvaluateToWorkers.addExternalLink(EntryCount.largerEquals(1),
       EntryTypes.EVALUATE);
11 masterPeer.addWiring(new Wiring(initMasterGuard, initMasterService,
       sendToursToEvaluateToWorkers));
12
13 //MASTER JOINING OF WORKER PEER
14 IGuard joinWorkerGuard = new Guard();
15 joinWorkerGuard.addLink(LinkOperation.TAKE,
       EntryCount.largerEquals(1), EntryTypes.JOIN);
16 IService joinWorkerService = new JoinWorkerService(this);
17 masterPeer.addWiring(new Wiring(joinWorkerGuard, joinWorkerService,
       null));
18
19 //MASTER CONTINUOUS GENETIC ALGORITHM
20 IGuard takeEvaluatedGenerationGuard = new Guard();
21 takeEvaluatedGenerationGuard.addLink(LinkOperation.TAKE,
       EntryCount.exactly(populationSize), EntryTypes.EVALUATED);
22 IService geneticAlgorithmService = new
       TSPDistributedGeneticAlgorithmService(this);
23 IAction sendForEvaluationAction = new Action();
24 sendForEvaluationAction.addExternalLink(EntryCount.largerEquals(1),
       EntryTypes.EVALUATE);
25 sendForEvaluationAction.addInternalLink(ContainerType.PIC,
```

```
      EntryCount.largerEquals(1), EntryTypes.EVALUATED);
26  masterPeer.addWiring(new Wiring(takeEvaluatedGenerationGuard,
      geneticAlgorithmService, sendForEvaluationAction));
```

Listing 7.1: Wirings for the master peer defined in the *initializePeers()* method of the *Android* service for the application.

7.3.2 Worker peer

The worker peer in comparison to the master peer is quite simple. It has one wiring that takes all available entries of type *EVALUATE*. Then it executes the *EvaluateIndividualsService* that computes the fitness of the individual(s). The action of the wiring then sends the entry with the new type *EVALUATED* back to the master peer. Listing 7.2 shows the code inside the *initializePeers()* method that is relevant for the worker peer.

```
1  (...)
2  IPeer workerPeer =
     runtimePeer.createPeer(WorkerRegistry.workerPeerName);
3        IGuard takeIndividualsToEvaluateGuard = new Guard();
4        takeIndividualsToEvaluateGuard.addLink(LinkOperation.TAKE,
            EntryCount.largerEquals(1), EntryTypes.EVALUATE);
5        IService evaluateIndividualsService = new
            EvaluateIndividualsService(this);
6        IAction sendEvaluatedIndividualsToMasterAction = new Action();
7        sendEvaluatedIndividualsToMasterAction
8        .addExternalLink(EntryCount.largerEquals(1),
            EntryTypes.EVALUATED);
9        workerPeer.addWiring(new
            Wiring(takeIndividualsToEvaluateGuard,
            evaluateIndividualsService,
            sendEvaluatedIndividualsToMasterAction));
```

Listing 7.2: Wiring of the worker peer defined in the *initializePeers()* method of the *Android* service for the application.

7.3.3 The TSPDistributedGeneticAlgorithmService

The most relevant MPM service of the application is the *TSPDistributedGeneticAlgorithmService*. It performs the generation of new individuals out of the old population by crossover and mutation and runs in a wiring in the master peer. The service implements the abstract class *AbstractDistributedGeneticAlgorithmService*. The class contains the *execute()* method of the service, which performs the steps of getting the fittest individual and sending it to the UI, selecting parents and creating new individuals by crossover of the parents, mutating the new individuals, wrapping them in an entry of type *EVALUATE* and addressing them to workers from the worker registry with a round-robin algorithm. This is done until the whole new population has been created and addressed.

The implementation *TSPDistributedGeneticAlgorithmService* of this abstract class just implements the crossover and mutation operators of the GA. Other problems could be implemented easily by a developer, who has to create an implementation for the new optimization problem by extending the abstract class.

Generally, a developer has to implement the representation class (in case of the TSP a class *Tour*) of the problem and has to provide an implementation of the crossover and mutation operators in a class extending the *AbstractDistributedGeneticAlgorithmService*. Moreover, the initialization service for the master peer has to be implemented. Then, the developer has to exchange these two services in the *initializePeers()* method (see listing above) and the whole system will be able to optimize a different problem. The worker peers are completely general and do not have to be redefined. Only the representation of a solution must extend the abstract class *Individual*, which exposes one method *getFitness()* that is called by the worker peer for fitness evaluation, no matter which optimization problem the individual instantiates.

Crossover and mutation operators and representation of a *City* and *Tour* for the TSP have been implemented based on the blog article by Jacobson on the *theprojectspot.com* website[1]. The used crossover operator for two parents is the *order crossover* [Cic06]. Mutation is done at a configurable mutation probability for each city inside the tour by just swapping the city with another randomly selected city, this guarantees some additional diversity of new individuals. In the code it is also configurable whether the fittest solution is always kept in the next generation.

7.3.4 The UI

The UI for this demonstration application can be used for master and worker peers as shown in Figure 7.3. The user has to provide the credentials to log in to the network and has to specify a population size for the GA. When pressing start, the *Android* service gets started with the population size as a parameter. Depending on whether the user is logged in with the configured master user name, a *INIT_MASTER* or a *JOIN* entry is injected to the RTP and the system starts. On the master peer, at every new generation the currently fittest solution is sent to the UI like described and printed out to the user. In this example application, the cities for the TSP are provided in the static class Constants and the list is retrieved from there by the MPM service that initiates the master. In the graphics you can see the optimization result for 29 cities, including all EU cities and also Bern with their real coordinates. The tour in this demonstration is fetched from a configuration file where it has been hard-coded. After around 30 minutes of evaluation the master in a network with two workers already shows the overall geographical linear distance of a tour through all cities with 15,617 km. After that, the evaluation has been canceled. A further improvement to the application could be to show the evaluated best tour in a list or another graphical representation.

[1] `http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5` accessed 04.2017

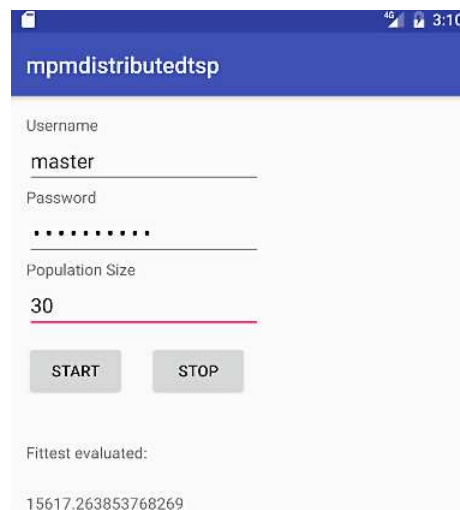
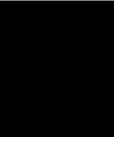


Figure 7.3: User interface of the distributed GA application that is based on the MPM

7.3.5 Further considerations

The distributed GA demonstration application is kept quite simple, here some further consideration shall be made on how to improve the application. First, more parameters could be configurable using the UI and not only a configuration file to let the end user adjust the GA more dynamically. An example for such parameters could be the mutation rate or if the fittest individual shall always survive to the next generation. The second aspect is that it is possible for workers to join the network at any time to contribute to the evaluation of individuals. Nevertheless, if a worker crashes or disconnects in this scenario, the master might not receive all evaluated individuals that are necessary. Here, it might be advantageous to implement some sort of time-out at the master in which a worker has to respond with evaluated individuals. If the worker does not respond in time the master would then remove the worker from the worker registry, fill up the population again with random solutions and let the other workers evaluate them. A scenario to use such a master-slave pattern using the MPM framework in a public application would in this case not necessarily trigger any security issues as usually no secret data is transferred, although it would probably be a good idea to somehow check if a slave is returning reasonable results and is not trying to pollute a generation of individuals.

CHAPTER 8



Evaluation

In this chapter an evaluation of my main responsibilities of this work shall be done. Therefore, scalability of the centralized parts of the system are tested. The concept for scalability for the reference implementation of the framework has been discussed in Section 5.6. Tests are performed in a local area network and also in a cloud environment. Details on the used hardware and set-up constellation are provided. The aim of the scalability evaluation is to measure how the system components of the MPM implementation, especially the communication and serialization components, perform under heavy load in a single instance and in a horizontally scaled set-up.

Furthermore, an assessment regarding the security of the implemented framework is done and analyzed in accordance to some main attack scenarios that have been discussed in Chapter 2.

8.1 Scalability evaluation

The evaluation of scalability of the central parts in the communication layer are done in this section. Accordingly, a test application has been programmed, which simulates a configurable number of client runtime peers that communicate with each other by exchanging entries. This test application is implemented to run on a desktop or server machine and not on an *Android* device as the purpose is to test only the central parts of the MPM network. Configurable parameters to the test application are the number of client peers to simulate, the amount of payload data (string data in bytes) that shall be sent in each exchanged entry and also the amount of time (milliseconds) that a peer shall pause after receiving an entry before sending a reply entry. A fixed and randomly selected communication partner peer is assigned to each peer in the test application and these peers then exchange entries in a *ping-pong* manner, that means after receipt of an entry an answer entry is sent back to the sender and so on. By controlling the amount of peers and the pause time between receipt and answer, the general throughput of the relay

XMPP server or server cluster can be measured. Furthermore, by configuring the payload of the entries, also the performance of the serialization component can be tested to some amount. The throughput of a single server instance and also of clustered instances shall be compared in a local and in a cloud environment.

Notification functionality is disabled in the tests, because when notifications are switched on, simply the amount of entries that are sent is doubled, because each sent entry triggers a notification entry being sent to the notifier peer(s). Instead, a stability test is performed on a notifier peer in a local and in a cloud environment. It is also shown that scaling the notifier by just starting more notifier peers in the network reduces the load on each of them.

8.1.1 Infrastructure

Local environment

The testing devices of the local scalability tests are named like shown in the list below. In the consecutive sections these names are used to refer to the specific hardware:

- *host1*: quad-core Intel Core i7-4770K processor with 4 Ghz , 8 GB DDR-3 RAM, Windows 10
- *host2*: dual-core Intel Core i7-4600U processor with 2.1 Ghz, 8 GB DDR-3 RAM, Windows 10
- *host3*: dual-core Intel Core i5-2467M processor with 1.6 Ghz, 4 GB DDR-3 RAM, Windows 10
- *host4*: dual-core AMD E2-1800 APU processor with 1.7 Ghz, 8 GB DDR-3 RAM, Windows 10

The hosts are locally connected in a wireless LAN of the standard *802.11n*.

The load balancer used in the local environment is hosted on a *Hyper-V* virtual machine running Ubuntu 16.10 on *host1*. It has been configured to be able to use up to 1 GB of RAM and up to 50% of the available processor power of all 4 cores. The used load balancing software is *HAProxy* (see the *HAProxy* website[1]) and is configured to use a TCP round-robin balancing algorithm to the clustered instances on port 5222 (see Section 6.5). The database for the XMPP servers is hosted on *host1* and is a *PostgreSQL* database of version 9.5.

[1] `http://www.haproxy.org/` accessed 04.2017

Cloud environment

The cloud infrastructure is hosted on Amazon web services (AWS) elastic cloud computing (EC2) instances of type *t2.micro*. The hardware of these instances is a virtual Intel Xeon E5-2676 processor with 2,4 Ghz and 1 GB of RAM. The instances are configured as *shared*, which means that they are hosted together with other EC2 instances of other *Amazon Cloud* users on one dedicated machine (this does not mean that all the VMs for the test run on the same host). For horizontal scaling, 3 of these instances have been provisioned in an *Amazon* VPC (see the VPC description on the *Amazon* website[2]). This means that they are hosted in a virtually isolated area of the AWS, where they can connect to each other via private IP addresses like in a LAN. The *Openfire* XMPP servers are hosted on these machines running a Windows Server 2016 operating system. The test application described above runs locally on *host1*. When performing the tests against the cloud environment, the used internet connection provides effectively around 90 Mbps upload and around 8.7 Mbps download rate from the side of the local internet service provider. The AWS EC2 instances are hosted in the AWS region *US West (Oregon)*. The database for the XMPP servers is a *PostgreSQL* 9.5 database hosted on an *Amazon* relational database service (RDS) (see the RDS description on the *Amazon* website[3]) in the same VPC.

8.1.2 Scaling the relay server

Throughput of communicating peers

Tests on average throughput of entries and average server CPU usage have been performed for a local single and clustered XMPP server and also for a single and clustered XMPP server hosted in the cloud. There have been 38 measurements in the local environment and 46 measurements in the cloud environment. Every measurement has been performed with different amounts of communicating peers in the described *ping-pong* manner for exactly 10 minutes without a pause time between receiving and sending entries. An average of entries received and sent per minute has been calculated over these 10 minutes by the test application. Average CPU usage in percent for the *Openfire* server process has been calculated using the built-in Windows performance monitor tool for each measurement timeslot. All measurements have been performed one time with entries without payload data (data property) and one time with 100 KB of text data.

The statistics in Figure 8.1 shows the results of all measurements in the local environment. For these tests, the test application and the XMPP database have been run on *host1*. The single instance relay server tests have been performed with the XMPP server running on *host4* and the clustered tests with servers running on *host2*, *host3* and *host4*. The *HAProxy* load balancer has been hosted on a virtual Ubuntu 16.10 machine on *host1*. The results for the single instance tests is kept in blue color in the diagrams, the results for the 3 instance tests in orange. The first column of the figure shows results for entries

[2]`https://aws.amazon.com/vpc/` accessed 05.2017
[3]`https://aws.amazon.com/rds/` accessed 05.2017

without payload and the second column for tests with entries with 100 KB of payload data. The first row shows the average throughput of entries per minute at each peer for specific amounts of communicating peers. The second row depicts the average, overall throughput of entries per minute of the server(s) for specific amounts of peers. The last row shows the average server processes CPU usage in percent for specific amounts of peers. Please notice that only the y-axis is linear for reasons of scaling. The maximum amounts of participating peers in the tests have been adjusted to each scenario (cloud or local and data or no data). The amount of the users has not been increased any more when the average throughput of entries per user per minute reached 1 or below or when connectivity became unstable, e.g. when single users lost connection to the server.

Figure 8.2 demonstrates the results of the server throughput tests where server instances have been hosted in the cloud. The test application is once again run on *host1* and the XMPP database is hosted on *Amazon* RDS. For load-balancing an AWS *Classic Load Balancer* (see the load balancer description on the AWS website[4]) has been used. It can easily be configured to distribute TCP connections (on port 5222 for XMPP) to EC2 instances in the same VPC. The scheme of the statistics diagrams is analogous to the results of the local environment.

When analyzing the throughput statistics you might notice that in the local environment for fewer users a single instance can even outperform a cluster of three XMPP servers, this obviously is due to the overhead that the control messages of the clustering software add. Still, at 700 communicating peers the cluster can already exceed the single instance by around 33% for entries without payload. For entries with payload of 100 KB the cluster of 3 instances unexpectedly does not perform better in terms of throughput. This might be due to a network bottleneck at the hosts with the additional overhead of the clustering control messages. Especially *host3* and *host4* have quite limited resources (5 year old low budget notebook and 5 year old tablet computer) and seem to slow down the whole clustering control mechanism. At least the cluster consistently shows a much lower CPU workload of the XMPP server process on each instance, especially for the tests with payload data, where the serialization component generates workload. Here, with roughly the same results for throughput at higher amount of participants, at least the CPU load of the single instance is around 285% of the same clustered instance (measured on *host4*).

In the cloud environment the throughput measurements for clustered and for a single instance showed roughly the same results without payload data. Also, the throughput increased with a higher amount of participants, but it happened that connections became unstable for some users in the clustered scenario with no payload when reaching 400 participants. With payloads of 100 KB, the clustered server performed better than the single instance with lower amount of users. When increasing the participants, the single instance scenario even outperformed the cluster for some range of user numbers. Here again probably the reason is the overhead that the clustering control messages add. The server CPU load was, analogous to the local environment tests, always much lower for clustered instances.

[4] `https://aws.amazon.com/elasticloadbalancing/classicloadbalancer/`

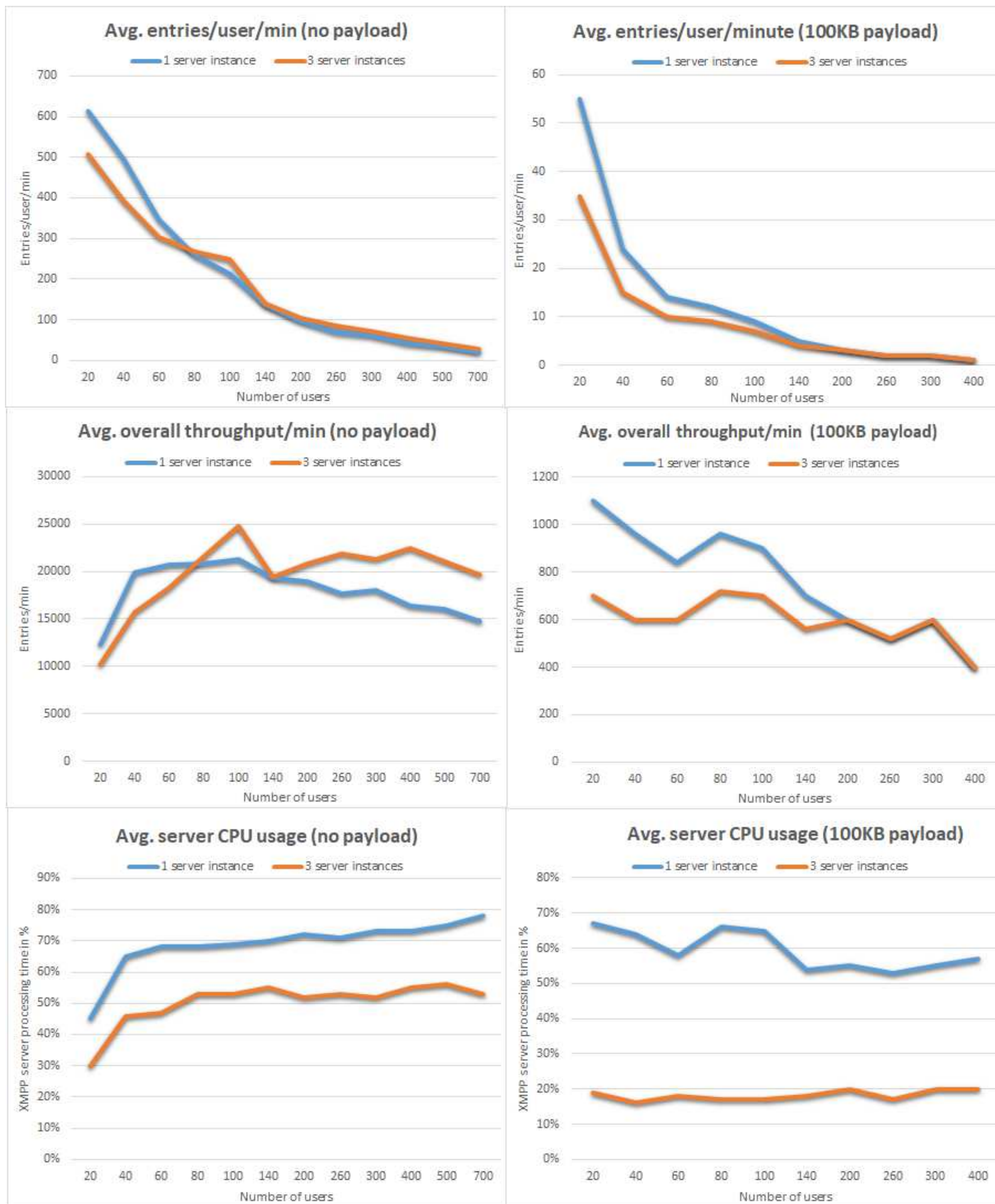


Figure 8.1: Statistics of throughput and server CPU usage of local XMPP server(s)

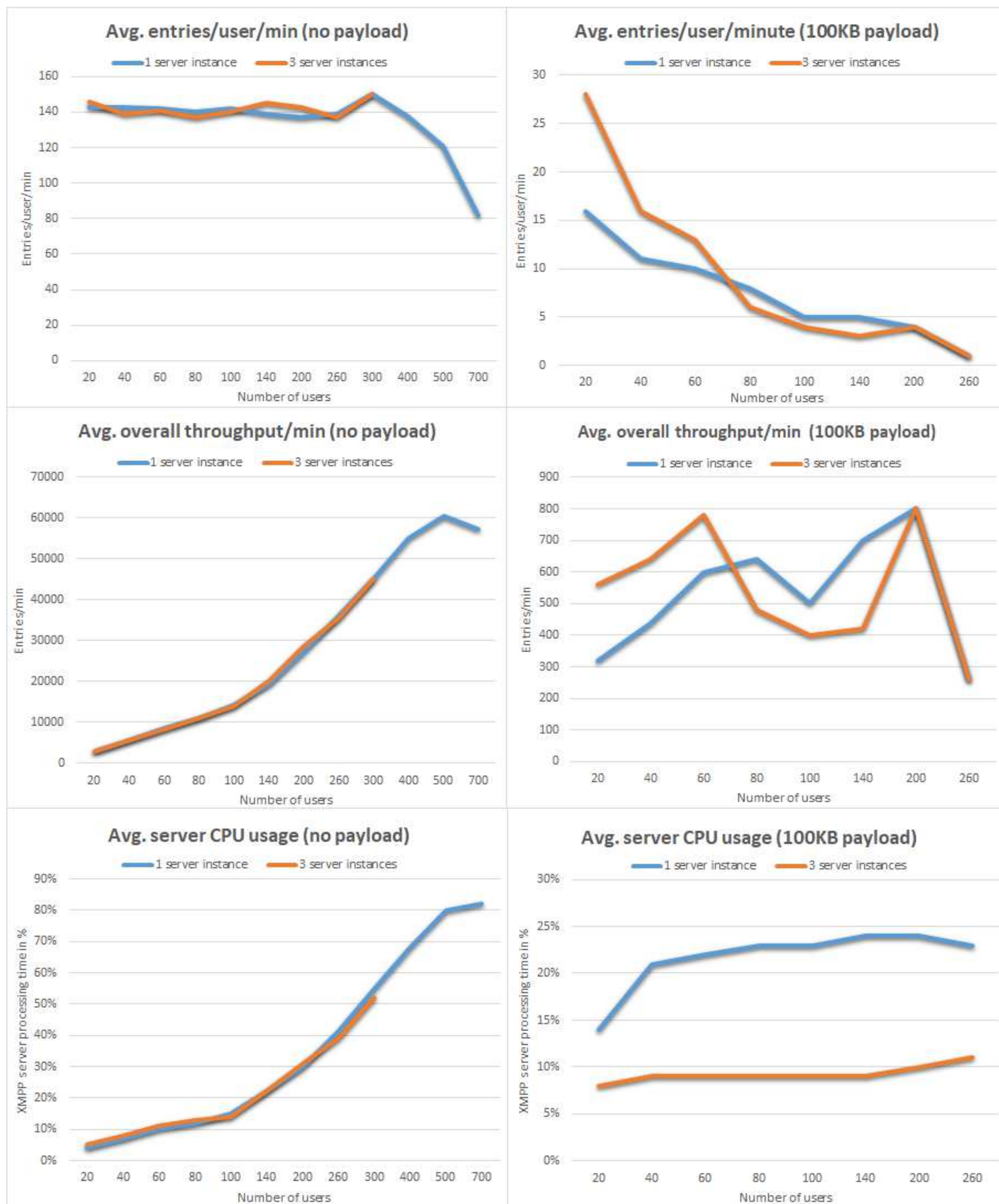


Figure 8.2: Statistics of throughput and server CPU usage of XMPP server(s) hosted on AWS

Studying the results one might argue that besides better reliability in the chosen cloud environment the cluster only shows a performance benefit of lower CPU load on all the instances, but not in terms of entry throughput. Nevertheless, it has to be considered that the tests have been performed on quite undersized cloud machines with low resources. Probably, the network interface becomes the bottleneck here where cluster control messages counterbalance a possible increase in throughput of the cluster. It would be interesting to perform the same tests on large-scale cloud machines and with more than 3 servers in the cluster.

Simultaneously connecting peers

In a productive system it might occur that many client peers try to connect in a small timespan. Therefore, a test has been conducted where many peers try to connect quasi simultaneously to a network in order to be able to determine which amount of connecting peers a server or cluster can handle without producing communication exceptions on one or more of them. The test has been performed with the test application running on *host1* and has been executed 10 times with a configured amount of peers to determine if the establishment of the connection is repeatedly successful. Then, the amount of users has been increased by 10 and the test has been reiterated. The already described test application has been configured to start the specific amount of runtime peers as threads from a thread pool in a loop, which then immediately try to connect to the server.

In a local set-up with 1 XMPP server instance running on *host3* up to 220 peers were able to connect in the timespan of start-up of all peers without issues. At 230 and more peers communication exceptions occurred in at least one test run of the 10 iterations. When adding one more XMPP server role on *host2*, the number of peers being able to connect without issues could already be increased to 320. At 330 and more peers communication exceptions occurred. With one additional instance this is an increase of around 45%.

Hosting one relay server in the AWS cloud, 30 peers were able to connect in the timespan of their simultaneous start-up, whereas exceptions occurred at 40 or more. With 2 clustered cloud instances already 50 peers were able to connect, exceptions occurred at 60 or more connecting peers in one or more of the test runs. That is an increase of about 66%.

These tests at least prove that clustering the relay server roles significantly increases the amount of peers that can simultaneously connect to the network without getting communication exceptions.

8.1.3 Scaling the notifier

When notifications are enabled, each connecting client peer automatically selects one connected notifier peer from all currently available notifiers. The notifiers can be identified by their host-name, which is configurable in the framework. The performance of the notifier is dependent on the hosting machine and on the number of incoming notification

entries. For every sent entry on a network, a notification entry is sent to one notifier peer, which then calls the FCM servers for user notification.

In the local environment a single notifier has been hosted on *host4* and has been put under heavy load by communicating peers. They have been simulated by the test application, the notifier database and a single XMPP server all running on *host1*. At a throughput of around 37.500 notification entries per minute, created by 20 connected peers, the notifier crashes after 17 minutes with a memory exception. When adding 2 more notifiers on *host2* and *host3*, the first notifier (on *host4*) only crashes after 53 minutes with the same exception. This is an increase of around 311%.

In the cloud environment the notifiers and the XMPP servers have been hosted on separate EC2 *t2.micro* instances in the same VPC. The notifier *PostgreSQL* 9.5 database, like the XMPP database, is also hosted on *Amazon* RDS in the same VPC. A single notifier crashes after 16 minutes at a load of around 49.000 notification entries per minute, created by 500 connected peers from the test application running on *host1*. With the same amount of notification entries a scaled notifier with 3 instances did not crash also after 3 hours of test run, when the test was aborted. An overload on the scaled notifier was not possible, because the probably needed throughput could not be reached by the relay server.

8.2 Performance of E2E encryption

A performance test has been executed on the E2E encryption module. The described test application has been used running on *host1*. Two communicating peers have been started to exchange as many entries as possible (without pause between receiving an sending) for 15 minutes. The tests have been performed with and without encryption and with and without payload. The XMPP server has been running on *host3* and the identity database has been running on *host1*, notifications have been disabled. The tests showed the following results (showing avg. CPU usage of *host1* and avg. send/received entries per minute):

- No encryption and no payload: avg. 8855 entries/user/min and 1% CPU usage
- No encryption and 100 KB payload: avg. 316 entries/user/min and 2% CPU usage
- Encryption and no payload: avg. 3070 entries/user/min and 13% CPU usage
- Encryption and 100 KB payload: avg. 209 entries/user/min and 3% CPU usage

Analysing the results it can be derived that enabling the OTR E2E encryption decreases the throughput by around 65% when no payload is sent and by around 34% with 100 KB of payload. When additionally considering that the CPU usage increased drastically when encryption is enabled with no payload and did not increase significantly with payload it

can be deduced that encryption has a much higher impact on the throughput and on the workload than the serialization component. Nevertheless, it could be examined how the serialization component performs on more complex data objects (than just string data) in the future.

8.3 Security assessment

A security assessment on the first version of the delivered framework is done in this section. The framework is measured against different possible attacks that have been discussed when evaluating possible background technology in Chapter 2. Different kinds of attacks are grouped and assessed in attack classes below.

8.3.1 Eavesdropping

An attacker might eavesdrop the traffic in a network and might get access to the transmitted data in clear-text. To avoid that, the framework communication layer offers encryption of network traffic with TLS over XMPP, which can be configured to be enabled for all communication. In this case, the server needs to provide a trustable and verifiable PKI certificate. The framework offers the opportunity to implement an *X509TrustManager* interface to verify the server certificate and to specify which CAs signing the certificate are allowed. Also, the framework allows implementation of a *HostnameVerifier*, which allows verification of allowed hostnames of the server in an existing TLS session.

It might be the case that the server cannot be trusted because it does not provide a trustworthy certificate or that a developer wants to implement an application where the relay server does not need to be trusted by the application users at all. When using TLS over XMPP only, the server might still read the transmitted data in clear-text. To avoid eavesdropping completely for all other entities but just the two communicating partners, the framework offers the possibility to implement an interface *ISecureConnection* over that any transmission of data is E2E encrypted. Here, keys for encrypting the data are generated on the communicating devices and are bound to that devices only. The framework offers an implementation of this interface using the OTR protocol, like described in Section 6.4.4.

8.3.2 Data modification and replay

An attacker might modify the transmitted data without knowledge of the sender or receiver. The provided OTR implementation of E2E encryption adds a MAC to each transmitted, encrypted message, which makes it possible for the sender to verify the authenticity and integrity of every single received message. See Section 5.5.4 for more details.

Furthermore, an attacker might repeatedly send (replay) data packets to trigger some malicious side-effects in the application. This is also prevented by the OTR protocol,

which uses new shared secrets on every message, so that a repeatedly sent message would be identified as invalid already at the communication layer level of the receiving peer. See Section 5.5.4 for more details.

8.3.3 Identity spoofing and MitM

Attacks on the identity include forging of data packets that appear to originate from a different identity in the network, taking over completely the identity of a specific node and receiving and transmitting data on their behalf during a conversation or even faking an identity from the beginning of an initial conversation with a partner and claiming to be someone else. To address these attacks, the chosen communication protocol, XMPP, serves as a central identity provider and offers a mapping of an authenticated (with user credentials) connection to a username in the network. When additionally E2E encryption with OTR is switched on for communication, it is impossible to forge the origin of a packet or to take over someone's identity during an established secure conversation, because only the two communicating peers hold the necessary secret keys to encrypt and authenticate a message and following messages. Any forged message would be identified by the receiving peer.

One could argue that the initial handshake of an E2E encrypted channel could be intercepted by a MitM, e.g. an untrusted relay server, and key negotiation could happen with that attacker instead of the real identity by both communication partners, making it possible for the attacker to decrypt all exchanged data. A second scenario could be that a communication partner claims to be a specific identity from the beginning of the registration at the network but instead is an attacker. The framework integrates the possibility to use the SMP in conjunction with OTR to verify the identity of a communication partner by proving that both hold a common secret that only those two can know. This assumes that such a secret exists and is done without actually transmitting the secret. The framework provides an API that offers this functionality in a usable way to the application developer, making it possible to verify the channel in a simple question and answer manner for the user. Once a verification is done, future communication channels can be verified automatically if the secret is locally stored by both devices. By that means, any identity spoofing and MitM attacks can be avoided.

8.3.4 Compromising secret keys and passwords

As E2E encryption is based on the concept that identities hold private keys, an attacker that compromises this key might be able to establish secure channels with others on the behalf of that identity. Nevertheless, if the private key got compromised but the password for the identity provider did not get compromised and the relay server is trustable, the attacker would have to forge packets on behalf of that identity and additionally would have to make them look like having been received from the relay server, which altogether is not an easy task and has the assumption that no additional TLS with the server is enabled. In the framework implementation, the private secret key is stored in an

encrypted and password protected key file. In the first implementation the same password as for the identity provider is used. An attacker would therefore have to get access to the device to steal the private key of the user and additionally would have to know the user's password to be able to decrypt the file. With that information the attacker would be able to establish secure channels with communication partners, but would only be able to verify those channels if also the common secret is known to the attacker or has been cached by the application using the framework. By design, the OTR protocol makes it possible to decrypt future messages but no sent messages. An attacker in possession of the private key would therefore not be able to decrypt past messages. The other way around, if an attacker would compromise the user's password for the identity provider, it would still not be possible to verify a secure channel without additionally knowing the common secret for the SMP, also when connection from a different device with a newly generated private key is allowed (this is the case in the first implementation but could be made configurable to the application developer in future versions). Thus, as the attacker is not able to verify the secure channel, the communicating user would recognize at least that the channel is not verified, also when private key and the password are compromised.

8.3.5 Deniability and Privacy

It might be of interest to users of specific applications based on the implemented framework to not only preserve the confidentiality of the transmitted data, but also of their own identity. A registration at the relay server might generally be possible anonymously, depending on the used communication channel to the registration server role. Still, the relay server has to be trusted, because it knows the IP address of the connecting peer. The framework itself in the first version does not offer a user-configurable proxy server to hide the IP from an untrusted relay server, this might be done in future work. Also interesting would be to integrate the possibility to e.g. tunnel all traffic through the *Tor* (see the *Tor* project website[5]) network to the relay server. This could be done by providing a proxy configuration for the framework and using a transmitter application like *Orbot* (see the *Orbot* application[6]) to redirect any traffic from that application first through the *Tor* network. When traffic is additionally E2E encrypted, the relay server would only be able to see which usernames are communicating, but not their real identities (IP addresses). Additionally, anyone that would spy the traffic from outside would not be able to identify that a specific IP is communicating with the relay server, avoiding also any possibilities of IP harvesting. Even when *Tor* is not yet integrated, only a very general identity harvesting would be possible to an outside spy when monitoring the E2E encrypted traffic. To be exact, a harvesting of all IPs communicating with a specific relay server could be achievable.

What is already a feature of the OTR E2E encryption implementation of the framework is the deniability of sent messages. There might be a scenario where, e.g., a law enforcement or governmental institution wants to prove that someone has sent a specific message to

[5] `https://www.torproject.org/accessed05.2017`

[6] `https://guardianproject.info/apps/orbot/` accessed 05.2017

someone. This could be possible by proving that the MAC of a specific message originates from a specific private key on a personal device. Nevertheless, as OTR publishes the MAC key of a previous message with the current message in clear-text, anyone sniffing the traffic could have forged messages, signing it with the MAC and making it look like it originates from that sender. Therefore, deniability for messages is guaranteed to the outside, but also authenticity of a message is still verifiable for the intended receiver at the time of the communication.

8.3.6 DoS and spam

An attacker might flood central parts of a network to disrupt the whole network traffic and even prevent users from joining the network. Also, an attacker might run a DoS attack on specific users or send them spam messages. The implemented framework offers an API to block single peers that try to perform a DoS attack or send spam already at the relay server. The functionality is provided by the XMPP server implementation *Openfire*. Also, a white-list of communication partners can be defined to be able to contact a specific peer. This functionality has to be disclosed to an end-user by the application developer in the UI. Nevertheless, avoiding DoS on central parts of a public network is a hard problem and is not generally solvable. It is the same with other public server roles on the internet like e.g. web-servers. Avoiding DoS on the XMPP server(s) and on the notifiers are in scope of the administrator of the machines hosting these central network roles.

8.3.7 Application-layer attacks

Application-layer attacks subsume all attacks that happen on the application layer, like e.g. abnormally terminating the application or operating system or reading, adding, editing or deleting data in an unauthorized way. The first version of the framework does not provide many application layer security countermeasures out-of-the-box. It accepts all entries of non-blocked or white-listed users and checks if the *from* property of the XMPP message matches the *from* property of the received entry. Then it delivers the entry to the local peer that is designated in the *dest* property of the entry, if it exists. In all other cases and also when the type of the entry data object is not registered in the local type registries the entry gets ignored. Moreover, when an entry gets delivered to a specific local peer, containers of the peer only accept the entry, if there exists any wiring in the peer that might take or read entries of that entry type from that container. This is a small countermeasure against an attacker trying to flood a peer with entries that will always stay in a container without ever being taken, e.g. to let the peer run out of memory.

For future versions of the framework further application layer security countermeasures could be tackled, e.g. a general boundary for container sizes to avoid running out of memory. Also, a security concept for the peers or containers itself could be introduced,

where only specific senders are able to write entries to specific local peers and containers like proposed by Craß et al. in [CJK15].

Regarding further application layer security, the application developer is responsible, e.g. what happens to the data in the entries, which other side-effects are triggered in the implemented services or if a garbage collection of old entries in containers is performed (e.g. by appropriate TTL properties or clean-up wirings).

8.4 Fulfilment of imposed requirements

In this section a short conclusion about the fulfilment of the imposed requirements from Chapter 3 is drawn, beginning with the non-functional requirements.

The source-code of the provided implementation of the MPM framework is sufficiently documented and covered by unit and integration tests, is licensed under a copyleft license and will be published (requirements *NFR1*, *NFR5*). The framework core project has been designed in a modular way, making it possible to exchange important components, e.g. the communication, encryption and serialization that were in my scope (requirement *NFR6*). Besides pervasive tests, functionality of the framework has been proven by implementing two example applications based on the framework (proof-of-concept of requirement *NFR8*). All important interfaces of the components have been designed intuitively and have been discussed in the thesis paper to provide an additional information source for future framework developers (requirement *NFR4*). When implementing the framework based applications, it has been discussed in the thesis how the framework code is structured and how it can be generated by a modeller in the future (requirement *NFR7*). Peter has evaluated the benefit using the framework in comparison to implementing all functionality in his thesis (requirement *NFR9*). Also in his scope was the resource efficient implementation of the framework in terms of storage, battery consumption and network bandwidth (requirement *NFR10*), which could be fulfilled by using concepts like allowing the application to be suspended at any moment and the concept of the notification system. Reliability (requirement *NFR11*) can still be guaranteed by offering an implementation for persistence (also in the scope of [Til17]). The concept, implementation and evaluation of scalability for the central parts of the system as well as for appropriate security in an internet-scale network has been provided in this work (requirements *NFR2*, *NFR3*).

As Peter was responsible to design and implement the functionality of the MPM RTP, the requirements for coordination (requirement *FR1*), for the RTP being able to run in the background (requirement *FR2*), possible autonomous start-up of the RTP (requirement *FR3*) and the decoupling of the RTP to an outside *Android* application by implementing an *Android* service (requirement *FR4*) have been fulfilled in his work [Til17]. Requirement *FR5*, which states that the framework shall offer connectivity with local and mobile carrier networks, has been fulfilled in this work by using an XMPP server for relaying and thereby making it possible to work behind any NAT without any needed network configuration. The automatic handover when a new connection is re-established has been

implemented in the communication part of the framework integrating with an outside application triggering the appropriate events.

CHAPTER 9

Conclusion

In this conclusive chapter, a summary of the outcome and the results of the thesis is done. Also, some lessons learned are presented to give hints and insights for probable future work. Finally, some suggestions are made for probable future work based on the thesis and the delivered framework.

9.1 Summary

The outcome of the thesis is a P2P coordination framework for mobile environments. In the preceding literature review possible background technologies like P2P communication protocols and coordination frameworks have been researched and evaluated. The selection of technologies for communication, serialization, security, scalability and the coordination model has been done by evaluation against imposed requirements on the framework and the outcome was to use XMPP as communication protocol and the PM as coordination model. A mobile profile of the PM has been specified in accordance with the PM technical board.

The reference implementation for the proposed framework has been implemented for *Android* with respect to interoperability with probable implementations for *iOS* or with other profiles of the PM. The communication component was implemented using XMPP to guarantee operability in any network constellation. Besides encryption at the transport layer (TLS over XMPP) an additional E2E encryption module has been added to the framework, offering an OTR implementation in the first version of the framework. Furthermore, a concept for scalability of the needed central parts of the communication layer (XMPP server and notifier-peer) has been developed.

In the evaluation phase it has been shown that the central parts of the system are scalable by performing several load tests. The results of the scalability tests showed that the central parts when hosted in a local environment as well as when hosted in a

cloud environment perform better when horizontally scaled (clustered) in terms of entry throughput and stability. A performance test on the encryption layer has shown that enabling E2E encryption decreases the performance in terms of throughput by 63% with entries without payload and by 34% with entries with 100 KB of payload. Moreover, the security countermeasures offered by the framework have been evaluated in respect to common security threats.

Additionally, an example application (master-worker pattern to evaluate the fitness in a distributed GA) has been developed based on the implemented framework. The application, together with the second example application by Peter Tillian [Til17], serves as a proof-of-concept and provides some possible use-cases for the framework.

9.2 Lessons learned

Some lessons learned shall be recorded here for future developers to consider. Firstly, the serialization component with its two implementations *Google gson* and *protobuf* has been an unexpectedly high effort in the engineering process. Besides the selection among a very broad pool of available protocols, also the reading up on the functionality was a lot of effort, especially for binary serialization concepts like *protobuf*. A lot of considerations about serialization with the necessary registries for data types have also been discussed in the PM technical board before finding a way to go.

A good test coverage turned out to be very important during the software engineering process in this complex project. Without being able to execute regression tests on the whole project after some parts have been developed or adapted, the development would not have been successful, especially when relying on components of another developer. Furthermore, the tests serve as an additional source of documentation.

In the evaluation phase the testing of scalability was a higher effort than expected, that includes set-up of the testing infrastructure, implementation of a test application and execution of the tests, while doing the measurements on all devices. Also, the tests did not show the expected results in all scenarios. Nevertheless, in the end it could be shown that the chosen design of the network is scalable and also reliability and availability of the central parts can thereby be increased.

Moreover, it became clear that it is an additional organizational effort to do a collaboration on a thesis project, but on the other hand it was very helpful to get a second opinion before taking an important decision.

9.3 Future work

There are several suggestions for future work related to my responsibilities on the thesis. Firstly, framework developers can develop further implementations of exchangeable components like the communication, serialization and E2E encryption. To provide some examples, there could be a SIP implementation of the communication module or an

implementation for local communication with bluetooth or ethernet. Different serialization modules could be implemented, e.g. using *Apache Thrift* or an XML based approach. Other E2E encryption protocols than OTR, e.g. *Signal*, could also be supported by implementing the corresponding interface.

Also, the general features of the MPM could be extended. During the development of the thesis another master student was already using the framework for his thesis, it turned out that supporting *flows* (see Chapter 4) in the framework would be a feature he would like to have for his application. The plan is to integrate this feature in the next version of the framework.

For interoperability with other profiles of the PM a consistent way for addresses of peers and used protocols in networks as well as a consistent naming scheme for properties of entries have to be formally specified together with the PM technical board.

The scalability of the XMPP implementation of the communication component could be tested with a different XMPP server than *Openfire*, which probably also uses another clustering middleware than *Hazelcast*. Also, maybe the suggested set-up performs better on machines with more resources than have been used for the scalability tests.

Regarding security there could also be enhancements, like e.g. offering a configurable proxy server over which all traffic is routed first. This would also be a first step to integrate with the *Tor* network, like already discussed in Section 8.3.5. Moreover, further application layer security concepts could be introduced, e.g. an access control management for containers based on senders of entries like described in [CJK15].

List of Figures

2.1 *Napster* communication model [SGG03] 9
2.2 *Gnutella* communication model [LCP+05] 11
2.3 Freenet routing model [LCP+05] 13
2.4 *FastTrack* two-layered P2P network [LCP+05] 15
2.5 BitTorrent architecture with .torrent file, tracker and peers [LCP+05] 18
2.6 JXTA resource lookup [HYAK+04] 26
2.7 Session initiation over SIP [BAD06] 28

4.1 Example: Graphical notation of a peer with one sub-peer, one wiring and two services 46

5.1 Global market share of operating systems for mobile devices[1] 50
5.2 Components of the MPM framework including communication with enclosing application and remote nodes 51
5.3 Architecture of the MPM RTP including important components and flow of entries 53
5.4 Architecture of the access control and communication in the MPM network . 55
5.5 Sequence diagram of a new node registering at the registration server 56
5.6 Sequence diagram of a client peer sending an entry to another client peer . . 58
5.7 Activity diagram of de-serialization of entry and contained data object with *protobuf* 62
5.8 Sequence diagram of OTR handshake between Bob and Alice (see the post of *matthewdgreen* on *tumblr*[2]) 68
5.9 Sequence diagram of OTR data exchange and key publication between *Bob* and *Alice* (see the post of *matthewdgreen* on *tumblr*[3]) 69
5.10 Architecture diagram of XMPP clustering for scalability of the MPM framework 71
5.11 Diagram of a client peer randomly choosing one notifier amongst two available ones 73

6.1 Main classes of communication and encryption of the MPM framework with their dependencies (packages yellow, interfaces green, classes gray) 77
6.2 Main classes of the serialization component of the MPM framework with dependencies (packages yellow, interfaces green, classes gray) 78
6.3 The *IConnection* interface 80

6.4 The *IEntrySerializer* interface 85
6.5 The *IDataSerializer* interface 86
6.6 The abstract class *ProtobufTypeAdapter* 87
6.7 The *ISecureConnection* interface 91

7.1 The master-slave model of a GA [GCZ+15] 98
7.2 Sequence diagram of a master and worker communication of the distributed GA 100
7.3 User interface of the distributed GA application that is based on the MPM . 104

8.1 Statistics of throughput and server CPU usage of local XMPP server(s) . . . 109
8.2 Statistics of throughput and server CPU usage of XMPP server(s) hosted on AWS 110

List of Tables

2.1 Comparison of popular unstructured P2P protocols, state at 09.2016 22

3.1 Probable fulfilment of requirements by presented P2P technology 40

Acronyms

AES advanced encryption standard. 67, 68

API application programming interface. 29, 30, 33, 35, 36, 44, 45, 51, 54, 60, 64, 71, 88, 92, 114, 116

ARM advanced RISC machine. 10

ASCII American Standard Code for Information Interchange. 59

AWS Amazon web services. 107, 108, 110, 111, 124

CA certificate authority. 63, 65, 89, 113

CBJX crypto-based JXTA transfer. 24

CHK content-hash key. 11, 12

DDoS distributed denial of service. 10, 14, 18, 21, 63, 64

DHT distributed hash table. 16, 17, 24, 27, 29–32, 40

DNS Domain Name System. 80

DoS denial of service. 8, 14, 16, 18, 21, 29, 35, 63, 79, 88, 89, 116

DTLS datagram transport layer security. 29

E2E end-to-end. xiv, 51, 52, 63–67, 76, 79, 83, 88–92, 95, 112–115, 119–121

EC entry collection. 45

EC2 elastic cloud computing. 107, 108, 112

FCM Firebase cloud messaging. 54, 56–58, 70, 73, 81–83, 112

FIFO first in - first out. 44

GA genetic algorithm. 97–101, 103, 104, 120, 124

GUID globally unique identifier. 72, 73, 94

HTTP hypertext transfer protocol. 9, 14, 23, 27, 54, 57, 70, 71

ID identifier. 9, 12, 28, 29, 49

IDE integrated development environment. 35

IDL interface description language. 60, 83

IETF Internet Engineering Task Force. 26–28

IM instant messaging. 1, 12, 26, 27, 30

IO input/output. 49, 51, 95

IP internet protocol. 8, 10, 14, 16, 21, 23, 26, 27, 93, 107, 115

J2ME Java Platform Micro Edition. 25

JADE Java agent development framework. 23, 37

JSON JavaScript object notation. 59, 60, 62, 77

JVM Java virtual machine. 50, 51, 56, 57, 59

JXTA Juxtapose. 23–25, 32, 37, 39–41, 125

LAN local area network. 2, 6, 35, 63, 71, 79, 106, 107

MAC message authentication code. 67, 68, 113, 116

MitM man-in-the-middle. 12, 21, 24, 29, 35, 114

MPM Mobile Peer Model. 5, 7, 19, 28, 34, 37, 43, 47–57, 59, 60, 63, 64, 67, 70–73, 75–79, 81–83, 85, 88–92, 94, 95, 97–100, 102–105, 117, 121, 123, 124

NAT network address translation. ix, xi, 2, 7, 8, 10, 13, 14, 18, 19, 22, 24, 26, 27, 34, 39, 41, 54, 70, 117

NAT-PMP NAT port mapping protocol. 18, 22

NFC near field communication. 6

OTR off-the-record messaging. 64, 66–69, 76, 88–92, 112–116, 119, 121, 123

P2P peer-to-peer. ix, xi, xiii, 1–3, 5–9, 11, 13–17, 19–27, 29–32, 35, 37–41, 47, 63, 70, 119, 123, 124

P2PSIP peer-to-peer session initiation protocol. 26–30, 39, 40

PGP pretty good privacy. 64–67

PIC peer-in-container. 44, 45, 49, 52, 53, 101

PKI public-key-infrastructure. 63, 65, 88, 89, 113

PM Peer Model. xiii, 1, 3, 32, 41, 43–45, 47–50, 119–121

POC peer-out-container. 44, 45, 49

RDS relational database service. 107, 108, 112

RELOAD resource location and discovery. 26–30, 39

RIAA Recording Industry Association of America. 14

RPC remote procedure call. 61

RTP real-time transport protocol. 27, 48, 49, 56

RTP runtime peer. 44, 45, 48–53, 56, 57, 75, 78, 79, 81, 89–91, 94, 95, 99, 100, 103, 117, 123

S/MIME secure/multi-purpose internet mail extensions. 64–67

SIGMA sign and mac. 67

SIP session initiation protocol. 23, 26–30, 37, 40, 120

SMP socialist millionaire problem. 67, 69, 90–92, 95, 114, 115

SRDI shared resource distributed index. 24

SSK signed-subspace key. 11

TCP transmission control protocol. 26, 56, 57, 70, 79, 80, 93, 94, 106, 108

TLS transport layer security. 24, 29, 54, 63, 64, 70, 81, 88, 89, 113, 114, 119

TSP travelling salesman problem. 98, 100, 103

TTL time-to-live. 9, 12, 44, 45, 48, 53, 117

TTS time-to-start. 44, 45, 48

UDP user datagram protocol. 11, 15, 18, 22, 26, 27, 39

UI user interface. 48, 51, 52, 99, 100, 102–104, 116

UML unified modelling language. 75

UPnP Universal Plug and Play. 13, 18, 22, 39

URI uniform resource identifier. 27, 44, 49, 79

URL uniform resource locator. 6, 82

VCS version control system. 25

VM virtual machine. 57, 107

VoIP voice over IP. 1, 27, 30

VPC virtual private cloud. 71, 107, 108, 112

W-LAN wireless local area network. 13, 34

WAN wide area network. 6, 63

WWW world wide web. 12

XML extensible markup language. 23, 59, 121

XMPP extensible messaging and presence protocol. 23, 37, 41, 50, 51, 54–58, 63, 64, 67, 70–73, 76, 79–82, 88, 89, 92–94, 106–114, 116, 117, 119, 121, 123, 124

Bibliography

[AG07] Chris Alexander and Ian Goldberg. Improved user authentication in off-the-record messaging. In *Proceedings of the 2007 ACM Workshop on Privacy in Electronic Society*, WPES '07, pages 41–47, New York, NY, USA, 2007. ACM.

[AMHJ09] Joan Arnedo-Moreno and Jordí Herrera-Joancomartí. A survey on security in jxta applications. *Journal of Systems and Software*, 82(9):1513–1525, 2009.

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(03):329–366, 2004.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM computing surveys (CSUR)*, 36(4):335–371, 2004.

[BAD06] Nilanjan Banerjee, Arup Acharya, and Sajal Das. Seamless sip-based mobility for multimedia applications. *IEEE Network*, 20(2):6–13, 2006.

[Bai02] Damien Bailly. Cbjx: Crypto-based jxta. Technical report, Sun Laboratories Europe, 2002.

[BCJ04] Martin Boldt, Bengt Carlsson, and Andreas Jacobsson. Exploring spyware effects. In *Nordsec 2004*, 2004.

[BF11] Aiden A. Bruen and Mario A. Forcinito. *Cryptography, information theory, and error-correction: a handbook for the 21st century*, volume 68. John Wiley & Sons, 2011.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84. ACM, 2004.

[BM06] Joseph Bonneau and Andrew Morrison. Finite-state security analysis of otr version 2. Technical report, Stanford University. Stanford, CA, 2006.

[BST01] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1–2):23 – 36, 2001.

[BX11] Leonard Barolli and Fatos Xhafa. Jxta-overlay: A p2p platform for distributed, collaborative, and ubiquitous computing. *IEEE Transactions on Industrial Electronics*, 58(6):2163–2172, 2011.

[CGCD+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *Security and Privacy (EuroS P)*, pages 451–466. IEEE, 2017.

[Cic06] Vincent A. Cicirello. Non-wrapping order crossover: An order preserving crossover operator that respects absolute position. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pages 1125–1132, New York, NY, USA, 2006. ACM.

[CJK15] Stefan Craß, Gerson Joskowicz, and Eva Kühn. A decentralized access control model for dynamic collaboration of autonomous peers. In *International Conference on Security and Privacy in Communication Systems*, pages 519–537. Springer, 2015.

[CKS09] Stefan Craß, Eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, pages 301–306, New York, NY, USA, 2009. ACM.

[CMH+02] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, Jan 2002.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[Coh08] Kai Michael Cohrs. Implementation and evaluation of the peer-to-peer-protocol (p2pp) for p2psip. Master's thesis, Computer Networks Group, Institute of Computer Science, Georg-August-University of Göttingen, Göttingen, Germany, 2008.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies*, pages 46–66. Springer, 2001.

[DPK12] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

[DSMPR03] H. De Sterck, R. S. Markel, T. Phol, and U. Rüde. A lightweight java taskspaces framework for scientific computing on computational grids. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, pages 1024–1030, New York, NY, USA, 2003. ACM.

[EDGM07] Karim El Defrawy, Minas Gjoka, and Athina Markopoulou. Bottorrent: Misusing bittorrent to launch ddos attacks. *SRUTI*, 7:1–6, 2007.

[EMH16] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. End-to-end encrypted messaging protocols: An overview. In *International Conference on Internet Science*, pages 244–254. Springer, 2016.

[Fil10] James Filbert. Developing a multi-purpose chat application for mobile distributed systems on android platform. Bachelor thesis, Helsinki Metropolia University of Applied Sciences, 2010.

[FMB+16] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is textsecure? In *2016 IEEE European Symposium on Security and Privacy*, pages 457–472, March 2016.

[GCZ+15] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286–300, 2015.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[HB02] Oliver Heckmann and Axel Bock. The edonkey 2000 protocol. *Multimedia Communications Lab, Darmstadt University of Technology, Tech. Rep. KOM-TR-08-2002*, 2002.

[HBMS04] Oliver Heckmann, Axel Bock, Andreas Mauthe, and Ralf Steinmetz. The edonkey file-sharing network. *GI Jahrestagung (2)*, 51:224–228, 2004.

[HKLF+06] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 359–371, New York, NY, USA, 2006. ACM.

[HYAK+04] Erkki Harjula, Mika Ylianttila, Jussi Ala-Kurikka, Jukka Riekki, and Jaakko Sauvola. Plug-and-play application platform: towards mobile peer-to-peer. In *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, pages 63–69. ACM, 2004.

[JC10] Xing Jin and S-H Gary Chan. Unstructured peer-to-peer network architectures. *Handbook of Peer-to-Peer Networking*, pages 117–142, 2010.

[JKW07] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[JXJY06] Y. Jiang, G. Xue, Z. Jia, and J. You. Dtuples: A distributed hash table based tuple space service for distributed coordination. In *2006 Fifth International Conference on Grid and Cooperative Computing (GCC'06)*, pages 101–106, Oct 2006.

[KBM07] Marlom A. Konrath, Marinho P. Barcellos, and Rodrigo B. Mansilha. Attacking a swarm with a band of liars: evaluating the impact of attacks on bittorrent. In *Seventh IEEE International Conference on Peer-to-Peer Computing (P2P 2007)*, pages 37–44. IEEE, 2007.

[KCJ+13] Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In *International Conference on Coordination Languages and Models*, pages 121–135. Springer, 2013.

[KMKS09] eva Kühn, Richard Mordinyi, László Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '09, pages 625–632, Richland, SC, 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[Kra03] Hugo Krawczyk. Sigma: The 'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer, 2003.

[LA10] Lu Liu and Nick Antonopoulos. From client-server to p2p networking. *Handbook of Peer-to-Peer Networking*, pages 71–89, 2010.

[LCP+05] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys Tutorials*, 7(2):72–93, Second 2005.

[Li08] Jin Li. On peer-to-peer (p2p) content delivery. *Peer-to-Peer Networking and Applications*, 1(1):45–63, 2008.

[LKM+99] P. Larrañaga, C.M.H. Kuijpers, R.H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[LKR06] Jian Liang, Rakesh Kumar, and Keith W. Ross. The fasttrack overlay: A measurement study. *Computer Networks*, 50(6):842–858, 2006.

[LNR06] Jian Liang, Naoum Naoumov, and Keith W Ross. The index poisoning attack in p2p file sharing systems. In *INFOCOM*, pages 1–12, 2006.

[LP05] Zhen Li and M. Parashar. Comet: a scalable coordination space for decentralized distributed environments. In *Second International Workshop on Hot Topics in Peer-to-Peer Systems*, pages 104–111, July 2005.

[Mae12] Kazuaki Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, pages 177–182. IEEE, 2012.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[Rau14] Dominik Rauch. Peerspace.net. Master's thesis, TU Wien, 2014.

[RD10] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

[Rol09] Antoine Roly. Analysis and prototyping of the ietf reload protocol onto a java application server. Master's thesis, Facultés Universitaires Notre-Dame de la Paix de Namur, 2009.

[RPV11] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, 2011.

[RSHS14] Stefanie Roos, Benjamin Schiller, Stefan Hacker, and Thorsten Strufe. Measuring freenet in the wild: Censorship-resilience under observation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 263–282. Springer, 2014.

[SdCCGB10] Flávio Roberto Santos, Weverton Luis da Costa Cordeiro, Luciano Paschoal Gaspary, and Marinho Pilla Barcellos. Choking polluters in bittorrent file sharing communities. In *2010 IEEE Network Operations and Management Symposium-NOMS 2010*, pages 559–566. IEEE, 2010.

[SGG03] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia Systems*, 9(2):170–184, 2003.

[SL03] Sechang Son and Miron Livny. Recovering internet symmetry in distributed computing. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 542–549. IEEE, 2003.

[SM12] Audie Sumaray and S. Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th international conference on ubiquitous information management and communication*, page 48. ACM, 2012.

[SMR12] Anil Saroliya, Upendra Mishra, and Ajay Rana. A pragmatic analysis of peer to peer networks and protocols for security and confidentiality. *International Journal of Computing and Corporate Research*, 2(6), 2012.

[SP94] M. Srinivas and L. M. Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, June 1994.

[TC06] Richard W. Thommes and Mark Coates. Epidemiological modelling of peer-to-peer viruses and pollution. In *INFOCOM*, volume 6, pages 1–12, 2006.

[Til17] Peter Tillian. Mobile peer model - a mobile peer-to-peer communication and coordination framework. Master's thesis, TU Wien, 2017. (in preparation).

[TR11] Claudio Testa and Dario Rossi. On the impact of utp on bittorrent completion time. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 314–317. IEEE, 2011.

[TSIS12] D. S. Touceda, J. M. Sierra, A. Izquierdo, and H. Schulzrinne. Survey of attacks and defenses on p2psip communications. *IEEE Communications Surveys Tutorials*, 14(3):750–783, 2012.

[TVHVL13] Ha Manh Tran, Khoa Van Huynh, Khoi Duy Vo, and Son Thanh Le. Mobile peer-to-peer approach for social computing services in distributed environment. In *Proceedings of the Fourth Symposium on Information and Communication Technology*, pages 227–233. ACM, 2013.

[WDŻR10] Adam Wierzbicki, Anwitaman Datta, Łukasz Żaczek, and Krzysztof Rzadca. Supporting collaboration and creativity through mobile p2p computing. In *Handbook of Peer-to-Peer Networking*, pages 1367–1400. Springer, 2010.

[ZY02] Demetrios Zeinalipour-Yazti. Exploiting the security weaknesses of the gnutella protocol. Technical report, Department of Computer Science, University of California, 2002.