FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mobile Peer Model

A mobile peer-to-peer communication and coordination framework - with focus on mobile design constraints

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Peter Tillian, BSc
Matrikelnummer 01026312

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

Peter Tillian	Eva Kühn

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Mobile Peer Model

A mobile peer-to-peer communication and coordination framework - with focus on mobile design constraints

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Peter Tillian, BSc
Registration Number 01026312

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

Peter Tillian	Eva Kühn

Erklärung zur Verfassung der Arbeit

Peter Tillian, BSc
Schußwallgasse 1/10/13, 1050 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

Peter Tillian

Acknowledgements

Firstly, I would like to thank my thesis advisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eva Kühn and my thesis assistant Projektass. Dipl.-Ing. Stefan Craß of the Space Based Computing Research Group from the TU Wien, who both supported me from the beginning of my project.

Furthermore, I would also like to thank Martin Planer, Matthias Schwayer and Gerson Joskowicz for the useful comments in the weekly technical board meetings or during presentations of the current implementation status of the framework. Mention should also be made here of Konrad Steiner who used early versions of the framework and gave great feedback for improvements.

Next, a big thank you to my best friend since years, Jörg Schoba, for making this collaboration thesis a success. Thanks for the interesting discussions and not to forget for giving me a couple of motivation boosts.

I would also like to thank Sabrina and Daniela for their patience while proofreading the quite long thesis.

Finally, I want to express my very profound gratitude to my family, my father Arnold, my mother Sabine and my sister Marlene with her boyfriend Manfred for all the support in the last years. And last but not least, a big thank you to my girlfriend Anita, who always stood behind me - I hope in the next time we can compensate some long nights in front of the computer with some more activities together. Without you all, the completion of my study would not have been possible. Thank you!

Abstract

Various peer-to-peer coordination models and frameworks have evolved in the last 20 years, facilitating the design and implementation of complex coordination and collaboration tasks in highly distributed systems. However, none of them were specifically designed for mobile environments and they do not cope with typical constraints, like battery consumption, limited processing power and discontinuous availability. Nowadays, a mobile device can act as an autonomous node in such a distributed network and is also capable of performing complicated tasks due to increasing capabilities in hardware and software. Therefore, in the course of this work, a coordination framework has been implemented which is particularly tailored for mobile devices. The Peer Model developed at the TU Wien has been chosen as the underlying coordination model, for which also a reduced feature set has been elaborated as part of this work. By using the provided framework, application developers are supported in terms of P2P communication and coordination logic and can hence focus on the application logic only. The framework is implemented for the Android platform, but is developed with cross-platform considerations in mind. In addition to a proof-of-concept Android messenger application, the framework has been evaluated with performance benchmark tests.

Kurzfassung

In den letzten 20 Jahren wurden diverse Peer-to-Peer-Koordinationsmodelle und Frameworks entwickelt, die Designer und Entwickler von Applikationen beim Koordinieren von komplexen Datenflüssen unterstützen. Keine der bisherigen Lösungen wurde jedoch speziell für mobile Geräte in einem verteilten Netzwerk konzipiert und behandelt typische Einschränkungen wie z.B. limitierte Akkulaufzeit, schwächere Prozessorgeschwindigkeit und begrenzte Netzwerkverfügbarkeit. Heutzutage können mobile Endgeräte, aufgrund immer stärker werdender Hardware und ausgereifter Betriebssysteme, allerdings bereits komplizierte Aufgaben in einem P2P-Netzwerk übernehmen. Aus diesem Grund wurde im Zuge dieser Arbeit ein speziell für mobile Geräte optimiertes Koordinations-Framework entwickelt. Als zugrundeliegendes Koordinationsmodell wurde das an der TU Wien entwickelte Peer Model ausgewählt, für welches auch ein reduziertes und für ein mobiles Umfeld optimiertes Feature-Set ausgearbeitet wurde. Mit Hilfe des bereitgestellten Frameworks werden Entwickler und Designer, die eine P2P-Kommunikationsarchitektur benötigen und außerdem den Datenfluss zwischen verschiedenen Peers im verteilten Netzwerk koordinieren möchten, beim Entwurf und der Implementierung von mobilen Applikationen unterstützt. Einerseits wird die Entwicklungszeit verringert und andererseits können sich Entwickler auf die Applikationslogik konzentrieren. Das Framework steht für die Android-Plattform zur Verfügung, plattformübergreifende Aspekte wurden jedoch von Beginn des Softwareentwicklungsprozesses an beachtet. Das Framework wurde neben einer Proof-of-Concept Messenger Applikation für Android Geräte auch mit Benchmark-Tests evaluiert.

Contents

CHAPTER 1

Introduction

The Space Based Computing Research Group from the TU Wien is developing a programming model that facilitates the design and implementation of complex coordination and collaboration tasks in highly distributed environments.

The design of the so-called *Peer Model* [KCJ+13] is inspired by tuple-space communication, data-driven workflow and a staged, event-driven architecture. In a nutshell, the main components of the system are structured, re-usable and addressable constructs called Peers. They can exchange data with each other, encapsulated in so-called Entries, and have an internal coordination mechanism (Wiring). Only communication and coordination parts of the system are described in the model in order to enable separation of concerns and let application developers focus mainly on business logic.

1.1 Motivation and problem statement

In the last couple of years, the capabilities of mobile devices have been enhanced drastically. This is due to increasing processing performance, higher network availability and data transmission speed on the one hand and the usage of sophisticated mobile operating systems on the other hand. Therefore, a mobile device might act as an autonomous node in a highly distributed network and is also capable of performing complicated tasks.

The integration of mobile devices in a peer-to-peer network modeled by the Peer Model seems also certainly conceivable. However, the design of the model and also the current implementations are focussing on desktop and server environments. Although the capabilities are increasing steadily, a mobile device has still several limitations in contrast to desktop or server machines. These constraints, like limited processing power, battery consumption and uncontinuous availability, were not taken into account when designing and implementing the current profiles of the model. Furthermore, a thorough research

confirms the lack of existing P2P coordination frameworks that are designed specifically for mobile environments.

The Peer Model has to be modified in a way that the coordination principles and the semantics mostly remain the same, while typical mobile limitations are considered. On the one hand, the framework shall be usable for desktop and server applications to realize i.e. a central service, but on the other hand, it shall be possible to seamlessly integrate it into a current mobile platform. There, it shall not consume too much system resources even under heavy load in order to maintain usability. This is particularly important if the application is running in the background of a mobile device. Furthermore, in contrast to applications that are executed on a desktop or server machine, applications and especially background threads or services on a mobile device might be terminated by the mobile operating system at any point in time. The lifetime of a background process on a mobile device depends among other things on the amount of currently used resources, the hardware performance of the device and also the version of the operating system.

1.2 Aim of the work

The main goal of this thesis is the design and implementation of a framework that facilitates the development of applications that make use of P2P communication and coordination/collaboration functionality, while considering mobile limitations and constraints. As pointed out in the previous section, mobile devices can definitely act as autonomous peers in such a network due to increasing capabilities in hardware and software. Together with the technical board of the Space Based Research Group from the TU Wien, a reduced version of the original Peer Model shall be formulated, which is particularly designed for mobile environments. This model shall act as the underlying coordination model of the framework. On the one hand the framework shall be usable on any machines that can run a Java virtual machine (i.e. desktop or server) and on the other hand it shall also be possible to seamlessly integrate with the Android platform. Although it is only developed for one mobile platform, cross-platform considerations will be taken into account from the beginning of the software development lifecycle.

Before starting with the implementation of the Mobile Peer Model framework (MPM), existing coordination models and frameworks shall be evaluated and compared with the original Peer Model. Also peer-to-peer overlay network protocols and further potential background technologies shall be analysed and compared to each other.

1.3 Structure of the work

This master thesis is performed in cooperation with Jörg Schoba, also a software engineering student at the TU Wien. Individual areas of responsibility have been established before starting to impose requirements on the framework. However, some parts are conducted in close collaboration and those results will therefore be described in each work.

My area of responsibility is the core framework (including coordination functionality and important system components), mobile design considerations (i.e. how to integrate the core module in current mobile operating systems) and the persistence layer. Jörg's focus is on the communication and serialization layer as well as the security and scalability aspects of the system.

Before the implementation phase of the project started, we conducted a scientific literature research on P2P overlay networks, protocols and other background technologies that might be utilized in the framework. Also existing P2P frameworks and coordination models are presented and evaluated in Chapter 2. The subsequent Chapter 3 first deals with the functional and non-functional requirements on the framework (Section 3.1.1 and Section 3.1.2) and then covers an evaluation process if the found P2P systems and background technologies are suitable in accordance to the imposed requirements. An overview and relevant features of the existing coordination framework, the Peer Model, which shall act as the underlying coordination model for the framework is presented in Chapter 4. Chapter 5 describes the design part of the software development process. It starts by defining a reduced mobile version of the Peer Model, followed by an overview about the overall system architecture and important components. Then, crucial mobile design decision are discussed. Chapter 6 deals with the implementation of the framework and gives deeper insights into some particular parts of the system. In Chapter 7 important setup steps and implementation details of a proof-of-concept Android application are described. The subsequent chapter points out the benefits of the developed software solution by comparing the implementation effort of the presented P2P application with and without the MPM framework. Additionally, this chapter provides a brief performance analysis of the persistence layer. Chapter 9 contains a short summary and provides some ideas on future work.

CHAPTER 2

Analysis of existing approaches and background technologies

The first part of this chapter aims to find and present popular peer-to-peer overlay networks and their implementations. Then, more advanced peer-to-peer coordination frameworks and relevant background technologies of P2P systems are presented.

The research is performed in close collaboration with Jörg Schoba [Sch17a]. The peer-to-peer overlay networks are split up, me focusing on *structured* networks whereas Jörg is presenting the *unstructured* ones. Likewise, background technologies and more advanced coordination/collaboration frameworks are split up into two parts.

2.1 Review process

The review process is conducted in a structured and well defined way. The literature research is mainly performed via Google's search engine *Scholar*[1], which delivers results from different scientific literature pages. The search includes terms like 'peer-to-peer networks', 'mobile peer-to-peer', 'internet peer-to-peer', 'android peer-to-peer', 'iOS peer-to-peer', 'peer-to-peer protocol', 'internet peer-to-peer', 'mobile peer-to-peer communication' and 'mobile coordination framework'.

Most results linked to academic journals, books and papers of the digital libraries of *IEEE Xplore*[2], *Springer*[3] and the *ACM*[4]. Articles, papers and books that solely deal with Peer-To-Peer (P2P) communication in distributed systems, like P2P overlay networks, and additionally more advanced frameworks that also have an internal coordination

[1] `https://scholar.google.at/` accessed: 2016-10-01
[2] `http://ieeexplore.ieee.org` accessed: 2016-10-01
[3] `http://http://link.springer.com` accessed: 2016-10-01
[4] `http://dl.acm.org` accessed: 2016-10-01

mechanism are included. As a quality measure the citation count of the papers has been considered.

Works that describe technologies or frameworks applicable only in a Local Area Network (LAN) and not for distributed systems are excluded from the research. Also informal documents, like web forums or wikis, were not used. Official websites of reviewed technologies are added as an additional information source for the reader.

2.2 Structured P2P overlay networks

In the following subsections some important structured P2P network protocols that gained popularity in the last couple of years are presented and evaluated. In contrast to unstructured P2P networks, the content in all structured P2P networks is well organized. Nearly all of them make use of so-called Distributed Hash Tables (DHT), which store information about the location of objects. More precisely, for each data item a unique key is generated, which is then mapped to a specific peer. With this approach, an efficient discovery of data items is possible. Furthermore, the protocols define the behavior of the system when nodes join or leave the network to stay in a consistent state.

In Chapter 3 it is analyzed if the presented protocols can be chosen or easily adapted for a mobile environment, especially in the context of the requirements of the Mobile Peer Model (MPM).

2.2.1 Content Addressable Network (CAN)

The Content Addressable Network (CAN) [RFH+01] is a distributed P2P infrastructure and was one of the first implemented DHTs, developed in 2001. It is designed in a decentralized way and its key characteristics are scalability, fault-tolerance and self-organization.

Architecture and lookup mechanism: As described in [LCP+05], a CAN network uses a *d*-dimensional coordinate space to store key-value pairs of available files. The space is partitioned among multiple zones where each participating peer is responsible for a distinct part of the stored data (see Figure 2.1). The shared deterministic hash function maps a key K of a value V onto a point P in the coordinate space and can be applied by any peer. Each peer stores the coordinates of its immediate neighbors in the coordinate space. In a *d*-dimensional space two nodes are adjacent to each other if their coordinates overlap along *d-1* dimensions. If there is a lookup request on a peer for a data item whose calculated point P is not within its assigned coordinates, the peer routes the request to the closest neighbor peer by using a greedy forwarding approach. In such a situation a so-called *CAN* message is sent, which already holds the calculated coordinates of the items destination. In case a peer actually holds a requested data item, the item's value V is routed back in the same way through the network.

Leaving and joining the network: The architecture of CAN allows nodes to leave and join arbitrarily. A new node, that wants to join, has to know at least one node of the network and generates a random new point *NP* in the space. Then a JOIN request is sent through the network with the routing procedure mentioned above until it reaches the node which is responsible for the point *NP*. Next, this node is splitting up its zone and one half is assigned to the new node. As a last step, all nodes adjacent to the new node and the new node itself have to update their routing tables. The new node uses the routing information of the node that was split up before to get its neighbors and finally both the new node and the one that was split up send out update messages to their current neighbors.

In the event that a node wants to leave the network its neighbor's Internet Protocol (IP) addresses and the hash table have to be transferred to one of its adjacent nodes [ATS04]. The new owner of the zone then merges the zone of the leaving node and notifies its new neighbors about the change in the node structure. Apart from that, if a node unexpectedly fails and is not able to notify its neighbors about leaving the network, a periodical update message technique will detect the absence of a node with the result that a controlled takeover mechanism will be enforced. Under normal circumstances the neighbor of the unexpectedly failed node that is currently responsible for the smallest zone will take over the orphaned zone. In the unlikely case that more than one node is failing, an expanding ring search mechanism is initiated.

Further information: In [RWBB05] Reidemeister et al. reveal several attack vectors of the CAN protocol and provide some counter measures, for example for a Man In The Middle (MITM) attack. However, they do not deal with attacks on the application level (i.e. inserting unwanted content into a node). Potential applications built upon the CAN protocol are large distributed storage management systems that need fast retrieval and insertion of files. Different research work on CANs could be found - one implementation is for example the P2P Information Exchange Retrieval (PIER) query engine [HHL$^+$03], but none of them are still in production or in development. Aside from that, no implementation for a mobile platform could be found.

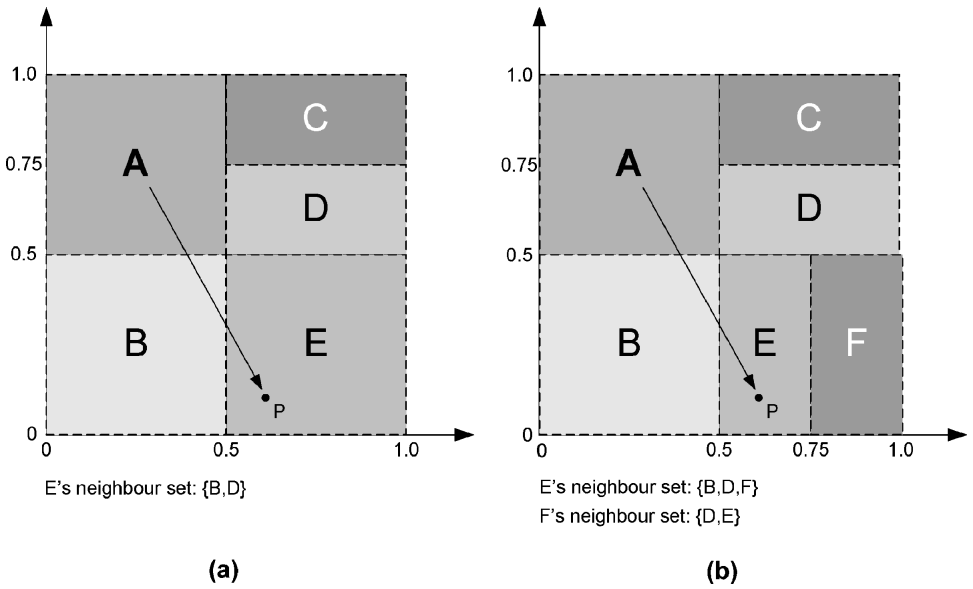

Figure 2.1: A *2-d* CAN coordinate space partitioned into (a) 5 nodes or (b) 6 nodes after node F joined the network [ATS04].

2.2.2 Chord

Chord [SMK+01] is another P2P overlay network protocol based on a Distributed Hash Table (DHT) and was introduced by a group of researchers at Massachusetts Institute of Technology (MIT) in 2001.

Architecture and lookup mechanism: Chord makes use of a variant of so-called consistent hashing, which was first described by Karger et al. in [KLL+97]. Generally, the advantage of consistent hashing is that when the size of the hash table changes, on average only K/N keys instead of K have to be remapped, where K is the number of already hashed keys and N is the number of different containers used. In the case of Chord this means that if a node joins or leaves the network, only a fraction of already hashed keys has to be transferred to another node. Additionally, consistent hashing ensures a normal distribution of keys on participating peers.

For each key and also each node an m-bit long Identifier (ID) is generated, according to [CC10] and [DGKW10]. This integer m is depending on the used hash function – for example if *SHA-1* is used the length of m is 160. Identifiers are ordered on an identifier circle modulo 2^M and range from 0 to 2^{M-1} (see Figure 2.2). A key K gets stored on the first node whose identifier is equal to or follows K regarding the identifier space. This node is then called the successor node of key K, denoted by *successor(K)*. Conversely, the predecessor node *predecessor(K)* of a key is the next node in the identifier circle in the counter-clockwise direction. In the simplest implementation each node only needs the routing information of its successor node on the circle. For a given key K, the query

is then routed along the circle from successor to successor until a node contains the requested key.

Leaving and joining the network: When a node n joins the network, specific keys that were previously assigned to the successor of n must be reassigned to the new node [ATS04]. Aside from that, if a node n gracefully leaves the network all its keys have to be moved to its successor.

Improvements: The implementation mentioned above, where each node simply stores its successor as routing information, has some disadvantages. First of all, the lookup speed is quite slow (by traversing over all nodes) and furthermore the availability decreases if different nodes fail simultaneously. Therefore, Chord introduces a second routing table per peer. The so-called *finger table* contains, apart from the node's successor and predecessor, m links to other peers of the network. Let's assume a node with ID n, then the i-th entry in the finger table points to the $n + 2^i$-th node. On an unsuccessful lookup request with key K on node n this node now forwards the lookup to the highest node with ID between n and k. As a result, the time required for resolving lookups is $\mathcal{O}(logN)$. In Figure 2.2 a sample lookup request is illustrated.

Further information: There are several open-source implementations of the Chord DHT in different programming languages (Java, C, C#, etc.). The reference implementation *The Chord Project*[5] is written in the C language. Projects in Java are for example *Open Chord*[6] and *Chordless*[7]. Unfortunately, there is no ongoing development on any of the projects. In addition, it has been proposed as a good candidate as an overlay technology used for P2PSIP (see Section 3.2.7). In [ZO09] some improvements for the protocol were presented to better support real-time communication systems.

Open-source implementations of Chord for the Android[8] and the iOS platform[9] exist, but those can be seen as experimental activities only. Nevertheless, plenty of research has been conducted regarding the Chord DHT, for example a distributed Domain Name System (DNS) application built upon the Chord infrastructure as described in [CMM02]. Moreover, there exists a version of Chord with the goal of providing censorship resistance by restricting the knowledge of the network for each node [HA02]. It makes use of a similar approach as the *Darknet mode* of the unstructured P2P protocol *Freenet*, which is described in more detail by Jörg in [Sch17a].

[5] `https://github.com/sit/dht/` accessed: 2016-10-01

[6] `http://open-chord.sourceforge.net/` accessed: 2016-10-01

[7] `https://sourceforge.net/projects/chordless/` accessed: 2016-10-01

[8] `https://github.com/puneetar/Chord---A-Distributed-Hashing-Technique--Android-` accessed: 2016-10-01

[9] `https://github.com/jeremytregunna/Outset` accessed: 2016-10-01

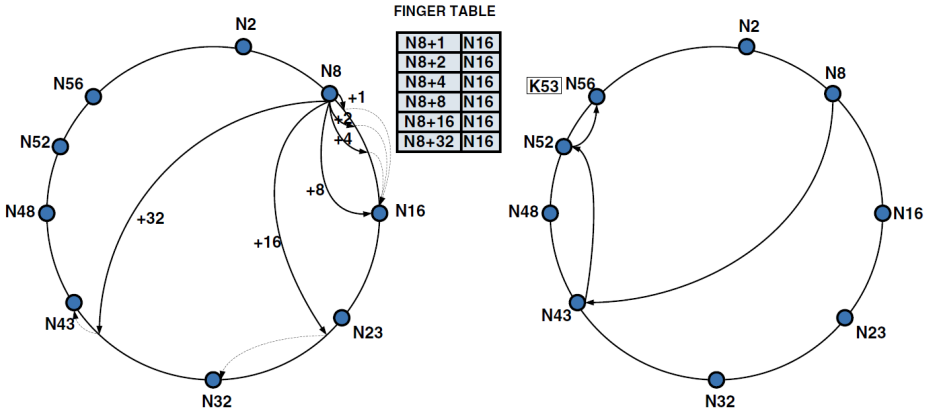

Figure 2.2: Example of a Chord identifier circle, including the finger table of node *N8* [DGKW10]. On the left-hand side the routing possibilities of *N8* are shown. The image on the right-hand side depicts the sequence of routing steps for a lookup with key *53* starting from node *N8*. Note that the finger table in the image is not correct. The value for entry *N8+16* should be *N32* and the correct value for *N8+32* is *N43*.

2.2.3 Pastry

Another self-organizing decentralized P2P overlay network called Pastry [RD01] was introduced by Microsoft Research Ltd. and the Rice University in 2001.

Architecture and lookup mechanism: Similar to the previously presented protocol *Chord* (Section 2.2.2), Pastry generates randomly and normally distributed IDs (with 128-bit length) for each participating node by hashing i.e. the IP address of the node. According to [SMR12], the node ID ranges from 0 to 2^{128-1} and is used to define the position in a circular identifier space.

To route messages, Pastry makes use of prefix matching (introduced by Plaxton et al. in [PRR99]). Therefore, each participating peer manages a routing table with $\log_B N$ rows and *B-1* columns. Variable B is a system parameter with the typical value of 16, defining the base of the chosen identifer (in this example hexadecimal). Each entry in row r defines a link to a node whose ID shares the first r digits with the present node (see also Figure 2.3). With that information Pastry routes a message to the peer whose ID matches the largest prefix of a given key. In addition, each node keeps a so-called leaf set L containing $|L|/2$ numerically closest larger node IDs and $|L|/2$ numerically closest smaller node IDs, in relation to the present node's ID [DGKW10]. Furthermore, a node manages a neighborhood set M, which contains the node IDs and the IP addresses of the $|M|$ closest peers, according to a proximity metric described in [RD01]. The size of L and M is typically B or a multiple of B – and therefore usually bigger than 16. By using those three routing properties eventual message delivery is guaranteed, unless

$|L|/2$ adjacent nodes fail simultaneously. With larger number of nodes in the leaf and neighborhood sets, reliability and fault resilience increase, but also the configuration effort. The maximum hop count from peer to peer for a lookup request is $\mathcal{O}(logN)$.

Leaving and joining the network: When a new peer with ID X wants to join the network it has to send a *JOIN* message to an arbitrary peer. This message is routed to the node with ID numerically closest to the new node's ID. All nodes which are visited during the join request update their routing tables accordingly and also the new node initializes its three routing properties with the received information. Finally, the new node sends routing information to all nodes that need to know about its arrival. When all tables are in a consistent state, periodical keep-alive messages are exchanged between connected nodes. If a node failure is detected, all connected nodes must be informed to update their routing tables as well as their leaf and neighborhood sets.

Further information: Several applications with different purposes built upon Pastry exist, i.e. group communication/event notification, archival storage, co-operative web caching, high-bandwidth content distribution and more. All of them can be accessed via the website of the open-source Java implementation *FreePastry*[10], developed by the Rice University (last version from 2009). Other implementations in various programming languages exist and a lot of research applications were developed built using Pastry. No mobile applications could be found, neither for the Android nor the iOS platform. Generally, an adaptation of *FreePastry*, which is written in Java 5, to a mobile platform would be possible. Nevertheless, necessary firewall and port-forwarding configurations[11] could not be performed in a public Wireless Local Area Network (W-LAN) - thus a mobile application would work only with restrictions.

[10] `http://www.freepastry.org/` accessed: 2016-10-02

[11] `http://www.freepastry.org/FreePastry/nat.html` accessed: 2016-10-03

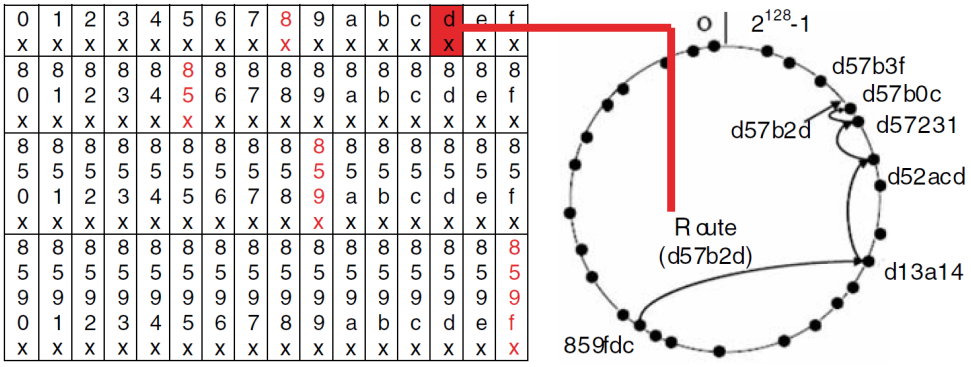

Figure 2.3: Example of a Pastry's routing table (left) and an identifier circle (right) [DGKW10]. On the right side a lookup chain is shown: the peer with ID *859fdc* is requested for an item with key *d57b2d*. Because the key shares *0* digits with the current node the *0*-th row is chosen and the column with the common prefix (here *d*) is selected. A routing message to the node is sent, which is stored under this index (here *d13a14*) in the table and the lookup process is continued. This procedure is repeated until node *d57b0c* is reached, which holds the key.

2.2.4 Tapestry

A similar structured P2P overlay network as the previously mentioned *Pastry* (see Section 2.2.3) is *Tapestry* [ZKJ01]. Also developed in 2001 by a team of researchers from MIT, University of California and Berkeley, the main goal of Tapestry is to provide a high-performant, scalable and location-independent message routing system. Its main routing mechanism is the same as in Pastry by using prefix matching, presented by Plaxton et al. [PRR99] in 1999.

The main difference between the two overlay protocols is the handling of network locality. Tapestry constructs optimal routing tables with focus on local proximity of different nodes [LA10]. A further difference is the possibility of using data replicas to optimize performance, as described in [DGKW10]. In Tapestry the lookup mechanism distinguishes between several replicated items using metrics that can be defined for each specific application.

Similar to other P2P systems introduced before, Tapestry is self-organizing and also remains in a consistent state if nodes join or leave the network.

Further information: One implementation of Tapestry is *Chimera*[12]. It is a light-weight C implementation that uses functionality of *Pastry* and *Tapestry*. *OceanStore*[13]

[12] `http://current.cs.ucsb.edu/projects/chimera/` accessed: 2016-10-04
[13] `http://oceanstore.cs.berkeley.edu/` accessed: 2016-10-04

is a global persistent data storage system that is built upon the design and protocol of Tapestry [ZHS+04]. No application for a current mobile platform could be found.

2.2.5 Kademlia

Another decentralized P2P system that can be categorized as a structured P2P overlay network is *Kademlia* [MM02]. Its main features are consistency and performant queries within a distributed fault-prone environment by using an XOR-based metric topology.

Architecture and lookup mechanism: According to [LCP+05] and [CC10] Kademlia uses the same approach as other implemented DHTs, by assigning 160-bit random IDs to nodes and keys. Key-value pairs are stored on peers whose ID is close to the key. To lookup a key in the network, an XOR-based algorithm is used which is able to find an item in $\mathcal{O}(logN)$. Two nodes are further apart from each other if the result of the XOR operation between their node IDs is higher. For each bit i in the node ID, the routing table contains a separate list (called bucket), which stores contact information to nodes that differ in the i-th bit from the present node. All other first *n-1* bits are matching the ID of the present node. For example, the distance between the node with ID *1010* and a key with ID *1100* would be *0110*. Hence, the requested node would contact a subset of nodes of the second bucket next, because the present node ID and the key start to differ in the second bit. This approach is similar to the prefix matching approach used by Pastry (see Section 2.2.3) and Tapestry (see Section 2.2.4), but a Kademlia node sends the requests in parallel using asynchronous messaging. Another difference is that a requested node will not go on searching for the target node, but responds with a node ID that is closest to the searched key. The node that originated the request will then contact the nodes it has learned about and can so reduce the distance to a node by half for each iteration until the target node is found (see Figure 2.4). With the concurrent lookup mechanism, lookup latency can be shortened, but with the drawback of higher demand in the system.

Leaving and joining the network: In case a new node wants to join, it has to know one existing node within the network. The new node n first inserts the already known peer m into the appropriate bucket and afterwards performs a lookup request for peer m. The new node successively inserts contact information of other nodes based on the responses of lookup requests and at the same time fills up its node ID and address in buckets of contacted nodes. When a node leaves the network, no systematic mechanism takes effect. On a lookup request, a node initially creates a *PING* message to check if the peer is still alive - in case of unavailability the reference to the node in the specific bucket is simply removed.

Further information: As described, when a Kademlia node receives a message request or response from another node, it will update the sender's contact information. As a

result of that, peers handling a lot of messages are widely known and will therefore take on more workload in the network [LA10].

The protocol specified in the Kademlia paper [MM02] of 2002 was implemented with several different programming languages in a lot of of different projects. A very early Python application is for example *Khashmir*[14]. A quite young implementation from 2014, which is still in ongoing development, can be found on GitHub[15]. Here the developer claims to evade known Network Address Translation (NAT) problems by using Remote Procedure Calls (RPC) over User Datagram Protocol (UDP). This setup does imply, however, that message arrival is not guaranteed.

Further applications in production and in ongoing development are i.e. *RetroShare*[16] (a decentralized private and secure platform for filesharing, chatting and messaging) or *telehash*[17] (a decentralized and interoperable protocol which enables secure networking). *TomP2P*[18] is a Kademlia implementation in Java 6 that was also successfully tested on Android devices.

Moreover, popular implementations of unstructured P2P overlay networks (see Section 2.3) like *BitTorrent*, *Gnutella* and *Overnet* (a successor of *eDonkey*) use principles of Kademlia. IP addresses of peers are stored in a DHT and the put and get requests to insert or request those addresses are realized with the Kademlia algorithm. For example, client implementations of *BitTorrent* exist for all current mobile platforms. For more information on those network protocols and current development states see section *Unstructured P2P overlay networks* in the thesis of Jörg Schoba [Sch17a].

2.3 Unstructured P2P overlay networks

Unlike Structured P2P systems, several *unstructured* P2P overlay networks exist, where no algorithm controls the distribution of content and logic. More precisely, peers may join and leave the network without affecting any other peer's configuration or responsibilities, as it is the case for all *structured* networks reviewed above. A detailed discussion and evaluation on unstructured overlay networks and their implementations can be found in the same chapter in the thesis of Jörg ([Sch17a]), including *Napster*, *Gnutella*, *FastTrack*, *Freenet*, *eDonkey* and *BitTorrent*.

2.4 Coordination frameworks and models

Several frameworks could be found that facilitate the coordination of data flow in highly distributed networks, for example *DataSpaces* [DPK12], *TuCSoN* [OZ99], *DTuples*

[14] `http://khashmir.sourceforge.net/` accessed: 2016-10-04
[15] `https://github.com/bmuller/kademlia` accessed: 2017-01-18
[16] `https://github.com/RetroShare/RetroShare` accessed: 2016-10-04
[17] `http://telehash.org/` accessed: 2016-10-04
[18] `https://tomp2p.net/` accessed: 2016-01-18

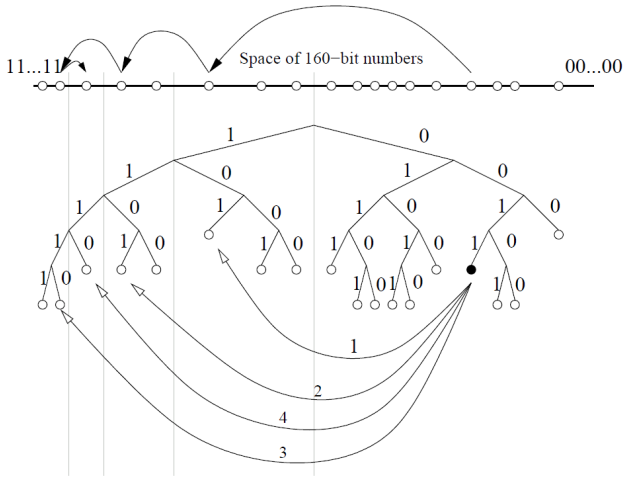

Figure 2.4: Example of a lookup request in the *Kademlia* overlay network [MM02]. The node with ID-prefix *0011* finds the node with ID-prefix *1110* by sequentially learning and querying closer nodes. Kademlia makes use of an XOR-metric to calculate the distance between two IDs.

[JXJY06] and *Comet* [LP05]. Many of them are based on *tuple-spaces* and a large part makes use of DHTs (Distributed Hash Tables), which were described in more detail in Section 2.2. In particular, reference is made here to Jörg's thesis [Sch17a], where these and other coordination frameworks are described in a more detailed fashion and also to [Alt16], in which Altschach describes dozens of tuple-space-based middleware systems in great detail.

2.5 General P2P frameworks and protocols

To conduct a complete research and analysis of existing technologies, also existing P2P frameworks and protocols were searched and examined. Here, focus is laid on higher-level distributed systems and those which only work in intra-process environments were excluded.

2.5.1 JADE - a platform for P2P agent based applications

The Java Agent DEvelopment framework (JADE) is an open-source framework for simplifying the development of interoperable multi-agent based applications. It was first introduced by Bellifemine et al. in 1999 [BPR99] and is a trademark of *Telecom Italia*[19]. Many applications in the research area are built upon the JADE agent-based platform (i.e. [UVG05]), but also mission critical applications in the telecommunication area (i.e. Italy and Britain) are implemented on JADE.

[19] `http://www.telecomitalia.com/` accessed: 2016-10-12

Characteristics and architecture

The conceptual model of the JADE platform contains on the first hand peer-to-peer networking and on the other hand the multi-agent paradigm. The communication of JADE is FIPA-compliant[20]. FIPA is a computer society standards organization, which intends to promote the interoperation of heterogeneous agents and the services that they represent. Agents can therefore operate across platform boundaries. The agent paradigm basically needs the peer-to-peer communication, because agents need to communicate with other agents to achieve their objectives. According to [BCP+03], an agent is an *autonomous*, *proactive* and *social* software component. Consequently, an agent may change its behaviour and take own decisions (autonomous), is able to take initiative (proactive) and in order to accomplish its tasks it has to interact with other agents (social).

Each JADE agent is located within a JADE runtime (*container*) and the set of all containers is called *platform* [BCP+03]. Each agent is able to perform basic service invocations like *discovery* of and *communication* with other agents. Furthermore, each agent has a unique ID and may implement a set of services, which can be provided to other agents. Asynchronous messaging enables temporal independence between two agents. If an agent is currently not available or even does not exist yet, the message is being cached in the container. Besides the agent abstraction, JADE provides a skeleton of typical interaction patterns, i.e. execution and composition of specific tasks including negotiations and auctions. Aspects that are not strictly related to application logic (i.e. synchronization issues, timeout handling, error conditions) can be neglected by the application developers.

To facilitate debugging and deployment phases, JADE includes graphical tool support to configure and control platforms and containers. Moreover, with the help of the so-called *sniffer agent* (GUI application) messages between agents can be eavesdropped.

According to Chmiel et al. in [CGK+05] the JADE agent platform is very efficient. They ran experiments with thousands of agents and could demonstrate that the processing time linearly increases with the number of agents or exchanged messages.

Implementations

In [UTG08] and [BCG14] sample applications for the Android platform were developed with the help of Add-ons, which contain additional classes and replacements for the JADE core module. Those Add-ons enable the deployment on Java-enabled cell-phones and the current version of the Android Add-on is also compliant with the core JADE. In contrast to previous versions, interaction between mobile and core agents is no problem any more. The so-called Lightweight and Extensible Agent Platform (LEAP) Add-on tries to optimize the communication mechanism by splitting up the JADE containers running on mobile devices into a front-end and back-end container. This improves

[20] `http://www.fipa.org/` accessed: 2016-10-13

high workload situations since the mobile client does not necessarily have to handle all incoming messages and with this approach additionally NAT traversal issues can be circumvented. However, a server part has to be implemented and as another drawback the resource-poor mobile device has to maintain a permanent bi-directional connection. On connection losses a re-establishment functionality is implemented, but no wake-up mechanism for long idle phases is implemented (see Section 5.6 and Section 6.6). An Android tutorial with sample applications can be found on the offical homepage[21]. No implementation for the iOS platform exists or is planned. However, interoperability with the currently available Android platform or other mobile platforms would be possible, due to the usage of FIPA compliant message communication. Nevertheless, the JADE platform is already rather complex and an adaptation would mean a great deal of time and human resources.

Furthermore, two more advanced agent-based platforms exist, which are based on JADE, namely Workflows and Agents DEvelopment framework (WADE)[22] and Agent-based Multi-User Social Environment (AMUSE)[23]. WADE is specialized for workflow and task execution applications and AMUSE is focused on distributed social applications, especially multi-player online games. Furthermore, the Android platform is supported by both frameworks.

In the last couple of years, a quite large community has gathered around the JADE project. The current stable versions of all three projects can be downloaded from the official JADE website. The software bundles are licensed under the GNU Lesser General Public License (LGPL), which is a corporate-friendly open-source license.

2.5.2 Juxtapose

Another generic open-source P2P framework is Juxtapose (JXTA). Jörg is performing a detailed evaluation of this framework in his thesis [Sch17a].

2.5.3 P2P communication using XMPP

The Extensible Messaging and Presence Protocol (XMPP) [SA05] was originally developed in 1999 as a project called *Jabber* by an open-source community. An advanced version of the protocol got standardized by the Internet Engineering Task Force (IETF) in 2002 with the name XMPP. In summary, it can be described as an application profile of the Extensible Markup Language (XML) protocol that facilitates near real-time communication to exchange extensible data between two or more network units. Because of its openness, flexibility and extensibility, it can be seen as a protocol of choice for internet real-time communication applications. The great built-in functionalities regarding security (authentication, access control, privacy) are another main advantage of the protocol.

[21] `http://jade.tilab.com/` accessed: 2016-10-13
[22] `http://jade.tilab.com/wadeproject/` accessed: 2016-10-13
[23] `http://jade.tilab.com/amuseproject/` accessed: 2016-10-13

In this section, the XMPP architecture, the core protocol and its functionalities, including communication primitives for messaging (see XML stanza), authentication, channel encryption and network availability, are discussed. A considerable number of client and server implementations have been developed in the last couple of years - popular ones are listed and reviewed in the last part of this section.

Architecture and core protocol

XMPP based networks make use of a distributed client-server architecture, see Figure 2.5. Here, clients can establish connections to several servers (which are also connected with each other) in order to exchange so-called XML stanzas. Message passing is implemented in an asynchronous way over direct and persistent XML streams. According to [SA11] the following steps have to be performed to open a stream:

- First of all the initiating entity has to open a connection (usually via Transmission Control Protocol (TCP)) to the server IP address and port by using an addressing scheme similar to that used for emails. Each XMPP entity (that can be a client, server or an additional service) has a unique address of the following form: *<localpart@domainpart/resourcepart>*. The *localpart* specifies a unique user account (i.e. *peter*) within a domain and resource. The *domainpart* defines the server name that can be resolved by a DNS, whereas also fixed IP addresses can be used. The *resourcepart*, as last part, is required if different services shall be hosted on one server or if concurrent server connections should be possible. For historical reasons this form of an XMPP address is called *full Jabber ID (JID)*, whereas the short form without the resourcepart is called *bare JID*. A possible *full JID* for a sample application may look like the following: *peter@tuwien-chat.at/chat*.

- The initiated entity then opens an XML stream (over *TCP*) by sending a stream header. After the receiving entity replied with a stream header response, a multi-stage negotiation process gets started. This process depends on the particular protocol interactions that the receiving entity (which must be a server after all) requires or provides. *Authentication* (i.e. via Simple Authentication and Security Layer (SASL)) is always mandatory, because services on a specific domain can be only used if the identity of the XMPP entity had been verified. Further optional protocol interactions are *application-layer compression* or *encryption* (i.e. via Transport Layer Security (TLS)).

- To properly address the previously authenticated client, the server has to bind a specific resource to the stream. This ensures that the client can send and receive XML stanzas to other registered clients within the same *resourcepart*.

- After a successful stream negotiation and resource mapping, each XMPP party can send and receive different kinds of *XML stanzas*, which are described in the next subsection.

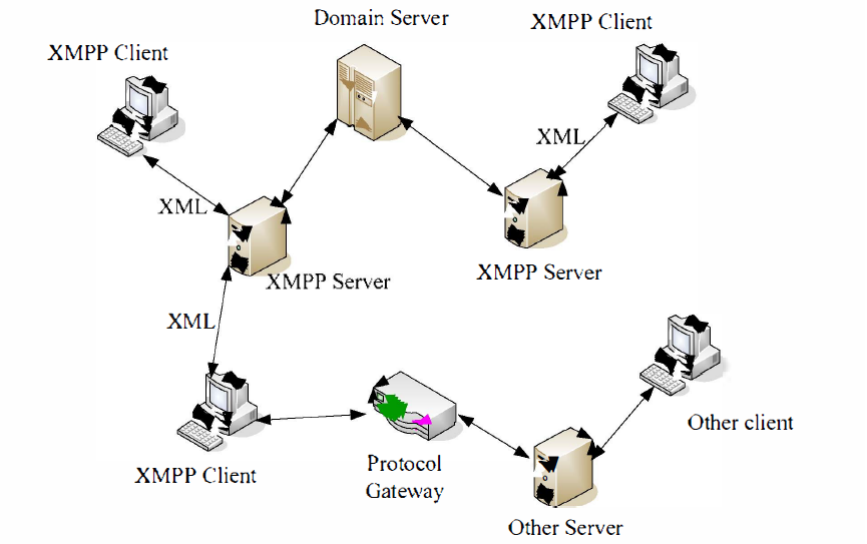

Figure 2.5: Example of an XMPP network [XM12]. The communiciation from client to client is logically peer-to-peer, but physically the data might be routed over different relay servers.

Under normal circumstances a connection gets closed by sending a *</stream:stream>* tag from the client to the server. Otherwise, if any unexpected stream errors occur, the connection is automatically destroyed.

In Figure 2.5 a potential XMPP network is shown. A client is able to exchange messages with another client over multiple XMPP servers, but can also transfer messages to non-XMPP instant messaging services via a *gateway server*, which is responsible for translating the XMPP protocol.

XML stanza

After a successful stream negotiation, two XMPP parties (i.e. client and server or two servers) can exchange three types of messages:

- **Message stanza:** The most commonly used stanza type is *message*. It is comparable with a *PUSH*-mechanism, where one entity pushes some information to another. The attribute *to* is necessary and assigns the desired receiver of the message. The attribute *from* can be specified by the sender, but must be exchanged by the server on message forwarding with the id of the actual sending entity to prevent message spoofing. A message stanza may include a generated value for the optional attribute *id*. In this case the response or error message for that particular message has to match the *id* of the originated stanza.
- **Presence stanza:** XMPP comes with a built-in availability information system for entities, especially clients. It is realized with a *publish-subscribe* mechanism, where

entity A, which subscribes for the status information of an entity B, gets notified if B changes its state. To signal availability, an entity sends an empty *presence* message to the server. To change the presence state or to manage subscriptions, an entity uses the *type* attribute with typical values like *unavailable*, *subscribe* or *unsubscribe*. Similarly to the *message stanza* the *id* attribute is optional.

- **IQ (info/query) stanza:** To enable a similar request-response approach like the Hypertext transfer protocol (HTTP), XMPP introduced *IQ stanzas*. Entity A may request information of entity B by sending an *IQ stanza* with a specific purpose (i.e. *get*, *set*) and a mandatory child element that specifies the semantics of this particular request. This data payload can be of an arbitrary type and must be defined by a schema or another structural definition. To match a response, the *id* attribute is mandatory for this message type.

XMPP extensions

The IETF is responsible for the design and development of the XMPP core specifications. In addition, the XMPP Standard Foundation (XSF), formerly the Jabber Software Foundation (JSF), develops XMPP extensions in its XMPP Extension Protocols (XEP) series. With those additional standardized protocols, more sophisticated but still interoperable applications can be developed. The official website[24] lists all XEPs, including obsolete, rejected, experimental, draft (appropriate for deployment, but minor changes are possible) and final (stable) versions.

At the time of writing (September 2016), 142 extensions were in the state *experimental*, *draft* or *final*. Some extensions that gained popularity in the last couple of years are for example:

- **XEP-0060: Publish-Subscribe** (State *Draft*) With this generic protocol extension XMPP gets the possibility to create topics at specific service endpoints where information can be published. Entities subscribed to a particular topic are then notified with a broadcast message on any service publications. A typical application using this kind of extension would be for example a news feed.
- **XEP-0030: Service Discovery** (State *Final*) This specification describes a robust protocol extension for determining features and information about other XMPP entities.
- **XEP-0045: Multi-User Chat** (State *Draft*) This extension enables XMPP entities to take part in a room or channel to communicate with multiple users, similar to a Internet Relay Chat (IRC). Beside the standard chatroom features (i.e. room topics and invitations) also room administrators and moderators can be nominated, which can control the room by requiring user registration and authentication or by kicking and banning users.

[24] `https://xmpp.org/extensions` accessed: 2016-10-22

- **XEP-0096: SI File Transfer** (State *Draft*) This protocol adds the functionality of exchanging files between entities. Also details about the transport negotiation and the exchange of file information is defined by this protocol.

Security

The XMPP core specification [SA11] also describes the basic security aspects for the communication. To achieve confidentiality and data integrity TLS (at the time of writing TLS 1.2 [Die08] is the stable and recommended version) should be used. By applying this security layer, each stream is encrypted, which makes eavesdropping and tampering impossible. The second security protocol described in the core specification is SASL, which ensures mutual authentication. In [Nie06] the detailed security setup process with TLS and SASL is described. In order to connect and exchange data with any other party, each participating entity must verify its identity. If only SASL is used (without encrypting the communication with TLS), packets of the authentication procedure could be intercepted and an attacker could record username and password. By using both layers of security, the communication is only truly safe if the server can be fully trusted. The reason for this is that the server holds different connections with two communicating clients (A and B). The server first decrypts the payload of entity A and is therefore able to store or tamper data before forwarding the message to entity B.

If the server part can not be trusted, end-to-end encryption is necessary. This means the payload is encrypted by the sending client and can be only decrypted by the receiver. In fact, if TLS is used as channel encryption, the server decrypts the message of entity A as before, but is no longer able to interpret the payload. One big disadvantage of XMPP is the lack of native end-to-end encryption support. However, several extensions regarding end-to-end encryption have been proposed and used in recent years, as it can be seen on the official XMPP website[25]. The first attempt, which was based on Pretty Good Privacy (PGP), was done in *XEP-0027 - Current Jabber OpenPGP Usage*. It got voted down to the state *Obsolete* in 2014 since it did not provide protection against replay attacks and messages were not signed[26]. Further effort to enable end-to-end encryption was taken in *XEP-0116 - Encrypted Session Negotiation* or *XEP-0200 - Stanza Encryption*. All of them are in development state *Deferred*. Another possibility to enable end-to-end encryption is Off-the-Record Messaging (OTR). This cryptographic protocol [otr] provides real *end-to-end encryption*, *deniability* and *forward secrecy*. More details regarding XMPP security and end-to-end encryption can be found in the thesis of Jörg Schoba [Sch17a].

Implementations

Numerous chat and IM network management applications, but also other mission-critical business applications, like real-time trading systems in the financial industry, are using

[25] `https://xmpp.org/extensions` accessed: 2016-10-22

[26] `http://wiki.xmpp.org/web/XMPP_E2E_Security` accessed: 2016-10-23

XMPP [XM12]. The main reason for that are the inherent security features in the core protocol and the decentralized architecture (similar to an email network), which allows people and organizations to take control over their own communication.

There exists a vast number of XMPP server and client implementations. The official XMPP website[27] is a good starting point for a quick comparison of existing popular open-source and proprietary XMPP servers and their implemented extensions. The Erlang Jabber Dameon (ejabberd)[28] server, written in *Erlang*, seems to support the most extensions and according to several internet forums and the book *XMPP: The Definitive Guide* [SASTT09] it is also very scalable, due to its clustering feature and the used concurrent functional programming language Erlang. A robust but easy to configure server implementation with a short learning curve is *Openfire*[29], which is written in Java. The official XMPP website also mentions established client products[30], including some for the Android and iOS platform. Furthermore, lots of code libraries are listed on the official XMPP website[31]. A popular library, fully working on the Android platform is for example *Smack*[32]. The so-called *XMPPFramework* is written in Objective-C and is suitable for programming XMPP client applications for the iOS platform. Beyond the XMPP core protocol it also supports more than 30 extensions[33].

2.5.4 P2P communication using SIP

Another popular protocol for peer-to-peer communication that also facilitates Voice Over IP (VOIP) and Instant Messaging (IM) is the Session Initiation Protocol (SIP). Jörg is describing this protocol in more detail in [Sch17a].

[27] `https://xmpp.org/software/servers.html` accessed: 2016-10-24
[28] `https://www.ejabberd.im/` accessed: 2016-10-24
[29] `https://www.igniterealtime.org/projects/openfire/` accessed: 2016-10-24
[30] `http://xmpp.org/software/clients.html` accessed: 2016-10-24
[31] `http://xmpp.org/about/technology-overview.html` accessed: 2016-10-24
[32] `https://www.igniterealtime.org/projects/smack/` accessed: 2016-10-24
[33] `https://github.com/robbiehanson/XMPPFramework` accessed: 2016-10-24

CHAPTER 3

Requirements on the MPM and selection of background technologies

In this chapter the requirements on the Mobile Peer Model are defined. To cover a lot of important requirements for mobile applications, some possible sample applications that could be built with the MPM were considered. Those include a simple messaging app, a distributed master worker example, which is also able to run in a background process without any user interaction and a multi-player game that has to communicate with multiple entities in real-time.

After defining the requirements, the technologies described in Chapter 2 are analysed and examined for suitability in accordance to these requirements (see Section 3.2).

The focus in this thesis is laid on general functional requirements and on those which are related to constraints found in a mobile environment. Nevertheless, all essential requirements, including those which are focussing on scalability and security, are listed here. The found technologies of Chapter 2 are evaluated against all imposed requirements, however, only those in my area of responsibility are described in more detail in the subsequent design and implementation chapters.

3.1 Requirements on the Mobile Peer Model

3.1.1 Functional requirements

Basically, the requirements are separated in *functional* and *non-functional* requirements. The functional requirements describe particular tasks or functions the system under construction shall be able to perform.

Coordination (FR 1)

The framework shall provide suitable concepts and components to support an application developer in coordinating the data flow and execution of business logic in a distributed system. At least the following functionality must be provided:

- Transport messages from one host to another.
- A message shall have a coordination-type for a proper classification.
- A message may hold an arbitrary payload.
- A message may be valid only for a specific duration.
- The developer shall be able to define rules that affect the execution of particular business logic. Such a rule may look like the following: *If one message of type A and two messages of type B are present -> execute service X.*
- The business logic may execute arbitrary code and may create new messages that are sent locally or remotely to initiate further tasks.

Running in the background (FR 2)

The framework should support building applications that can run entirely or partially in the background of the mobile device. In the first place this means that an application with no graphical user interface can be installed on a mobile phone that performs a particular background task (i.e. processing a specific work task or just calculating and transferring the current position for tracking purpose). Additionally, the framework should support that the runtime continues to run in the background, even if the user closed the application. The desired handling can be chosen by the application developer.

Autonomous startup (FR 3)

The application does not have to be explicitly started by a user, but may be started if a specific event occurs. Such an event can be device-specific (like a successful device start-up), but may also emerge from the outside world (like an incoming message). Especially the second one (event from the outside) is important, because on a resource-poor mobile device the application or its underlying P2P middleware should not run permanently in the background, or at least no permanent connection should be maintained all the time.

Decoupling from application (FR 4)

The framework shall be decoupled from an application that uses the framework. Furthermore, the library containing the framework shall be available as an independent module that can be imported in a standard Java or Android project.

Connectivity with local and mobile carrier networks (FR 5)

The framework should be able to communicate in any possible network constellation without further configuration effort taken by the end-user. In particular, this refers to

unrestricted functionality regarding communication in public W-LANs, where NAT is used. In addition, automatic re-establishment of the internet connection shall be managed seamlessly and as soon as possible by the framework.

3.1.2 Non-functional requirements

The second part of the requirements specification deals with *non-functional* requirements. In contrast to a functional requirement that specifies a particular behaviour, a non-functional requirement may describe constraints or desired quality characteristics on the system to build. Therefore, they are also known as *quality attributes*.

Licensing (NFR 1)

The license for the MPM framework, including the specification and the provided reference implementation, shall be of type *Copyleft*. In practice, this means that everybody has the right to access the publicly available source code and can modify and distribute that work, but with the restriction that the Copyleft license has to be preserved in any software derivation. As a result of that requirement, all libraries used in the MPM framework have to possess a similar or less restrictive license type (i.e. a similar protective Copyleft license like the General Public License (GNU) or more permissive ones like the MIT or Berkeley Software Distribution (BSD) license).

Scalability (NFR 2)

The system should scale in relation to participating users and their data sent over the wire. This means, that the system should be able to manage an increasing number of users and workload without losing effectiveness.

Security (NFR 3)

The framework should provide optional channel encryption to prevent eavesdropping and manipulation of data. Furthermore, end-to-end encryption can be activated to completely ensure private data exchange. In addition, Denial Of Service (DOS) attacks on a specific user shall be prevented by means of blocking concrete hosts in the network.

Simple API (NFR 4)

The Application Programming Interface (API) of the MPM framework shall be intuitive and meaningful, by using well defined interfaces for all components of the software.

Debugging and documentation (NFR 5)

The communication part shall be implemented as a two-way approach. One implementation should focus on performance by serializing the transferring data into a fast and compact binary format and one should facilitate debugging by preserving the data in a

human-readable form. Furthermore, the code, which is open-source by requirement NFR 1, shall be well documented. In particular, this means the complete documentation of interfaces, classes and their methods. Complex parts with more sophisticated functionality shall be supplemented with additional explanatory comments. As an additional part of documentation, well designed test cases shall be added in a separate test package.

Exchangeability of components (NFR 6)

As already described in requirement NFR 5, the framework developer should be able to decide the concrete communication implementation (performance or readability). By using interfaces, all important components shall be exchangeable by different implementations. Nevertheless, there should be a reasonable reference implementation of all components. Replaceable components are for example the communication layer, optional end-to-end encryption layer, serialization and deserialization as well as the persistence layer.

Design to work with a modeler (NFR 7)

The *API* of the MPM framework shall be designed in a way that a code generation tool or modeling application is able to generate the coordination specific part of an MPM application. Nevertheless, the code shall be clearly legible and also modifiable by an application developer.

Operability on popular mobile platforms (NFR 8)

The framework shall be designed in a way that allows its implementation on different mobile platforms. Those include in particular the current big players on the mobile OS market *Android* and *iOS*. Moreover, as already described in requirement NFR 5 a compatible data exchange between different platforms is of great importance.

Benefit in comparison to own implementation (NFR 9)

By using the provided framework the development of mobile P2P applications shall be accelerated significantly. Application developers shall be able to focus mainly on application logic, the communication and coordination part is abstracted by the framework.

Resource-efficient implementation (NFR 10)

The coordination framework shall not consume too much resources, including processing load and memory usage. Furthermore, if the app is running in the background and is in an idle state, no processing power shall be needed. Nevertheless, as described in requirement FR 3, the runtime shall be able to resume from an idle state to a running state by reacting to internal and external events.

Reliability (NFR 11)

The framework shall work in a failsafe fashion. This includes in the first place the correct and continuous execution of the MPM runtime if a mobile related event occurs (i.e. a phone call or a battery charging warning). Furthermore, in any other unexpected failure situation the data currently available in the framework shall be stored and reconstructed after the restart in a consistent manner. Such a situation may be the abrupt termination of the application by the Android platform because of resource bottlenecks or the shutdown of the device when the battery is empty.

3.2 Evaluation and selection of background technologies

From Sections 2.2 to 2.4, existing structured peer-to-peer overlay networks, P2P-frameworks and established communication protocols were introduced and analyzed. In [Sch17a] Jörg Schoba did the same with unstructured P2P overlay networks and further potentially helpful frameworks and protocols. In this section, those research results are evaluated regarding usability in the Mobile Peer Model. Finally, the actual technology selection process, which was also conducted in collaboration with Jörg, is described.

3.2.1 Structured P2P overlay networks

Although DHTs in structured P2P networks provide effective and quite reliable search capabilities, they fail entirely or suffer from different problems when it comes to multiple-keyword queries, including unbalanced load, hot spots, high network load and storage redundancy. Implementations exist that are built upon a keyword index approach as described in [JYF07] and further improvements with distance based pruning techniques for keyword lists as introduced by Kim in [KK07]. Nevertheless, structured P2P systems are not widely used, as the complexity increases drastically if effort is made to avoid problems in the protocol and additionally there is a lack of common use cases.

Structured P2P protocols also suffer from different security issues that are listed and discussed by Sit et al. in [SM02], which include lookup and storage attacks, denial of service attacks and problems that emerge if malicious nodes do not follow the protocol in the right way. Castro et al. in [CDG+02] reveals other problems regarding secure routing. However, this is beyond the scope of this thesis.

CAN: The main focus of CAN lies on large distributed storage management systems with fast retrieval and insertion of files. Furthermore, no ongoing development and no mobile implementation was found. Therefore, it will not serve as underlying technology for the framework.

Chord: Chord is a famous structured P2P overlay network that makes use of consistent hashing to achieve lookup resolution in $\mathcal{O}(logN)$. Also improvements to better support real-time communication systems were presented in [ZO09]. Nevertheless, the NAT traversal problem was still not solved with these enhancements and only a prototype

implementation for this version of the protocol exists. Additionally, only experimental versions for mobile platforms were found so that Chord is excluded as possible background technology for the MPM implementation.

Pastry/Tapestry: Two self-organizing decentralized P2P overlay networks are *Pastry* and the very similar *Tapestry*, which use prefix matching to route messages within their networks. Both protocols suffer from the NAT traversal problem and no mobile clients for any platform could be found, which implies the exclusion of the technologies for the MPM implementation.

Kademlia: Kademlia is the most popular member of structured P2P overlay networks in terms of implementations, possibly because of its low complexity and the performant lookup procedure. In contrast to other network representatives, the routing algorithm is conducted from one single node until the searched item is found and is therefore easier and quicker to implement and to debug. Secondly, parallel and asynchronous messaging is reflected by higher lookup speed and the preferability of choosing faster and long-standing nodes over new ones also prevents attacks (i.e. by ignoring a new malicious node). Two quite promising ongoing implementations of the protocol (*kademlia*[1] in Python and *TomP2P*[2] in Java) also tackle the NAT traversal problem by using RPCs over UDP. This setup, though, implies that message arrival is not guaranteed, because of the unreliable and connection-less communication characteristic of the UDP. However, since this is a fundamental requirement on the Mobile Peer Model this disqualifies the protocol as a background technology for the framework implementation.

3.2.2 Unstructured P2P overlay networks

In [Sch17a] Jörg Schoba went through the same process by first presenting and then evaluating P2P overlay networks, where his focus lay on unstructured ones. No popular player, including *Napster*, *Gnutella* or *BitTorrent*, could comply with the majority of the imposed requirements on the MPM. Reasons for this were similar to the ones presented in the previous section, such as no ongoing development, security vulnerabilities, additional firewall and network configurations, proprietary licenses or lack of open-source implementations.

3.2.3 Coordination frameworks and models

In Section 2.4 some frameworks and models have been presented that facilitate the coordination of data flow in distributed systems. A more detailed analysis has been conducted by Jörg in his thesis [Sch17a]. Also the comparison and evaluation of the introduced frameworks and models can be found in his work. However, the reasoning for the selection of the underlying coordination model of the framework is described in the last section of this chapter.

[1] `https://github.com/bmuller/kademlia` accessed: 2017-01-18
[2] `https://tomp2p.net/` accessed: 2017-01-18

3.2.4 JADE

The Java Agent DEvelopment framework can comply with a lot of requirements imposed in the previous section. The framework is open-source and has a corporate-friendly license. In addition, JADE has gathered a fairly large community and some further platforms that are built upon the core framework have been developed, implying that the framework is usable and reliable. Furthermore, implementations on Android are possible, whereas also the NAT traversal problem can be circumvented by splitting up JADE containers into a front-end and back-end container. With that relaying approach, however, a server part has to be implemented.

Drawbacks are in the first place the complexity of the framework, the resulting steep learning curve and the usage of the programming language *Java*, which means that iOS is currently not supported. However, adding an iOS implementation would not imply a radical communication refactoring, because JADE is Foundation for Intelligent Physical Agents (FIPA) compliant. Therefore, JADE is a potential candidate for the final selection of technologies.

3.2.5 XMPP

The *Extensible Messaging and Presence Protocol* is known for its openness, flexibility and extensibility, making it a protocol of choice for internet real-time communication applications.

XMPP is compliant with all the requirements on the MPM. The high significant requirement FR 5 (3.1.1) in regard to availability within a public W-LAN (NAT traversal) and all other network configurations can be fulfilled with the XMPP relay server. The server also provides identity management, blocking of specific users and group chats. Concerning the requirement of scalability (NFR 2) XMPP supports server clusters. Furthermore, many open-source server and client implementations exist and the core protocol defines important security measures, namely channel encryption (TLS) and mutual authentication (SASL). In summary, XMPP is a very promising candidate as an underlying technology for the framework (see the final selection in Section 3.2.9).

3.2.6 JXTA - A general purpose P2P framework

As described in more detail by Jörg Schoba in [Sch17a], Juxtapose (JXTA) is *"a candidate to base the framework on"*. The NAT traversal problem can be solved, adequate security measures have been implemented and a successful adaptation for the Android platform has been performed. Nevertheless, the final selection decision is discussed in Section 3.2.9.

3.2.7 SIP and P2PSIP

Details about SIP and the peer-to-peer version of that protocol (Peer-to-Peer Session Initiation Protocol (P2PSIP)) were introduced by Jörg Schoba. Although it first looked

	CAN	Chord	Pastry	Tapestry	Kademlia	JADE	XMPP
FR 1 - Coordination	-	-	-	-	-	≃	-
FR 5 - Connectivity	-	≃	-	-	≃	+	+
NFR 1 - Licensing	-	+	+	+	+	+	+
NFR 2 - Scalability	+	+	+	+	+	+	+
NFR 3 - Security	≃	≃	≃	≃	≃	≃	≃
NFR 8 - Operability	-	≃	-	-	≃	+	+
NFR 10 - Resource-efficient	≃	≃	≃	≃	-	-	≃

Table 3.1: Probable fulfilment of requirements by the presented P2P technologies.

as a promising candidate, due to Android providing a built-in SIP API[3] and several existing clients on iOS, it could not make it into the final selection (for more details check the section *Evaluation and selection of background technologies* in the partner work [Sch17a]).

3.2.8 Overview of technologies and fulfilment of requirements

Table 3.1 contains a short summary about whether the presented technologies of Chapter 2 can fulfil the most important requirements of the framework. *(+)* means that the requirement can be fulfilled, *(≃)* stands for a partial fulfillment or that this requirement could be met with an acceptable additional effort and *(-)* denotes that this requirement can not be fulfilled with this technology.

Only with JADE it is possible to control the execution of business logic by defining the behaviour of *agents* (FR 1 Coordination). However, the definition of rules as required by FR 1 is not possible. The important requirement *Connectivity* (FR 5) can only be fulfilled by JADE and XMPP. Also only those two technologies have successful implementations for current popular mobile operating systems (NFR 8). All presented protocols and frameworks are scalable (NFR 2) and all of them have implemented some security countermeasures (NFR 3). However, none of them provide real end-to-end encryption. Furthermore, none of the technologies is designed explicitly for mobile devices (NFR 10).

Summarizing, there is no technology that meets all requirements and therefore an own solution has to be implemented. To reduce the implementation effort, suitable existing technologies shall be used.

[3] `https://developer.android.com/guide/topics/connectivity/sip.html` accessed: 2016-01-18

3.2.9 Final selection of technologies

After the elimination of most of the discovered technologies, the final selection was performed in consultation with Jörg. Since XMPP is compliant with all requirements on the MPM and the protocol provides a lot of useful built-in functionalities, XMPP has been selected as baseline technology. To recapitulate the main advantages from Section 3.2.5, the protocol provides identity management, blocking of specific users, group chats, server clusters and optional channel encryption. Several server and client implementations exist, both open-source as well as proprietary ones. Furthermore, P2P communication works in every network configuration, because the XMPP server acts as relay and routes messages from one peer to the other.

Although JADE and JXTA have reached the final selection phase, they were not selected as technology to build the framework on. Both frameworks were designed for desktop and server applications. Even though adaptations for the Android platform exist, the frameworks are too heavy-weight and would require more memory space and computational power.

XMPP does not limit any requirement on the MPM. The only drawback of XMPP that could be found during the final evaluation process is the dispensable payload sent over the wire. First of all, XMPP is a text-based protocol, which means there is a network overhead compared to pure binary solutions. Additionally, XML uses so-called *tags* for encapsulating data, where the closing tag is redundant. However, that minor drawback is negligible, as the main focus does not lie on transmission speed and rate.

As underlying coordination model, the Peer Model (PM) has been chosen. This model provides a high-level programming abstraction, facilitates software reuse and strictly separates business and coordination logic. With the constructs of the PM it is possible to model the data flow and the execution of business logic as required by FR 1 (see Section 3.1.1). Furthermore, the model is relatively easy to understand and an adaptation to a mobile profile seems definitely feasible. In addition, the thesis advisor and assistant, Eva Kühn and Stefan Craß, were primarily responsible for the design of the PM. Therefore, they can provide support when elaborating and defining the reduced, mobile version of the model. For more details on the original model see Chapter 4, a deeper insight into the mobile profile is given in Section 5.2.

CHAPTER 4

The Peer Model

To cope with the complexity of modeling and designing distributed and concurrent software systems, several coordination models have been developed in the last decades. Many of them are focusing on the coordination part and abstract away the communication aspects of the system. In Section 2.4 (further coordination frameworks and models) some popular frameworks are listed. In addition, Eva Kühn et al. examined some models in [KCS15], for example *Petri Nets* [KCJ98], *Reo* [Arb04], the *Actor model* [HBS73] and *Web Service - Business Process Execution Language (WS-BPEL)* [JEA+07].

However, all of the approaches mentioned above have some drawbacks, i.e. too low-level programming abstraction, insufficient support of software reuse or mixing business logic with coordination logic. Therefore, another concurrent coordination model has been developed at the TU Wien in 2013. The so-called *Peer Model* [KCJ+13] and its Domain-Specific Language (DSL), presented by Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek and Thomas Scheller, facilitate the design and implementation of complex integration patterns in large distributed environments.

This chapter is devoted to the model presented in 2013, which got extended and improved until 2017. In the next chapter the scope and functionality of the reduced mobile version of the Peer Model is introduced (see Section 5.2).

4.1 The Peer Model

The design of the Peer Model is inspired by tuple-space communication, data-driven workflow and a staged event-driven architecture [KCH14]. In a nutshell, the main components of the system are structured, re-usable and addressable constructs called Peers. One Peer has an input and an output space container, which are used to store data. The data is encapsulated in so-called Entries and can be transported between different

containers via an internal coordination mechanism (Wiring). Only communication and coordination parts of the system are described in the model, but no application logic.

4.1.1 Entry

Data objects in the Peer Model are represented as *Entries*, which may represent an event, a message or a request. They are a coherent entity, consisting of an *application*- and a *coordination*-specific part. The coordination part holds necessary system meta-information and has the following properties:

- **type:** The coordination-type of this Entry. It is used to query and select Entries from a container and it is used to control the coordination flow in the system.
- **origin:** The URI of the Peer that created the Entry.
- **from:** The URI of the Peer that sent this Entry to another Peer.
- **dest:** The URI of the Peer that this Entry is sent to.
- **tts:** The *time-to-start* property, which defines from which point in time the Entry is valid.
- **ttl:** The *time-to-live* property, which defines for how long an Entry is valid.
- **flow identifier:** The *flow identifier* property is part of each Entry in a specific global task (workflow). More details about flows can be found in Section 4.1.5.

Application specific properties include:

- **data-type:** The data type of the application object stored in that Entry.
- **data:** Defines the actual data object of that Entry.

Consider the two different *type* and *data-type* properties. The first one is for coordination purpose and the second one defines the actual type of the data object encapsulated in that Entry.

4.1.2 Space and Container

The Peer Model uses a space-based middleware, which is used to store and query data of the system. As described in [CKS09], the space consists of shared containers that provide a flexible API for writing and fetching Entries. More precisely, Entries can be fetched and removed (*TAKE*), fetched without removal from a container (*READ*) and written into a container (*WRITE*). A container supports different coordination principles that define in which order Entries are fetched (i.e. *First-In-First-Out (FIFO)*, by *key*, by *template matching* or by an *Structured Query Language (SQL)-like query* statement). Each *TAKE* operation may also contain the Entry's *coordination-type* property to restrict the Entry by type and a parameter to define the quantity of Entries to be fetched. These are the *exact* number (=), a *minimum* (> or >=) or a *maximum* (< or <=) of desired Entries.

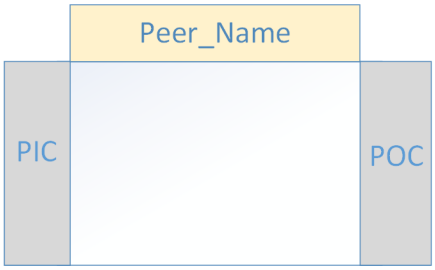

Figure 4.1: The graphical notation of a Peer without any components [KCJ+13].

The semantic of = should be clear, >= means all Entries of the container but at least n and <= stands for retrieving a maximum of n Entries. Further features provided by the space-based middleware are the support for transactions and bulk data processing.

4.1.3 Peer

A *Peer* is the main component of the Peer Model (see its notation in Figure 4.1) and is addressable in the network by a unique Uniform Resource Identifier (URI) with the format *<protocol>://<host-name>/<peer-name>*. It is constructed with the following elements:

- **Containers:** For storing Entries, a Peer holds exactly one Peer-In-Container (PIC) and one Peer-Out-Container (POC), which are referenced by a unique *URI*.
- **Wirings:** The active parts of a Peer are its Wirings. In a nutshell, they control which Entries move through the Peer and which Services are executed. The behaviour and the tasks of a Wiring are described in detail in Section 4.1.4.
- **Services:** During the execution of a Wiring, additional business code can be executed, which may be split up in several *Services*.
- **Sub-Peers:** Within a Peer several Sub-Peers can be created in order to divide the functionality into different parts.

If the Peer receives an Entry from another Peer, it is stored in the single PIC of that particular Peer. Those Entries can then be moved to PICs and POCs of different Peers and Sub-Peers by executing the Peer's Wirings (*internal collaboration*). Additionally, *inter-peer collaboration* takes place between the POC of a global Peer (not a Sub-Peer) and the PIC of another global Peer. All existing Peers on one particular host (the *Peer-Space*) are placed in a special runtime environment, called the *Runtime-Peer (RTP)*. Its API also provides the dynamic creation and deletion of Peers and Wirings.

4.1.4 Wiring

Wirings are the only active parts of the system and are responsible for the transport of Entries between containers. A Wiring is composed of the following three sections:

- **Guard-Links:** The first part of a Wiring is responsible for reading or taking Entries of containers. The coordination-type of the desired Entries and a count can be specified and when the Wiring is executed, it tries to fulfill all defined Guard-Links. Those Entries are then added to a temporary space container, termed Entry Collection (EC). If at least one link can not be satisfied, the Wiring execution is stopped.
- **Service (optional):** A Wiring can have zero, one or more Services. In the case that all Guard-Links can be satisfied, the constructed EC constructed is passed to the first Service of that Wiring. A Service may change the EC by removing or adding new Entries and hands over the EC to the next Service. After the completion of the last Service or if there has not been a Service, the EC is passed to the third and last part of the Wiring.
- **Action-Links (optional):** The Action-Links of a Wiring are responsible to write Entries from the previously constructed or filled EC into specific target containers of local or remote Peers. If the Entry shall be inserted into a local Peer, the target container has to be specified explicitly. In the case that the Entry shall be sent to a remote Peer, the destination property has to be set with the Peer-URI of the target Peer.

Each Wiring is associated with a transaction. On successful Wiring execution, the transaction is being committed and all Entry changes take affect. Furthermore, it can be configured that locked Entries are committed already after the Guard or Service phase. Those *early commits* can improve concurrency by releasing Entries for other Wirings that are waiting for particular locked Entries. Generally, an arbitrary number of Wiring instances can run in parallel, enabling very high concurrency. In reality there is a limit for parallel executions, which can be realized by configuring a thread pool size. By the design of the Peer Model, a Wiring connects a Peer with a Service in a dynamic fashion and data-driven way, which enables high decoupling. Furthermore, the Service implementations are provided by the developer to separate business logic explicitly from the coordination logic.

In Figure 4.2 a Peer with the name *SamplePeer*, two Wirings, two Services and one Sub-Peer are shown. The Wiring *Wiring1* contains two Guard-Links, the filled circle represent a Link that *takes* exactly two Entries with the type T from the PIC, the unfilled one represents a READ operation with at least one Entry of type U. In the case that all specified Entries are available in the PIC, the Entries are added to a newly created EC and the two Services are executed in sequential order. The EC is passed to the first Service, which can read and take Entries from the EC or write new Entries to it. As it can

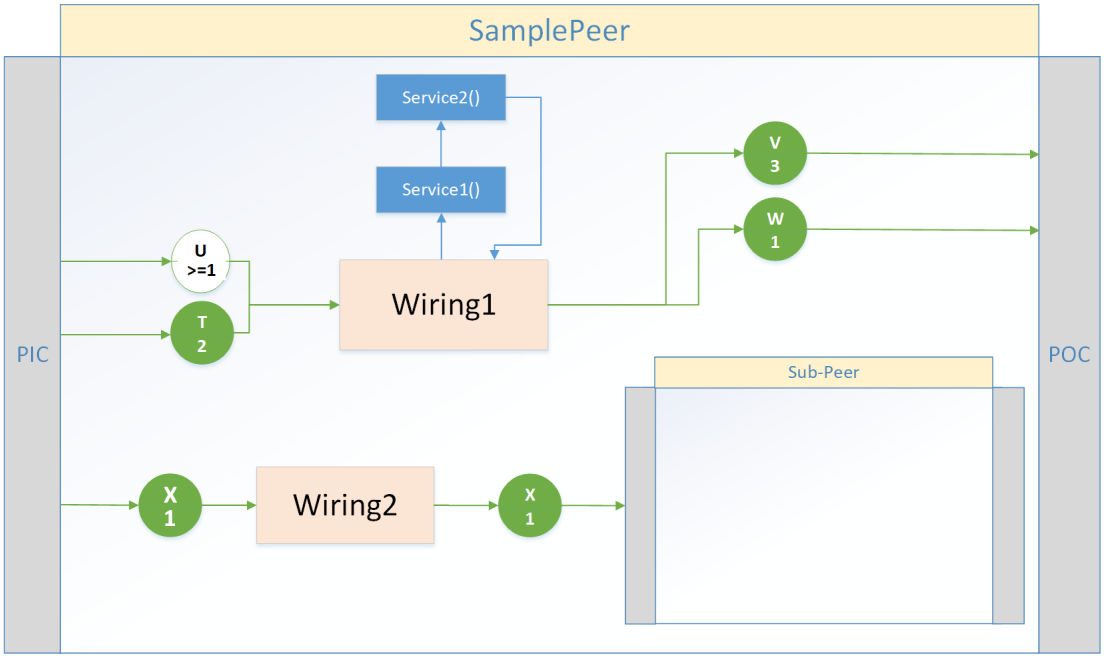

Figure 4.2: The graphical notation of the Peer *SamplePeer* with two Wirings *Wiring1* and *Wiring2*.

be seen in Figure 4.2, the logic of the Service is not modeled. Nevertheless, it is assumed that the Wiring's Action-Links are performed and three Entries of type V and one Entry of type W are written into the POC of the Peer after successful Service executions. If any Service was not working correctly, not all Action-Links might be performed successfully and for example only the Entry of type W would be written into the POC, whereas all three Entries of type V were skipped. However, the Wiring's transaction would be committed anyway, as the whole execution is considered to be successful. The second Wiring (*Wiring2*), *takes* exactly one Entry of type X and forwards it to the PIC of the Sub-Peer without executing a Service. The Sub-Peer may have additional Wirings that were not modeled here due to lack of space.

4.1.5 Flow identifier

A number of consecutive Wiring executions in not necessarily the same Peer-Space can be seen as a workflow that, collectively, solves a global task. To facilitate the realization of a workflow, each Entry can have a *flow identifier*. As described in [KCH14], all Guard-Links of a Wiring then only *read* or *take* Entries with the same flow identifier or Entries without a flow ID.

4.1.6 Further enhancements of the model

In [CJK15] Craß et al. present a flexible access control model to enable authorization of autonomous peers in a distributed system. Furthermore, it is shown how this model can be integrated into the architecture of the Peer Model.

Another remarkable concept, extending the Peer Model, was presented in [KCS15] and enables the design and implementation of distributed systems via generic, flexible and reusable coordination patterns. The concept introduces an extension mechanism where generic patterns can be configured at the time of deployment. In a nutshell, a pattern is reusable, parameterizable and can be composed of other patterns. As a proof of concept a *MapReduce* framework was built using different basic patterns, which also demonstrate the scalability of that approach.

CHAPTER 5

Design

Subsequent to the evaluation and selection of background technologies as well as the presentation of the underlying model that will serve as a fundamental coordination basis for the framework, this section describes the design phase of the software engineering process.

In the first part of this chapter a reduced mobile profile of the Peer Model and its characteristics are presented (see Section 5.2). In the second part important design decisions regarding the MPM framework are discussed. More precisely, the overall architecture of an MPM distributed system (including necessary system hosts as the Notifier-Peer and the registration application, see Section 5.3.3) is presented, followed by the architecture of a single MPM host (including important components as the Runtime-Peer, Peers and Wirings, see Sections 5.4 and 5.5). The last part of the design chapter deals with mobile design considerations (Section 5.6), including user interaction and persistence and possible code generation using a modeler (Section 5.7).

5.1 Distribution of work

The whole engineering process is conducted in close collaboration with Jörg Schoba [Sch17a], as it has been the case in Chapters 2 (analysis of existing background technologies) and 3 (requirements on the Mobile Peer Model). In this work the main focus regarding design and implementation is laid on the architecture of the Runtime-Peer (RTP), mobile constraints, user interaction and persistence. Jörg focuses on the communication and serialization and provides deep insights into scalability and security considerations.

5.2 The Mobile Peer Model

In Chapter 4 the specification of the Peer Model from 2013 was presented. The coordination model, introduced by Kühn et al., has been designed and developed for applications in the desktop and server area. The focus was laid mainly on complex and highly concurrent applications with high throughput capacity. In mobile environments, however, other requirements have higher priority, for example lower resource consumption. Also some assumptions cannot be applied to a mobile environment, like a reliable and continuous internet connection. Therefore, a reduced profile of the Peer Model was elaborated in cooperation with the designers of the original Peer Model. In several meetings with Eva Kühn and Stefan Craß some functionalities of the original model have been eliminated, whereas also some new mobile supportive features were added.

In the following subsections the main components of the adapted *Mobile Peer Model* are presented. Worth mentioning is that the feature decision was made in consideration of some mobile applications (i.e. a distributed master worker example or an online interaction multi-player game). The resulting profile was elaborated in a detailed fashion, nevertheless, it might be enhanced with further features in the future.

5.2.1 Entry

As described in Section 4.1.1, Entries represent data objects in the Peer Model that might be an event, a message or a request. In contrast to an Entry in the original Model, flow identifiers are not supported in the first version of the mobile profile. Furthermore, the mobile version supports two system-defined Exception-Entry types, namely the ones for Time to live (TTL) expirations and potential send exceptions.

TTL expiration

An absolute timestamp in milliseconds since 1970-01-01 (Universal Time Coordinated (UTC)) defines the point in time when an Entry loses its validity (TTL property). The default value for that property is *-1*, meaning the Entry is always valid. Relative timestamps are not applicable in a mobile environment with a relay server, because the server can cache Entries, which means that the relative time might not be valid anymore on the receiving side. It is assumed that all clocks are synchronized.

An Entry can expire while it is stored in the PIC of a Peer or during a Wiring execution. In such a case, the Entry gets wrapped into an Exception-Entry with type *exception* when it is read or taken from a container or EC. Secondly, an Entry can expire during the send process. In this case, the Entry gets wrapped into an Exception-Entry when it is written into a container of the receiving Peer. Furthermore, an Entry may expire while it is stored on a persistent storage. When this happens, the Entry gets wrapped into an Exception-Entry while the Peer's containers are filled on RTP startup. More details about exceptions and how developers can handle them can be found in Sections 5.5.6 and 6.3.6.

Potential Send exception

The second possible exception type of an Entry is *POTENTIAL_SEND_EXCEPTION* and may occur while an Entry is sent to a remote host (more details can be found in Sections 5.5.6 and 5.6.7).

5.2.2 Space and Container

Also the MPM uses concepts of space-based computing, whereas the shared memory in the MPM framework is simply called *container*. Those containers can be described as a repository of Entries which are accessible in a concurrent manner. In comparison to the specification of the original Peer Model, which assumes a tuple space-based communication middleware to be present, no existing implementation is used. The main reason is that existing middleware aims at processing speed and high concurrency, which also implies more resource consumption. By designing the container technology and its behaviour by ourselves, the space-based shared memory can be optimized for mobile and light-weight devices.

Like in the Peer Model, a container offers three operations (*READ*, *TAKE*, *WRITE*). A *count* argument can be specified to retrieve *exactly n* (n >= 1), *greater or equal n* (n >= 0) or *less or equal n* (n >= 1) elements. When fetching an Entry, the *coordination-type* of the Entry has to be specified, which is the only possible selection criteria. Other space-based middleware implementations often also offer more sophisticated selection opportunities (selectors). Another feature that is not provided for GET operations is the specification of a coordination principle, which defines the order of fetched Entries (i.e. FIFO), as for example the EXtensible Virtual Shared Memory (XVSM)[1] does. Transaction support is not needed by the container in the MPM, because a Peer does not support the concurrent execution of Wirings.

Further details on containers can be found in Section 6.3.1, where the main focus is laid on the actual API and the implementation details.

5.2.3 Peer

In contrast to the original model presented in Chapter 4, a Peer in the MPM can not have any Sub-Peers. As a result of that, also no POC is necessary, because Wirings between different Peers are not supported. However, apart from that restriction, a Peer has exactly one PIC and holds a list of Wirings, which may transport Entries and execute external Services, as it is the case in the original model.

Every Peer within the Runtime-Peer has a unique URI, which is of the form *<protocol>://<host-name>/<peer-name>*. As the communication layer shall be exchangeable, the first part of the URI shall define the used protocol. The second part (*host-name*) defines the actual user or host running an application that uses the MPM framework. Finally,

[1] `http://www.mozartspaces.org/` accessed: 2016-10-01

the last part of the URI (*peer-name*) identifies a specific Peer running on that host. Consequently, every Peer has a globally unique URI. Although a Peer in the MPM has actually no POC, the same graphical notation for a Peer is used as shown in Figure 4.1. An Action-Link that is pointing to the POC of the Peer illustrates an external Action-Link, meaning that the specified Entries will be written into the PIC of another local Peer or sent to a Peer of a remote host, according to their *dest* property.

5.2.4 Wiring

As described in the original paper of Kühn in [KCJ+13] *"... wirings are the only active part of the system."* In comparison to the original Peer Model, the mobile profile has some reductions regarding Wirings.

The original model allows the concurrent execution of Wirings, even a single Wiring can be executed in parallel within a Peer. Because mobile devices have restricted processing power on the first hand and restricted battery life on the second hand, in the MPM only one Wiring per Peer is allowed to run at the same time. Consequently, a Peer gives control of the current process (or thread) from one Wiring to the next, meaning no concurrent executions are possible on the Wiring level. Furthermore, one Wiring can only have exactly one Service associated. However, to realize multiple serial Service executions, as it is possible with the original Peer Model, all Services only need to be called from one single Service, that passes the returned EC from the previous Service to the next.

In addition, *flows* (see Section 4.1.5) are not supported by the first version of the mobile profile. On Entry retrieval in *Guards* or *Actions* only the Entry's type and no flow identifier are considered. To model the behaviour of a workflow in the Mobile Peer Model, the developer or modeler of an application has to emulate flows with a flow identifier encapsulated in the Entry's data object. Nevertheless, then also an advanced Service implementation, which handles the logic for fetching only Entries with a specific flow ID, would be needed. For a Wiring the same graphical notation as for the Wiring shown in Chapter 4 is used.

In newer publications of Eva Kühn (i.e. in [Küh17]) the notation *Guard* is synonymous for *Guard-Link*. However, in this thesis a *Guard* is used to describe all *Guard-Links* of a Wiring. The same notation applies for Action-Links. Therefore, in the course of this work, an Action may be constructed of several Action-Links.

5.2.5 Runtime-Peer (RTP)

The so-called Runtime-Peer was already described in the first paper of Kühn in [KCJ+13]:

> *"Together, all peers of a site form a "Peer Space", whose runtime environment is bootstrapped via a runtime peer (RTP), the name of which refers to the URI of the local site."*

This definition can be completely adopted. As described in the quote above, the RTP is the main component of the MPM and holds all user-added Peers of a local side and also system-defined Peers, like the IO-Peers (see Sections 6.3.4 and 6.3.5) and the Exception-Peer (Section 6.3.6). In addition, the RTP exposes an interface that can be used by the surrounding environment to start and stop the runtime or to inform the RTP about events from the outside (for example when the internet connection has been established on a mobile device).

5.3 Overall system architecture

In this section important system components of the MPM framework are presented. As the framework is developed to work in a mobile environment, the communication part between volatile mobile peers is quite challenging and needs some additional components accordingly. Discontinuous availability has to be considered by providing the seamless re-establishment of the communication with a participant. Due to requirement *NFR 1* (see Section 3.1.2), an unrestricted and reliable communication that is working properly in every network constellation is needed. As described in Section 3.2 (selection of background technologies), XMPP was chosen as communication protocol, which requires a relaying server to route messages. In this scenario every peer has to establish the connection on its own initiative. Due to the fact that applications on mobile devices should not run permanently in the background in order to save processing power and to reduce battery consumption, the devices also need some kind of notification approach to wake the application up if another peer has sent some data.

Figure 5.1 shows two application-specific hosts (Host 1 and Host 2) and all system components in the MPM reference implementation that are needed to guarantee a reliable communication. Those include the central XMPP server, the registration server and the Notifier-Peer, which are in the responsibility of an MPM developer, and the third party Firebase Cloud Messaging (FCM) server that is needed for sending notifications to mobile devices. More details about those components are supplied in the following subsections, more details about the communication layer is provided in the thesis of Jörg [Sch17a].

5.3.1 Communication between peers

The two components in the upper section of Figure 5.1 represent two hosts in an MPM application. Each of them might be a mobile device, a desktop PC or a server component (more details about the different execution environments of an application built upon the MPM framework are presented in Section 6.1).

If *Host 1* wants to send data to another host in the network, it first needs to establish an authenticated connection with the *XMPP server*. For now, it is assumed that Host 1 is authorized to communicate to the server (Section 5.3.3 deals with new users). Host 1 then sends the actual message to *Host 2* via the XMPP server (message M1). In case the

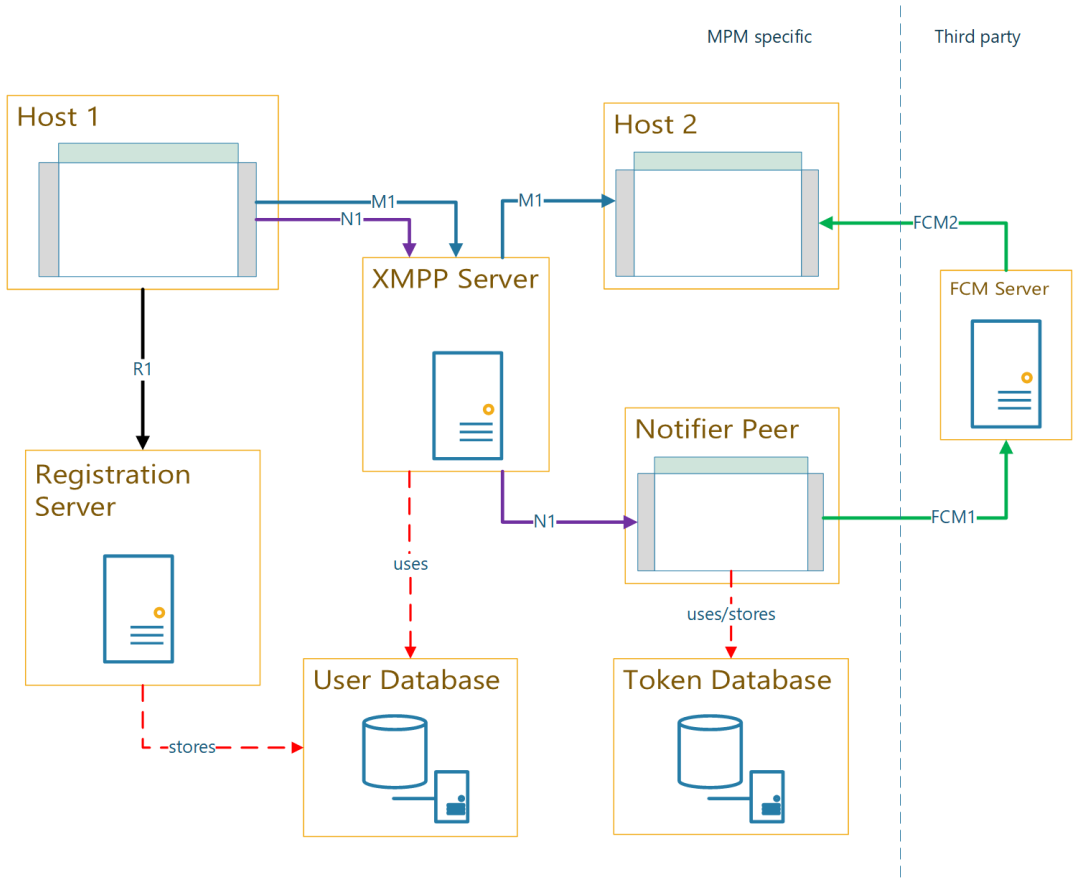

Figure 5.1: The overall architecture of an application that uses the MPM framework with two hosts and the mandatory system components (XMPP server, registration server, Notifier-Peer and the FCM server).

receiving peer Host 2 is currently also connected to server, the message is immediately forwarded and the peer can do its work. This direct communication is marked with blue arrows in Figure 5.1.

5.3.2 Notifier-Peer

However, in case that the receiving peer Host 2 is not connected to the XMPP server, the actual message of Host 1 is cached on the server until the receiver comes online again. Because a receiving peer does not know when a new message is delivered, the Notifier-Peer component is used to wake up the receiver of a message. With that approach repeating reconnections to the XMPP server (active polling) can be avoided, which would in the first place increase battery life of the device and in the second place is not as accurate as the notification approach, because a message might be sent exactly between the polling attempts.

In order to be able to notify a member in the network, the Notifier-Peer has to maintain a list of hosts (in the reference implementation a host is identifiable via its XMPP username) and its associated unique FCM token (see Section 5.6.5). Therefore, each mobile host has to deliver its FCM token to the Notifier-Peer whenever the token changes or the host connects to the XMPP server for the first time. More details about FCM push notifications and tokens are discussed in the design chapter (Section 5.6.5) and in the implementation part (Section 6.6.3).

Let's assume that the notifier component is running and the FCM token of host Host 2 is available. Therefore, the Notifier-Peer is ready to receive a notification message with the information about which peer has to be notified. This notification message *N1* is sent at the same time as the actual message *M1* and is immediately forwarded to the Notifier-Peer, which is assumed to be always online and might also be redundantly deployed. With the username contained in message N1, the Notifier-Peer performs a lookup for the associated FCM token of the receiving peer Host 2. That token is then sent via an HTTP request to a Google Firebase Server, which in turn sends a notification to the receiving device Host 2. Subsequently, a callback method will be invoked by the operating system on the receiving device Host 2, forcing the application to reconnect to the XMPP server to receive the cached message (more details about this callback method and FCM can be found in Sections 5.6.5 and 6.6.3).

The main advantage of this notification approach is that a device only has to maintain one single long-living connection to the FCM server.

A reliable communication is only given with at least one Notifier-Peer deployed. Nevertheless, in a mobile environment there are several other possible circumstances why a host cannot be notified at a specific point in time, for example if the mobile network is temporary unavailable, the phone is turned off or the battery is empty. However, after the successful reconnection to the FCM server, a device will receive pending push notifications and will subsequently start the RTP to receive incoming messages.

5.3.3 Registration

In the example above it is assumed that Host 1 and Host 2 are already registered users, meaning that usernames and passwords are successfully stored in the XMPP user database. Applications with a static user base are of course conceivable, but especially in the volatile mobile environment dynamic joining of new users might be necessary. Therefore, in addition to the MPM-Core, the Notifier-Peer and an Android library, also a registration component is delivered. More details about all provided software artifacts can be found in Section 6.2.

In comparison to the Notifier-Peer, this component is not a host within the MPM network, because new joining peers are not registered yet and can thus not communicate over XMPP.

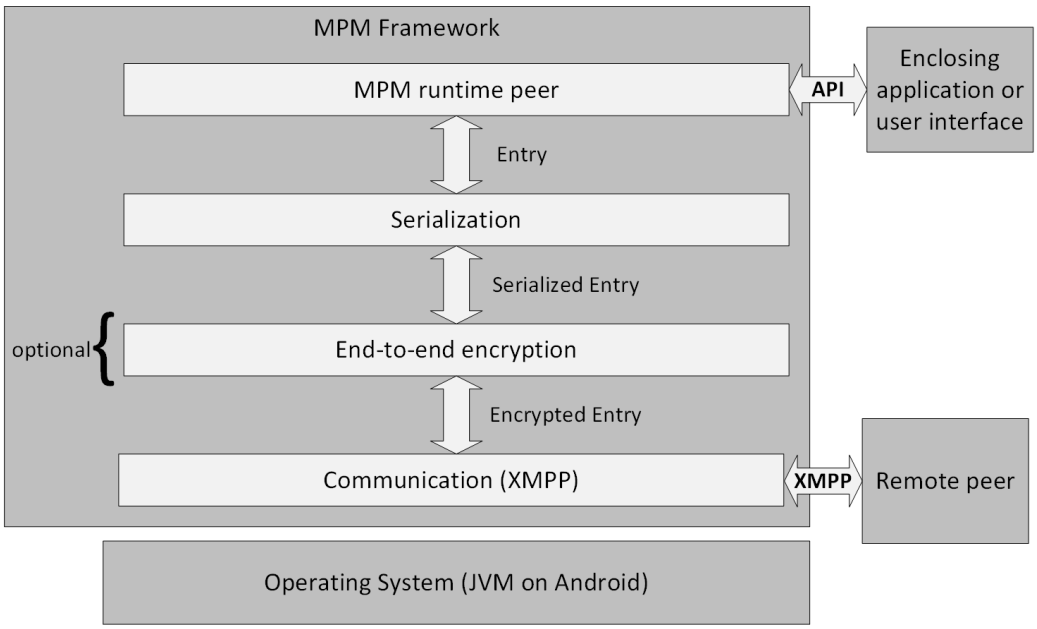

Figure 5.2: The architecture of a host that uses the MPM framework with its four layers [Sch17a].

5.4 Architecture of an MPM host

After the description of the overall architecture of system components involved in P2P applications that are built upon the MPM framework, this section will describe the architecture of a single host (i.e. *Host 1* in Figure 5.1).

5.4.1 Communication and security layer

Figure 5.2 depicts the four layers of a host that is based on the MPM framework. At the bottom, the transport or communication layer is located, which is responsible for exchanging Entries with remote hosts. In the reference implementation this layer is using the Extensible Messaging and Presence Protocol (XMPP) with the TCP/IP transport binding. The data sent over the communication layer itself can be encrypted by activating TLS. Additionally, an optional encryption layer is placed on top of the communication layer to enable end-to-end encryption between two hosts. In comparison, when using only the channel encryption of the communication layer (TLS over XMPP) the server has to be fully trusted, because the server could still eavesdrop packages between two hosts. In [Sch17a] Jörg is describing this layer in greater detail.

5.4.2 Serialization layer

The purpose of the next layer is to *serialize* and *deserialize* String data to Entries and vice versa. As defined in requirement *NFR 5* (see Section 3.1.2), two exchangeable

serialization layer implementations are delivered. The first one serializes the Entry into a human readable format and the second one, aiming at performance, serializes the data into a non-readable binary form (encoded as String). Each implementation itself is again split into two parts: the serialization of the *Entry* and its properties on the first hand and the serialization of the Entry's *data* on the second hand. A more precise description of this layer can be found in the design and implementation part of Jörg Schoba's thesis [Sch17a].

5.4.3 Runtime-Peer layer

The layer on top of the stack is the actual Runtime-Peer. It contains all necessary system Peers (IO-Peers and Exception-Peer) and all user-defined Peers and Wirings. Detailed insights into the structure and the components of the RTP are given in the subsequent Section 5.5 and in Section 6.3 of the implementation part. As shown in the architectural diagram the RTP exposes an interface in order to allow the enclosing application to communicate with it. Such an encapsulating application can be e.g. a simple Java program, a Spring application or an Android application. In particular, any device that is running a compatible Virtual Machine (VM) to interpret the Java 1.7 bytecode of the reference implementation can use the framework. The different execution environments are topic of Section 6.1 of the implementation chapter.

All layers are implementing well defined interfaces so that they can be exchanged by different concrete implementations, as it is required by *NFR 6* in Section 3.1.2.

5.5 Architecture of the Runtime-Peer

In the third part of the architectural overview the Runtime-Peer is described. Figure 5.3 shows the interface of the RTP to the enclosing application, the system Peers, all user-added Peers and the flow of Entries between these components. The following subsections describe fundamental insights and important components of the RTP in greater detail.

5.5.1 Overview

The RTP is the main component of the MPM framework and in each execution environment (VM) only one RTP exists. This single instance acts as container for all Peers of this host (including the three internal system Peers) and controls the execution of the underlying Peers and Wirings.

The blue, green, yellow arrows in Figure 5.3 represent Entries and illustrate the data-flow between different components. The blue ones represent Entries that are received from or sent to a remote host (external), the green illustrate Entries that are created within this RTP and yellow ones illustrate Entries that are transported from one Peer of this RTP to another Peer (internal).

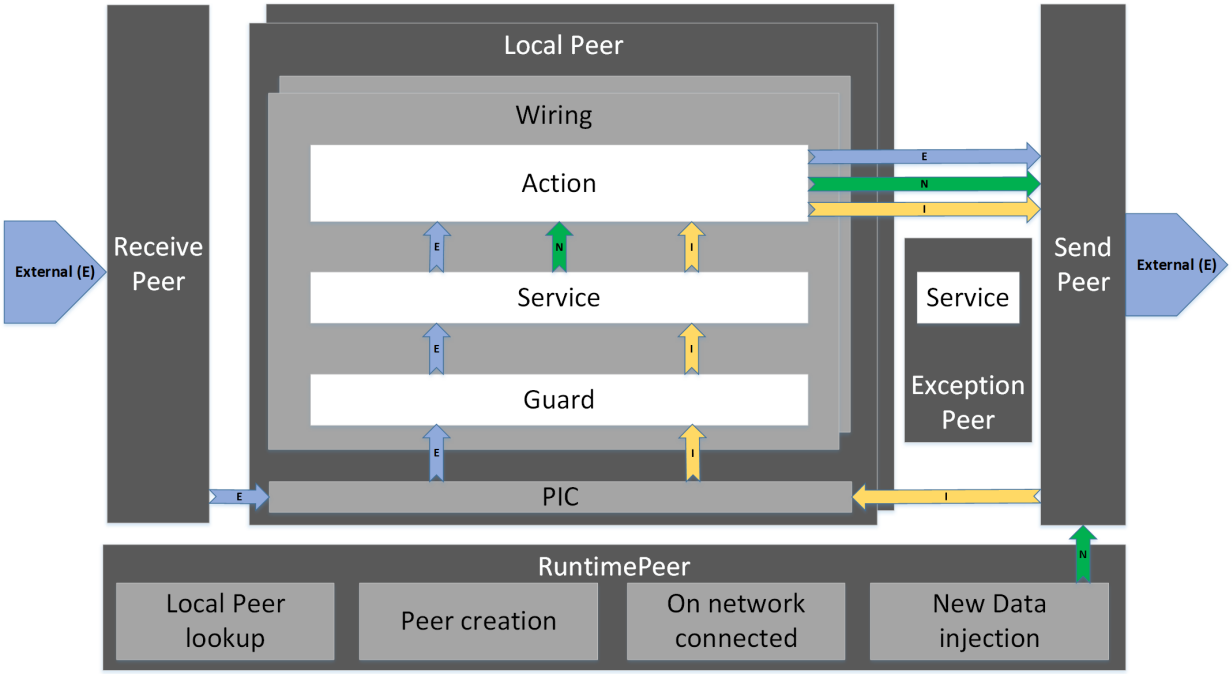

Figure 5.3: Architecture and important components of the MPM *Runtime-Peer* [Sch17a].

5.5.2 Interface of the Runtime-Peer

The interface of the RTP allows the surrounding application to call the *start()* and *stop()* methods. On startup, the RTP initializes and starts all internal Peers and a connection to the XMPP server is established. Each Peer gets its own thread within the process and tries to execute its Wirings. When the *stop()* method is called, the connection to the server is closed and the RTP forces the currently running Peers to stop and consequently to stop the running thread. Apart from the important *start()* and *stop()* methods, the RTP exposes the following essential methods:

- **createPeer(String name):** A Peer can only be created by using this method on the RTP instance, assuring that the Peer gets initialized with the correct values (i.e. Peer-URI). At the same time the new Peer is added to the internal list of local Peers.

- **onNetworkConnected():** This method can be called to inform the RTP that the network connection is available again, since a stable connection can not be guaranteed in a mobile environment. To avoid recurring and unsuccessful reconnection attempts in a predefined interval, the RTP will only try to re-establish the connection if the surrounding application requests the RTP to do so. The two currently predominant mobile operating systems, Android and iOS, provide an easy

subscribe mechanism for network change events. A notification about the network availability can then be forwarded to the RTP.

- **injectData(String entryType, Serializable data, PeerURI dest):** In the mobile profile of the Peer Model there shall also be the possibility to insert data from the surrounding application into the system. In the original model this was only possible via Services and Actions of predefined Wirings. The injected data is wrapped into an Entry and forwarded immediately to the internal *Sender-Peer* (see Section 5.5.5). Depending on the *dest* property, specified during the *injectData()* call, the Sender-Peer either inserts the Entry into the PIC of a local Peer or sends the Entry over the wire to a remote Peer.

5.5.3 Peers and Wirings

In Section 5.2 the concepts of a *Peer* and a *Wiring* were already introduced. Some further details, especially the functionality and the precise workflow of Wirings, including Guard, Service and Action, are described in this subsection. A Peer within a RTP can be seen as a self-contained and dedicated entity that separates the execution of different tasks or responsibilities. In order to be able to fulfil this task, a Peer contains a list of Wirings, each of them representing a simple and isolated execution step. As described in Section 5.2, the mobile version of the Peer Model only allows the execution of one Wiring per Peer at the same time. Therefore, each Peer is executed in an own thread and the control is given to each Wiring in a serial execution order. In the case that in one iteration no Wiring could be executed (because not all desired Entries for that Wiring have been available in the container), the thread gets suspended until a new Entry is written to the PIC of that Peer. A Wiring defines the flow of Entries from its Peer's PIC to other internal containers or to external hosts and may execute Services. It is constructed of the following parts:

Guard

The Guard of a Wiring defines whether the Service and Action of that Wiring shall be executed or not (see Figure 5.4). It consists of one or more *READ* or *TAKE* operations, whereby at least one consuming *TAKE* operation is mandatory to avoid unwanted endless loops. All defined *Link-Operations* are performed on the PIC of the Wiring's Peer. In the event that all operations could be successfully performed, the Guard is seen as satisfied. Afterwards, the fetched Entries are added to a so-called Entry-Collection (EC) and are forwarded to the next part of the Wiring - the Service. In the case that not all Link-Operations can be performed, the Wiring is stopped at this position and the next Wiring gets executed. All Entries that have already been taken from the PIC in an unsuccessful Guard execution are written back to the container.

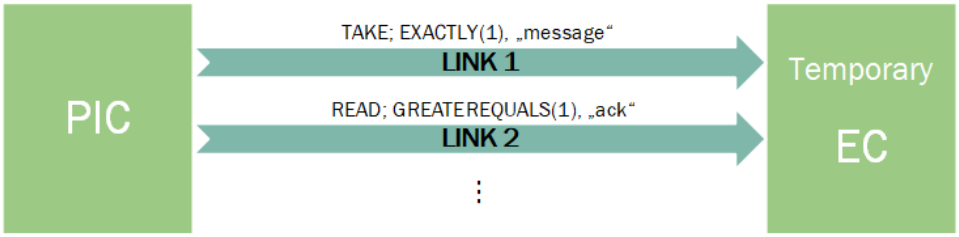

Figure 5.4: A sample Guard with two *Link-Operations*. Link 1 takes exactly one Entry of type *message*. Link 2 reads at least one Entry with type *ack*.

Service (optional)

Each Wiring can have zero or exactly one Service (see Figure 5.5) - this is a further restriction in contrast to the original Peer Model. Nevertheless, the developer can call various Services on its own within the single Service that is registered for a Wiring. The Service is only called if the Guard of that Wiring was successful. In that case, the constructed EC is passed as parameter to the *execute()* method of the Service. The Service may then use the Entries of the EC and can add new Entries or remove existing ones. The EC is actually a special kind of container similar to the PIC of a Peer and provides the same interface. The Service execution is sandboxed by the Wiring in order to avoid that unexpected errors have a negative influence on the system. Nevertheless, if a Service crashes unexpectedly, the Wiring will still try to execute the successive Action.

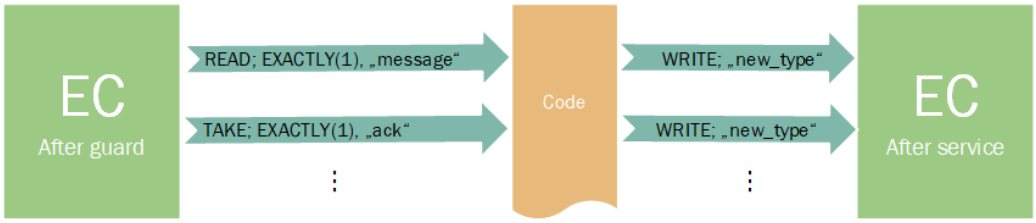

Figure 5.5: A sample Service that reads and takes Entries from the EC and writes two Entries with type *new_type* to the EC. After the execution there are three Entries in the EC - one with type *message* and two with type *new_type*. The Entry with type *ack* is taken during the Service execution and could therefore not be used in a subsequent Action.

Action (optional)

The last part of the Wiring is the optional *Action* (see Figure 5.6). Either the Service or the Action has to be defined, otherwise the Wiring is not valid and cannot be added to a Peer.

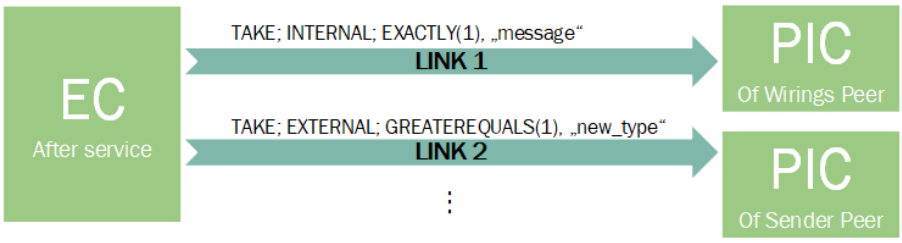

Figure 5.6: A sample Action with one internal and one external *Link-Operation*.

An Action can have 0, 1 or more *Link-Operations*. One Action-Link defines the amount and type of Entries that shall be transferred from the resulting EC of the Service (or the Guard, in the case no Service is defined). In the MPM the Entries involved in an Action are always taken from the EC, whereas in the original Peer Model also a read operation can be performed on the EC. An Action-Link can be *internal* or *external* with respect to the current Peer. An internal Action will take an Entry with the defined Entry type from the EC and inserts it to the PIC of the inherent Peer. An external Action will write Entries with the defined Entry type to the PIC of a Peer within the current (RTP) or to the Peer of a remote host. The actual destination Peer is depending on the *dest* property of the Entry taken from the EC. In contrast to the Guard-Links, not all Action-Links must be successful. For example, if the Action defines three Links and the second Link-Operation is not possible (i.e. because the Service did not add the desired Entry with a specific type), only the first and third Links are being performed.

The sequence diagram in Figure 5.7 illustrates the full execution of a Wiring. The second part is only executed if the Guard was successful. The EC is generated in the Guard and passed to the (optional) Service and finally to the (optional) Action.

5.5.4 Receiver-Peer

There are three automatically added system Peers in each RTP. One of them is the *Receiver-Peer*, which is responsible for receiving Entries from the XMPP server and forwarding it into the appropriate container of a local Peer. This special Peer is constructed as any other user-added Peer. The main difference is that the single Wiring of the Peer is predefined and added during the Runtime-Peer's creation. The Wiring comprises one Guard-Link that *takes* exactly one Entry of type *START_RECEIVER*, one Service that actually listens for incoming Entries, and no Action. On runtime startup one Entry of type *START_RECEIVER* is written into the PIC of the Receiver-Peer, forcing the Service to be executed exactly once. This is because the Guard operation is of type *TAKE* and therefore the Entry is also removed from the PIC after the first successful Wiring execution. The ReceiverService only registers a message processor for incoming Entries by using the interface of the communication layer.

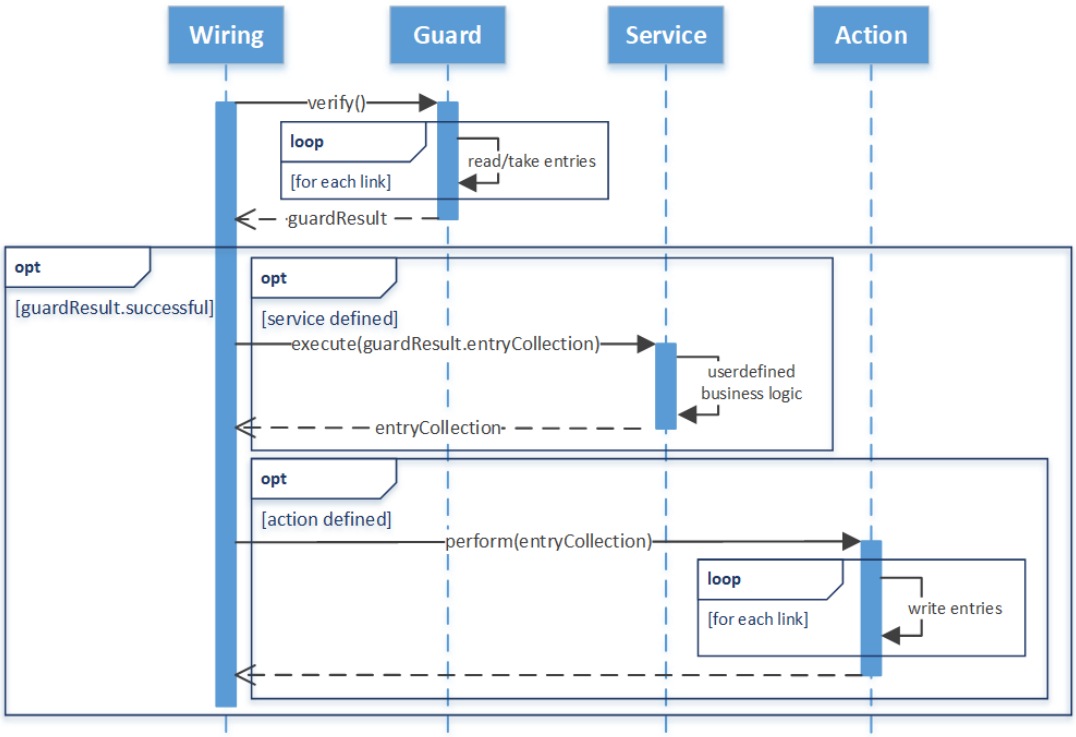

Figure 5.7: The sequence diagram of a Wiring.

5.5.5 Sender-Peer

The second system Peer that is automatically provisioned is the *Sender-Peer*. Its function is to transfer each Entry from its PIC to the Peer defined in the *dest* property of the specific Entry. Because the task of the two system Peers (Receiver and Sender) involves input and output of Entries, they are also termed as *Input/Output (IO)-Peers*. To achieve its objective, the Sender-Peer has one predefined Wiring, comprised of one consuming Guard-Link (*TAKE*) and one Service. The special Guard-Link of the Sender-Peer does not define any Entry type. This is because every Entry shall be sent - independently of its Entry type and data. The Link-Count is defined as *GREATEREQUALS(1)* so that in one Wiring execution all pending Entries are sent. The Service then iterates over all Entries of the delivered EC and either writes them to the PIC of a local Peer or sends them to a remote host (by calling the send() method of the communication layer). If the device is currently not connected to the internet or any other communication exception occurs, the Entries are added to a list of pending entries (see Section 6.3.5).

Send exceptions

As it is described in more detail in Section 5.6.7, the whole RTP (including the state of all containers) can be optionally persisted. The send process and the deletion of

the Entry from the persistence should be performed in one single transaction to avoid dropping an Entry or sending the Entry twice. This might happen if the application is killed unexpectedly exactly while one of these statements are executed. To fully avoid this behaviour, distributed transactions would be required. Nevertheless, depending on the current application, one case is usually more problematic than the other. It has been decided to let the software developer decide if the Entry shall be resent or not, by enclosing the *send()* method of the communication layer with a local transaction. In the very unlikely case that the transaction can not be committed (again this can only happen if the application is abruptly terminated exactly during the send process) the system will recognize the failed transaction on the next runtime startup. In this case, the Entry that was probably not sent is being wrapped into a special *Exception-Entry* and forwarded to the third predefined system Peer - the *Exception-Peer*. Depending on the application and on the importance of the concerned Entry, the software developer can decide whether the probable loss of the Entry or the duplicated transfer of the Entry cause less problems in the system. A more detailed description about this transaction process is provided in Section 5.6.7 in this chapter and in the implementation part (Section 6.6.5).

5.5.6 Exception-Peer

The third automatically added system Peer is the *Exception-Peer*. As the name suggests, it is responsible to manage exceptions that might occur while the Runtime-Peer is running. The idea behind the Exception-Peer is to forward all exceptions to a central component, where the developer can decide which specific action should be taken in consequence. In the version of the mobile profile of the Peer Model the concept of a so-called *Exception-Entry* is introduced. It extends a simple *Entry* by the *exception-type* property and restricts the *coordination-type* to the value *exception*. The actual Entry that was involved in an exception is stored in the object value of the Exception-Entry. Such an exception can be the expiration of the TTL property (*TTL_EXCEPTION*) or the previously described unlikely situation during the send process (*POTENTIAL_SEND_EXCEPTION*).

In a nutshell, every Peer (except the Exception-Peer) has one predefined Wiring that takes Entries with type *exception* from the PIC of the Peer and forwards it to the PIC of the Exception-Peer. There, the countermeasure for a specific error can be performed by the application developer. In order to achieve that the developer can define a particular Service (and Action) for the Exception-Peer. More details about the interface of the Exception-Peer and how such a Service and Action can be added to the Peer are described in the implementation chapter (see Section 6.3.6).

- **TTL_EXCEPTION:** The *time-to-live* TTL property specifies the point in time when an Entry loses its validity (see Section 5.2.1). During the attempt to read or take an Entry, the container checks if the available Entry is valid (no expired TTL property). To reduce resource consumption, this check is only done when container operations are performed, as otherwise a maintenance task would have to permanently check Entries for validity. In the case of a TTL expiration, the Entry

is immediately wrapped into an Exception-Entry and is not returned by the *read()* or *take()* method. In fact, this means that the Entry stays in the container, but gets transformed to an Exception-Entry. Instead, the next Entry in the container is checked and is again only returned by a *read()* or *take()* method if the TTL is valid. Existing Exception-Entries are then transported via a predefined Wiring to the Exception-Peer. This Wiring is added automatically to every Peer. It consists of a Guard that takes all available Entries with type *exception* from the PIC of that Peer and an Action that writes those Entries into the PIC of the *Exception-Peer*. Entries that expire during the execution of a Wiring are forwarded to the Exception-Peer with a predefined Action-Link that is automatically added to each Action of a Wiring. This Link takes all Entries of type *exception* from the EC and writes them to the PIC of the Exception-Peer. The predefined Wiring of the Exception-Peer will then be executed subsequently. This works also if the Entry expires just after the Service execution, at the time when the Action tries to take the Entries from the EC. This is because this predefined Link is executed as last Action-Link - so all Entries that might have been expired before are already wrapped into Exception-Entries and are then taken by the final exception Link from the EC.

The last possible situation, where an Entry can expire, is while the Action tries to write the previously valid Entry into the defined container. In that case, the container will check the validity of the Entry before it is added and wraps it, if necessary, into an Exception-Entry and the whole Entry itself as data property. Again, the predefined Wiring that is added to each Peer automatically will take this Entry and forward it to the Exception-Peer.

- **POTENTIAL_SEND_EXCEPTION:** As already described in Section 5.5.5, the send process is executed in a local transaction. In the case that the application crashes during the send process, the message could have either been sent or not. A potentially not sent Entry is not automatically resent, to avoid duplicated message delivery. In fact, the designated Entry is wrapped into an Exception-Entry and written into the PIC of the Exception-Peer. A predefined Guard takes the Entry from the PIC and executes a Service and a consecutive Action that, together, define the countermeasure. Because there is no adequate solution that fits for all applications, the actual Service and Action can be defined by the developer on its own.

5.5.7 Lifecycle of the Runtime-Peer

In Section 5.5.2 the most important methods of the Runtime-Peer were already described, including the essential *start()* and *stop()* methods, which control the execution of Peers and their associated Wirings. Figure 5.8 shows a coarse lifecycle overview of the RTP.

As initialization steps, the hostname and the password have to be defined, so that a successful connection to the XMPP server can be established and all local Peers can be added via the method *createPeer(String name)*. If there is any problem with the

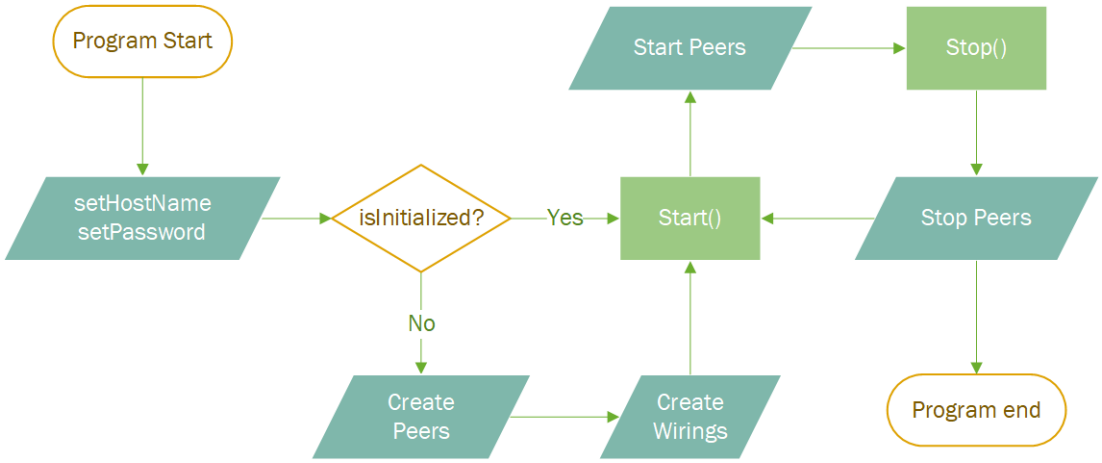

Figure 5.8: The lifecycle of the Runtime-Peer.

connection establishment on RTP startup, an exception is thrown. The developer can then perform an individual action depending on that exception (i.e. a pop-up for the user that the credentials were wrong or that there is currently no internet connection available). If the authentication was successful and the connection could be established, firstly the system Peers (Receiver, Sender and Exception-Peer) and secondly all locally added Peers are started. The *start()* method will also initialize all added Peers with Entries that were persisted during the last RTP execution (if persistence is enabled - see Section 5.6.7). After the RTP was started for the first time within an execution context, it is seen as initialized. From that time on no further Peers and Wirings can be added. As described in Section 5.5.3, each Peer has an own thread and will now try to execute its Wirings in a cyclic fashion.

If the RTP is being stopped the stop request is forwarded to all Peers. In that case, the Peer will not execute any further Wirings and will try to stop the running thread as soon as possible. However, a currently running Wiring will run until it has done its job. This depends mostly on the associated Service implementation. If the developer is doing any long-running calculations, a Service (and therefore a Peer) may still run in the background, although the RTP is stopped already. Any Entry transmission to a remote host would not succeed in such a case, because the connection was already closed. Although, on a successive start() call the Entry would be sent, because undelivered Entries are cached in the Sender-Peer. If the program is completely destroyed, those pending Entries are resent only if the persistence is enabled (see Section 5.6.7).

5.6 Mobile design considerations

Many P2P communication networks and frameworks are designed for desktop and server applications. This can be also seen as almost no work mentioned in Chapter 2 is directly usable in a mobile environment. As discussed before, a central XMPP server is used,

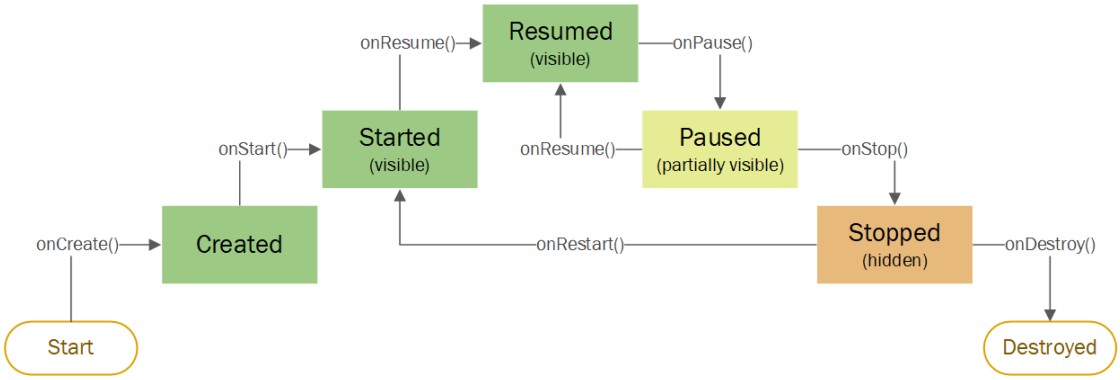

Figure 5.9: The compressed lifecycle of an Android Activity. The rectangles represent the possible states of the Activity.

which acts as a relay to assure reliable communication in any network configuration. Also most coordination models and frameworks are designed for non-mobile devices. This is why in Section 5.2 a mobile profile of the used coordination model has been elaborated that takes into account some limitations in mobile environments, like limited processing power or battery usage. Further mobile design decisions are discussed in this section. Since the reference implementation is provided for the Android platform, only Android specific background knowledge and considerations are presented here.

5.6.1 Application and Activity lifecycle

An Android application is built upon different fundamental components, while the so-called *Activity*[2] is playing a central role. An Activity represents a single page in an application and is usually responsible for user interaction. Unlike in different programming languages, where a program is started via a *main()* function, an Android app starts a declared Main-Activity of the app. As described in the Android developer guide[3], the developer has to extend from the predefined Activity class and can override different callback methods. As depicted in Figure 5.9, the first callback method (also known as lifecycle hook) is the *onCreate()* method where a developer can place code that should be executed before the Activity is created. Another important callback is the *onPause()* method, which is called when the Activity has lost the focus.

In the Android developer guide[4] the lifecycle of a whole Android application and its process is described. It is important to know that the lifetime of an Android application can not be directly controlled by a developer. In fact, the system's hardware configuration,

[2] `https://developer.android.com/reference/android/app/Activity.html` accessed: 2017-03-16

[3] `https://developer.android.com/guide/components/activities/index.html` accessed: 2017-03-16

[4] `https://developer.android.com/guide/topics/processes/process-lifecycle.html` accessed: 2017-03-17

the currently installed operating system and the concrete user behaviour are affecting the overall lifetime. Furthermore, the relevance of an Android application or component in the system differ depending on the kind and the current state of the process. Therefore, the Android system categorizes each process into an *importance hierarchy*:

- **Foreground process:** This is most likely the Activity the user is currently interacting with, but might also be a foreground *Service* or a component that is executed on the special User Interface (UI)-thread at the moment (i.e. a callback function of a BroadcastReceiver or Service). The number of such processes in the system is very low.
- **Visible process:** The user is currently aware of this process, although it is not the Activity at the top of the screen. This might be an Activity in the state *Paused* or a special *Service* that was started via the method *startForeground()*. There are usually also only a few such processes in the system.
- **Service process:** A *Service* is not directly visible to the user, but is doing important work in the background that the user cares about (see Section 5.6.3).
- **Cached process:** This last kind of process defines an application that was previously running, but is currently in the *Stopped* state and is not important to the user anymore. Those processes are kept in an Least-Recently-Used (LRU)-list and on application *restart* the app will start much faster than at the first execution.

The Android system will terminate applications from the list of *cached* processes first, if memory is needed elsewhere. Usually there are a lot of *cached* processes, especially on newer devices with at least 2 GB of Random-Access-Memory (RAM). So the operating system will hardly reach a state where it has to kill *service processes* or even *visible* or *foreground* processes.

5.6.2 Android Context

According to the Android reference documentation[5], the Android *Context* is, as the name suggests, *'an interface to global information about an application environment.'* It provides access to shared preferences and local files, system-level services (like sensors, network manager, ...) and it has to be used to execute application-level operations (like starting new activities or sending broadcast messages). Many system-provided methods need a *Context* reference as parameter to use those functionalities.

Therefore, the Context is also fundamental for the MPM framework, because an application developer may very likely want to use features mentioned above in its MPM-Service implementations. Fortunately, each Activity and also each Android Service extends from the system-provided class *Context*.

[5] `https://developer.android.com/reference/android/content/Context.html` accessed: 2017-03-17

5.6.3 Android Service

Let's recap the important functional requirement *Running in the background (FR 2)* from Section 3.1.1. By solely using the *MPM-Core* (see Section 6.2) that is specifically built for the Android platform (see left path of Figure 6.1), this requirement is already fulfilled, because the *Runtime-Peer* would already run in the background. More precisely, it will spawn a dedicated thread for each Peer assigned to it. However, if it is assumed that the Activity that created and started the Runtime-Peer is being closed, the Activity would switch to the state *Stopped* (see Figure 5.9) and the application's process would be added to the LRU-list of cached processes. The started RTP would still run in the background to do its work and could still receive Entries from a remote host. However, the Android system may kill the running threads and release the memory space used by the application. The point in time of this cleanup operation is undefined. It is depending on the concrete device and its hardware specification, the version of the operating system and moreover on the usage behaviour of end users (i.e. which and how many apps and services are running).

Therefore, to ensure a continuous and reliable execution of the RTP, another solution has to be found. Fortunately, Android offers so-called *Services*[6] that meet exactly the requirements of the MPM framework:

> *"A Service is an application component that can perform long-running operations in the background, and it does not provide a user interface."*

A Service can be started from any application component that has access to a valid Context object (see Section 5.6.2) and even if the user switches to another app, the Service remains running in the background. Components can also communicate with the Service in both directions. More details about the different possibilities to start a Service and its lifecycle are discussed in the implementation part (see Section 6.6.2).

As already described in Section 5.6.1, a currently running *Service* process is stopped by the operating system only if there is a memory shortage and a more important Service or a visible or foreground process needs the resources. According to the Android developer guide[7], a long-running Service (30 minutes and more) lowers its position in the importance hierarchy over time. Therefore, the longer the Service runs, the more likely it will be killed by the system. The reason for that is to avoid Services with memory leaks or other misbehaviour that possibly would consume too much RAM so that other applications could not use the benefits of cached processes. However, if specified, the Service will be restarted by the operating system as soon as the resources are available again. Because the application developer that uses the MPM framework will decide how long the Service shall run in the background and it is not known how much resources will be available on

[6] `https://developer.android.com/guide/components/services.html` accessed: 2017-03-19

[7] `https://developer.android.com/guide/topics/processes/process-lifecycle.html` accessed: 2017-03-17

each device later on, the Android Service has to be designed in a way that unexpected force-stops and subsequent restarts of the Service are gracefully handled. To guarantee reliability (i.e. no lost Entry in any container) the current state has to be persisted permanently. Therefore, also a *persistence layer* has been implemented that is described in more detail in Section 5.6.7.

5.6.4 Runtime-Service

With the introduction and analysis of fundamental concepts and important components of the Android platform it has been shown that the Runtime-Peer will only run long-term in a reliable way if it is started it in the context of an Android Service. As an Android Service is a subtype of the Android Context class, important application-specific resources are accessible and application-level operations can be executed. Therefore, a predefined Android Service is provided that can be used out-of-the-box by any MPM Android application developer. Necessary lifecycle hooks of the Android Service are already implemented, which will start and stop the RTP. Additionally, essential system event listener are pre-registered, for example a listener to receive network state changes of the device. The provided Runtime-Service (RTS) class has some *abstract* methods that have to be overriden by developers in a concrete subclass. In particular, those methods deal with coordination-related configurations (i.e. creating and adding Peers or inserting initial Entries). Figure 5.10 illustrates the Android RTS and its relationship to other components. The filled rectangle on the left side represents the abstract Runtime-Service. Other components of an Android application, like an Activity, can start the Runtime-Service and therefore also the RTP. Furthermore, components can bind to the Service in order to call methods, for example to inject new data into the RTP.

5.6.5 FCM Services

The application developer can decide if the Android Service (and therefore also the RTP) shall run permanently in the background or only while a specific Activity is displayed. For long running configuration tasks it might be a good decision to run the Service permanently in the background. However, if there is no workload expected when the Activity is closed, the Service should be stopped to save memory and processing resources. In this case, though, also the connection to the XMPP server is disconnected, meaning that incoming messages can not be received any longer.

In order to make use of both features, namely stopping the Service to save resources, but still be able to receive incoming Entries at the same time, notifications are used. The overall big picture and the architecture of the notification approach with FCM[8] was already described in Section 5.3.2 (Notifier-Peer). There are two Android Services that have to be implemented by an application developer to use FCM. The first is used to notify the application about the creation of a new unique FCM token and the second

[8] `https://firebase.google.com/` accessed: 2017-05-20

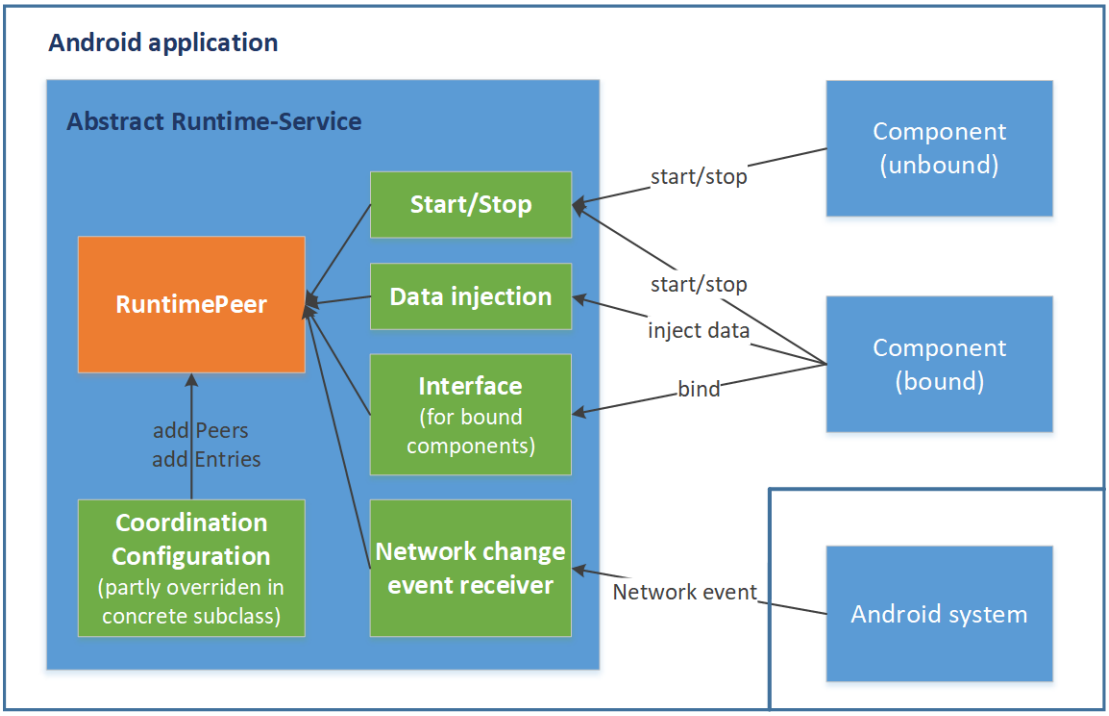

Figure 5.10: Overview of the provided Android Runtime-Service and its relationships to other components.

one implements the actual code that shall be executed on incoming notification messages. Both Services are described in detail in the implementation chapter (see Section 6.6.3).

Together with the *Runtime-Service*, described in the previous section, both mandatory Android Services are delivered fully implemented to application developers with the *MPM-Android* library artifact (see Section 6.2). These two Services can be used by an application developer out-of-the-box. Not a single line of code needs to be implemented - only a few simple setup steps have to be completed (see Section 6.6.3). If properly performed, the MPM RTP will be successfully notified on any incoming message, presuming that a correctly configured Notifier-Peer instance is deployed.

FCM is also available for the iOS platform and could be also used in simple web apps running in a browser. Thus, this notification approach is not disqualified by requirement *Operability on popular mobile platforms NFR 8* (Section 3.1.2). As concluding remark, FCM is only usable on devices with Android 4.0 or higher. However, at the time of writing only 1.0% of all Android devices were using a lower Android version[9].

[9] `https://developer.android.com/about/dashboards/index.html` accessed: 2017-03-21

5.6.6 User interaction

Now that we have a clearer understanding of the Android system and which techniques and components are used for the framework, the communication between the Graphical User Interface (GUI) and the MPM framework is discussed.

In the previous section, the Runtime-Service has been introduced, which controls the execution of the RTP and therefore also holds a local variable of its single instance. New Entries can be inserted into the RTP with the method *injectData(String entryType, Serializable data, Peer-URI dest, long tts, int ttl)*. This method, which is also available on the Runtime-Service class, creates a new Entry with the specified properties and inserts the Entry into the PIC of the local or remote Peer defined in the *dest* property. This means that this method allows the delivery of an Entry to a remote host without defining a Wiring. Usually, a dedicated Action-Link (with a specific type and count) has to be defined to achieve this goal. The *origin* property of the injected Entry is, in that case, the name of a pseudo system Peer (denoted as *UI_PEER*).

Because the method *injectData()* also exists on the Runtime-Peer, it could be called directly on that instance object too. However, the developer is responsible to ensure that the runtime is actually initialized and running at that moment. By using the Runtime-Service, this precondition is inherently satisfied because the Runtime-Service is only running if the underlying Runtime-Peer is started.

Communication from within the MPM system to the surrounding Android application is more complicated. The only place where this may happen is during the execution of an MPM Service. However, all Peers, Wirings and Services have to be added to the Runtime-Peer at a point in time where the GUI components (like activities or fragments) have most likely not been created yet. Furthermore, the Runtime-Peer could also be started in the background. Therefore, in general, no concrete object reference of a GUI component can be added to the MPM Service.

Fortunately, Android offers so-called *broadcasts*[10] to send and receive messages between different Android components. Those messages can be sent locally or even across application boundaries to inform the receivers about an event of interest. To receive a broadcast message, apps or components have to register for the desired event with a String identifier, similar to the well known *publish-subscribe* pattern. Obviously, when sending a broadcast, the same String identifier has to be declared to ensure only the desired subscribers receive the message. In the implementation part more details regarding this broadcast mechanism is explained and some code snippets are shown (see Section 6.6.4).

Also the Android system sends broadcasts when important events occur, for example when the device has been successfully started (*android.intent.action.BOOT_COMPLETED*) or the internet connection state has changed (*android.net.conn.CONNECTIVITY_CHANGE*). A broadcast receiver for the latter event is already implemented in the abstract Runtime-

[10] `https://developer.android.com/guide/components/broadcasts.html` accessed: 2017-03-25

Service. However, an application developer might override the lifecycle methods of the Service to register further events of interest.

5.6.7 Persistence

The Entries in all containers can be seen as the current state of the Runtime-Peer. However, this data is only available in the volatile memory (RAM) of the device, meaning that all Entries would be lost when the application is stopped completely or if there is a complete shutdown of the device.

The need for a persistence layer

When conceptualizing the fundamentals and the architecture of the MPM framework in the first place, a persistence layer was not part of the core design. It was seen as a convenient feature that could be implemented in a future work. During a more detailed elaboration of the Android platform (see Sections 5.6.1 to 5.6.3), however, it has been found out that the operating system may kill the Runtime-Service (that contains the RTP) without any prior notification, resulting in an unreliable product. This unwanted behaviour particularly applies to devices with low hardware capabilities and if long-term tasks are running in the background. In respect thereof, the only way to preserve data consistency and reliability is to steadily persist the current state of all containers. For this purpose, each *write* or *take* operation has to be reflected with a persist or remove operation on a persistent storage.

Persistence during a Wiring execution

Inserted or taken Entries change the current state of containers in different parts of the framework. First of all, an Entry received from a remote host is immediately inserted into its destination's PIC container by the *ReceiverService* (see Section 6.3.4). Here, the Entry has to be added to the persistence as well, in fact just before the Entry is actually written into the PIC. The reason for this is that any container modification immediately triggers the thread of the affecting Peer. Consequently, a recently inserted Entry might be involved in the execution of a Wiring, leading to inconsistencies in the persistence (if the insertion of the Entry has not yet been completed in the persistence). The same situation applies for Entries that are added to the system via the Runtime-Peer's *injectData()* method. Inside the method a new Entry is created and inserted into the PIC of the Sender-Peer. Again, the Entry must be persisted immediately before writing it to the container to avoid a race condition. For both cases the persistence layer has to provide a method with the following interface: *persist(IEntry entry, IContainer container).*

As already described in Section 5.2.2, no transaction handling is necessary on the container level, because only one Wiring per Peer can be executed at the same time. However, when introducing a persistence layer, Entries taken by a Guard and Entries written by the same Wiring's Action have to be persisted/removed in one single transaction to avoid inconsistencies in the persistence (see method *removeAndPersist()* in Figure 5.11).

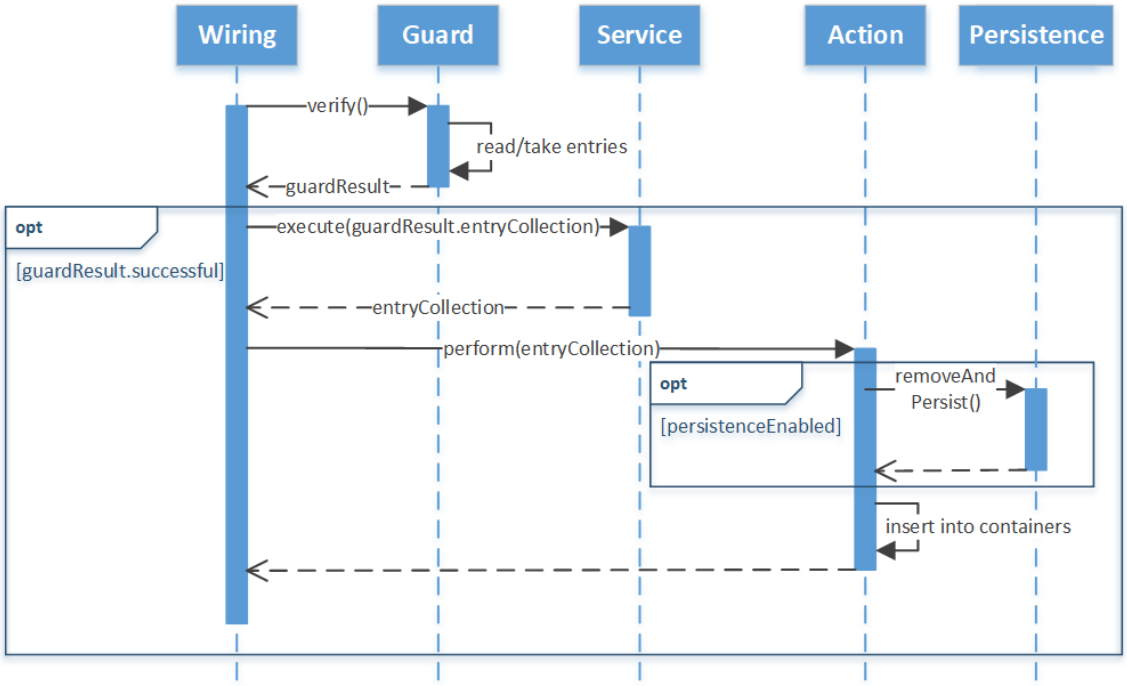

Figure 5.11: The sequence diagram of a Wiring execution with the call to the *PersistenceManager*.

Entries that are taken from a container could get lost if the application is unexpectedly terminated during a Wiring's execution if they are removed from the persistence without any transaction handling.

Figure 5.11 illustrates the call to the persistence layer during a Wiring's execution. The RTP can be terminated at any point in time - data integrity is always guaranteed. Prerequisite for this is the atomicity of the method *removeAndPersist()* (see Section 6.6.5 how this is guaranteed in the Android persistence implementation). If an application crashes before this method is called, the Entries taken by the Guard are still associated with the PIC of the Wiring's Peer in the persistence and would be reinitialized into that PIC on application restart. After calling the method *removeAndPersist()* all affected Entries that were taken by the Guard and that will be written by the Action are already removed from or inserted in the persistent storage. If the application fails after this point in time and the Runtime-Peer is started again, the affected containers would be initialized with the state of the containers after the Wiring's execution.

Persistence during the send process

In all previously mentioned situations data consistency can be preserved. However, when Entries leave the system, distributed transactions would be necessary to ensure this in every exceptional situation. As this kind of transaction handling is quite complex

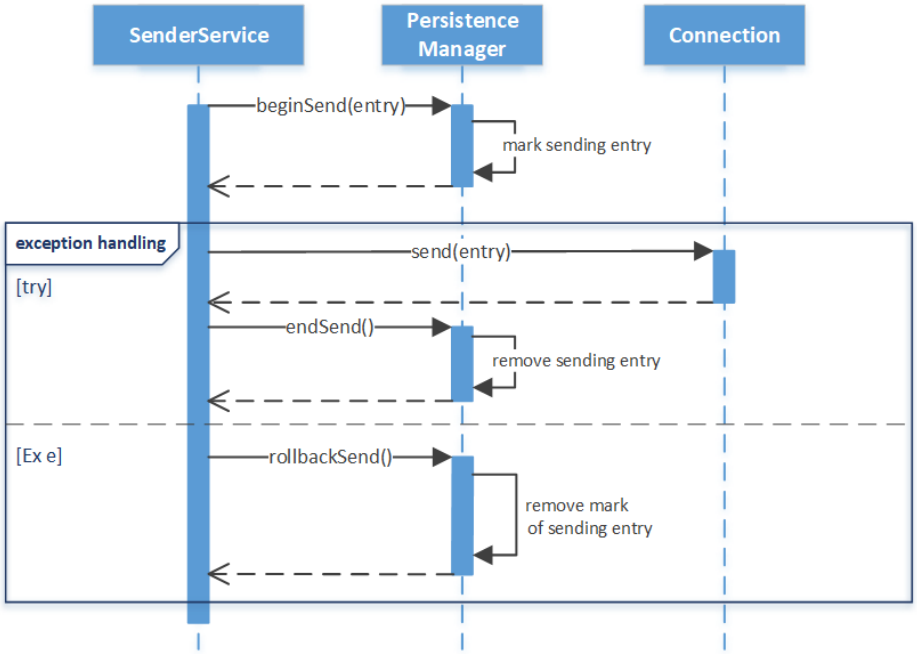

Figure 5.12: The sequence diagram of the transaction handling during the send process.

and also increases resource consumption, a compromise was made. Data integrity need not be guaranteed in every case, but failures during the send process shall be detected. Therefore, a transaction handling with the three methods *beginSend()*, *endSend()* and *rollbackSend()* has been introduced.

Figure 5.12 shows the sequence diagram of the send procedure in the SenderService. The Entry to be sent is marked in the persistence layer by calling the method *beginSend()*. If the app is unexpectedly terminated at this point in time, the Entry could probably not have been sent and the Entry is still marked in the persistence, meaning that the Entry is not valid anymore in the persistence context. On the next Runtime-Peer start this marked Entry gets wrapped into an Exception-Entry and forwarded to the Exception-Peer. The application developer can then decide further steps by defining a Wiring in the Exception-Peer. This Wiring can have an arbitrary Service, which, for example, just ignores or resends the Entry if the exception-type of the Entry is *POTENTIAL_SEND_EXCEPTION*. The actual countermeasure is depending on the current application and cannot be decided for all use cases. In case that the *send()* procedure already returned, implying that the Entry was successfully sent, but the *endSend()* method was not called (e.g. because the app got stopped by the system just between those two statements), also a send failure is detected. Again, the developer can decide which action should be taken on the next RTP startup.

If there is a problem with the internet connection, the *send()* method of the communication layer will throw a *CommunicationException*. In such a case the *rollbackSend()* method

of the persistence layer is called, which simply removes the mark of the stored Entry, making it valid again. In the Sender-Service this Entry is then stored in a list of *pending Entries*. All unsuccessfully sent Entries will be sent again if the internet connection could be re-established and have to go through the same call sequence as an Entry that is sent for the first time (beginSend() -> rollbackSend()/endSend()). Eventually, after a successful send process, the method *endSend()* removes the delivered Entry from the persistent storage.

Enable/Disable the persistence layer

In the first version of the MPM framework reference implementation only a persistence implementation for the Android platform is provided (see Section 6.6.5), due to the potential unexpected termination of the Runtime-Service.

As persisting data to the local persistent storage also means considerable extra cost in terms of processing time, the persistence layer is optional and can be disabled.

5.7 Modeler and code generation

As required by *NFR 7* (see Section 3.1.2), the API of the MPM framework shall be designed in such a way that coordination-related code can easily be generated by a modeler. However, the code shall be still easily legible and modifiable by a developer.

```
IPeer peer = runtimePeer.createPeer("peer1");

IGuard guard = new Guard();
guard.addLink(LinkOperation.TAKE, EntryCount.exactly(1), "type1");
IAction action = new Action();
action.addExternalLink(EntryCount.exactly(1), "type1");
IService service = new TestService();

IWiring wiring = new Wiring(guard, service, action);
peer.addWiring(wiring);
```

Listing 5.1: Sample code to create a Peer and a Wiring.

In Listing 5.1 the code to create one Peer with one simple Wiring (containing of a Guard, Service and Action) is illustrated. Although the code might be generated by a modeler, it is still readable and maintainable by an application developer. By the design of the Peer Model, the business logic (here *TestService*) shall be separated from the coordination logic and has to be implemented by the application developer.

CHAPTER 6

Implementation

In the previous chapter the profile of the MPM was introduced and important design considerations of the framework were presented. Furthermore, the overall system architecture and important system components have been described.

This chapter deals with the technical implementation of the framework and gives deeper insights into some particular parts of the system. The communication layer including serialization and deserialization of Entries, as well as more information on security aspects of the system, like channel encryption and end-to-end encryption, are described in Jörg Schoba's thesis [Sch17a].

Not a lot of code snippets are presented, as the source-code will be publicly available and well-documented. Focus is laid on essential interfaces and classes of core components and on the implementation details of some more complex parts of the framework.

Before going into detail about the implementation, all possible execution environments for applications built upon the MPM framework are presented (Section 6.1), followed by an illustration of the different software artifacts that are provided by the reference implementation (Section 6.2).

6.1 Execution environments

As required by NFR 8 (Operability on popular mobile platforms) the framework shall be designed in a way that it is implementable on different platforms and it should be possible to communicate across platform boundaries. Therefore, the language-independent data format JavaScript Object Notation (JSON)[1] and as a more performant alternative Google's Protocol Buffer[2], which is also available for various programming languages, is

[1] `http://www.json.org/` accessed: 2017-03-02

[2] `https://developers.google.com/protocol-buffers/` accessed: 2017-03-02

used in the reference implementation. More details about those technologies and how they have been utilized in the framework are described in Jörg's thesis ([Sch17a]).

The reference implementation is implemented in the Java programming language. The main reason for that is that the currently biggest player on the mobile market, *Android*, provides a comprehensive Software Development Kit (SDK) that is written in Java. Applications for the Android operating system can be developed with Java in language level 6, 7 or 8. All features of Java version 7 are supported. Starting with version 7.0 (API Level 24) of the Android platform, also a subset of features of Java version 8 is supported[3]. However, the Android platform versions 7.0 or higher are only used by 1.2% of all Android users at the time of writing[4]. Therefore, it has been decided to write the framework with Java 7.

The framework is usable on any device that is able to interpret the byte code produced by Java in version 7. For this purpose, any kind of VM or some other bytecode translator is needed. In practice, this will be the Java Virtual Machine (JVM) for simple Java programs and the Dalvik Virtual Machine (DVM) or the Android RunTime (ART) for Android applications. The DVM is used on the Android platform from version 1.0 (Base) to 4.4 (KitKat) and was designed particularly for embedded and low power systems.

An Android program has a more complicated compilation process than a normal Java application. First, the Java files are compiled to Java bytecode, which would be interpretable by a usual JVM. Next, this bytecode gets translated to *Dalvik* bytecode, stored in Dalvik EXecutable (DEX) files, which is then interpretable by Android's virtual machine (DVM). The successor of the DVM is the Android Runtime (ART), used by version 5.0 (*Lollipop*) and upwards. It can use the same DEX bytecode, meaning that apps compiled for older versions should work also when running with ART on newer devices. ART makes use of Ahead-Of-Time (AOT) compilation, which improves app performance by translating the bytecode of the produced DEX-files to native machine instructions. Google states[5] further advantages over the older Dalvik-VM, including improved garbage collection, debugging enhancements and reduction of power consumption.

6.2 Software artifacts

In the previous section different environments were described in which the reference implementation of the MPM framework may be implemented. Since some libraries used by the framework only work on Android, in particular the communication libraries, two different artifacts have to be provided - the first one for Android and the second one for environments that use a normal JVM. Furthermore, another wrapper around the MPM framework is provided that simplifies the development for the Android platform. The reason for this is the special lifecycle of Android applications, which has to be considered.

[3] `https://developer.android.com/guide/platform/j8-jack.html#supported-features` accessed: 2017-03-02

[4] `https://developer.android.com/about/dashboards/index.html` accessed: 2017-03-02

[5] `https://source.android.com/devices/tech/dalvik/` accessed: 2017-03-02

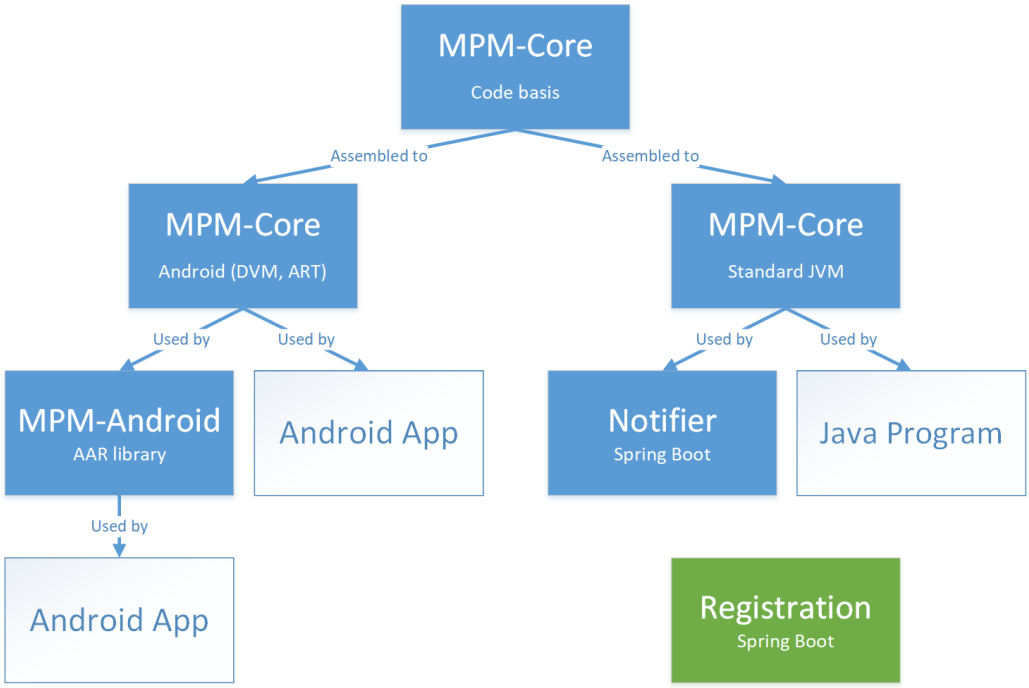

Figure 6.1: Overview of the delivered software artifacts.

More details about this artifact, including an abstract Android Service that can be integrated into an Android project, are described in Sections 5.6 and 6.6.

Figure 6.1 gives an overview of delivered software artifacts. At the top there is the *MPM-Core* project, which includes the source-code of the framework. Depending on the target platform (Android or standard Java program), this code base is assembled with different library-dependencies. The assembled library for standard Java programs can be directly used by any Java program. For example, the *Notifier* project of the reference implementation, which is also provided, uses the standard *MPM-Core* within its *Spring Boot* application. *Spring Boot* facilitates working with databases and encourages a rapid application development and deployment (more details can be found in Section 6.4).

On the left side in Figure 6.1 the Android MPM-Core artifact is shown, assembled specifically for the Android platform. This library could be used out of the box for any Android application. The provided Runtime-Peer could be used as in any other Java program, but the developer may face some difficulties. Especially if user interaction is needed (in the context of an MPM Service), if the runtime should run permanently in the background or if the RTP should be notified about important events of the Android system, a developer would have to add a lot of additional code on his own. To facilitate this functionality and to make sure that the developer does not have to care about those features, an Android library (Android Archive (AAR)) is provided that can be imported by any Android project.

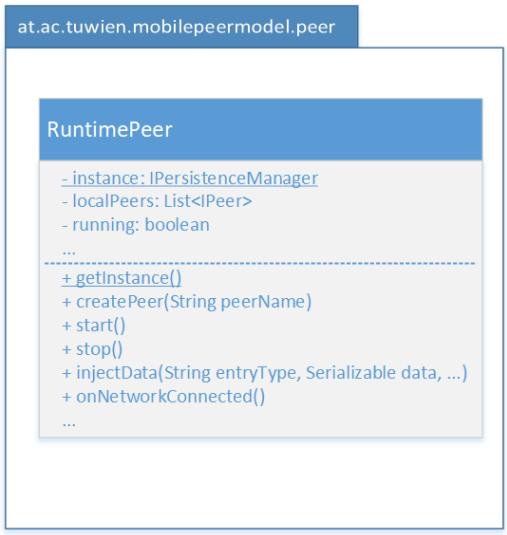

Figure 6.2: The class diagram of the Runtime-Peer.

The last software artifact included in the reference implementation is the registration project. Similar to the Notifier project it is using the popular Java Framework *Spring Boot* that also facilitates web application development. The main purpose of the registration application is to add new users to the XMPP identity database (see Section 5.3.3).

As demanded by requirement NFR 1, the source code of the MPM reference implementation has to be publicly available. Additionally, the fully assembled libraries will be uploaded to a public repository like *maven central*[6] or *jcenter*[7]. In order to use the MPM framework for an Android app, a developer will therefore only need to define one line of code in the build file, e.g. if *gradle* is used, the line could look like:
compile 'at.ac.tuwien:mpm-androidlibrary:1.7.0'.

6.3 Runtime-Peer

To recapitulate from Section 5.5, the Runtime-Peer (RTP) is the main component of the system and controls the execution of all Peers and Wirings. For each execution environment (VM) only one single instance of the RTP can exist. The RTP class is therefore implemented as a *Singleton*, to ensure that only one such object can be created. The class diagram in Figure 6.2 shows important members and methods of the RTP class.

[6] `https://search.maven.org/` accessed: 2017-03-03
[7] `https://bintray.com/bintray/jcenter` accessed: 2017-03-03

6.3.1 Container

Before going into greater detail about the implementation of a Peer and its components (Wiring, Guard, Service and Action), the used tuple space technology is presented. The tuple-based memory space is simply called *container* and is able to store and access Entries with arbitrary data payload concurrently. Containers are used for the PIC of a Peer, but also as bucket to transport Entries through a Wiring, called Entry-Collection (EC). A new EC is created after a successful Guard execution, is forwarded to the optional Service and is finally used by the Action to send the Entries to internal or external Peers.

Existing middleware

There are several space-based middleware implementations in Java 7 available - for example *Mozartspaces*[8], which was developed by the Space Based Computing Research Group of the TU Wien or EXtreme Application Platform (XAP)[9], developed by GigaSpaces. They provide high performance in-memory data spaces with several features like querying Entries with different coordination principles (like FIFO or template matching) and transaction handling.

Although these software solutions bring a lot of benefits, they have been developed for desktop and server applications. All of them have a considerable high resource consumption and are not applicable for mobile devices. Furthermore, transactions are not necessary by the design of the MPM, because each Wiring of a Peer is executed one after the other in one single thread. Various coordination principles would be a nice-to-have, but in the first mobile profile of the PM only the *Type-Coordinator* has to be available, which returns Entries of a specific type in a random order (if available).

Implementation details

A *Container* in the mobile profile of the PM has a similar interface as the JavaSpaces[10] API, but is tailored for the use in the MPM. In Figure 6.3 the class diagram of the Container reference implementation is shown.

As Entries can be written from Actions of different Peers concurrently and at the same time also read or take method calls can happen, the data structure that is used to store the Entries has to be thread-safe. Therefore, a *ConcurrentHashMap*[11] is used with the Entry's coordination-type as key. The value of each Entry in the map is again thread-safe, namely a *ConcurrentLinkedQueue*[12]. This type of queue orders elements in the FIFO

[8] `http://www.mozartspaces.org/` accessed: 2016-10-01

[9] `https://docs.gigaspaces.com/xap100/` accessed: 2016-10-01

[10] `http://www.oracle.com/technetwork/articles/java/javaspaces-140665.html` accessed: 2017-04-12

[11] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html` accessed: 2017-04-23

[12] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html` accessed: 2017-04-23

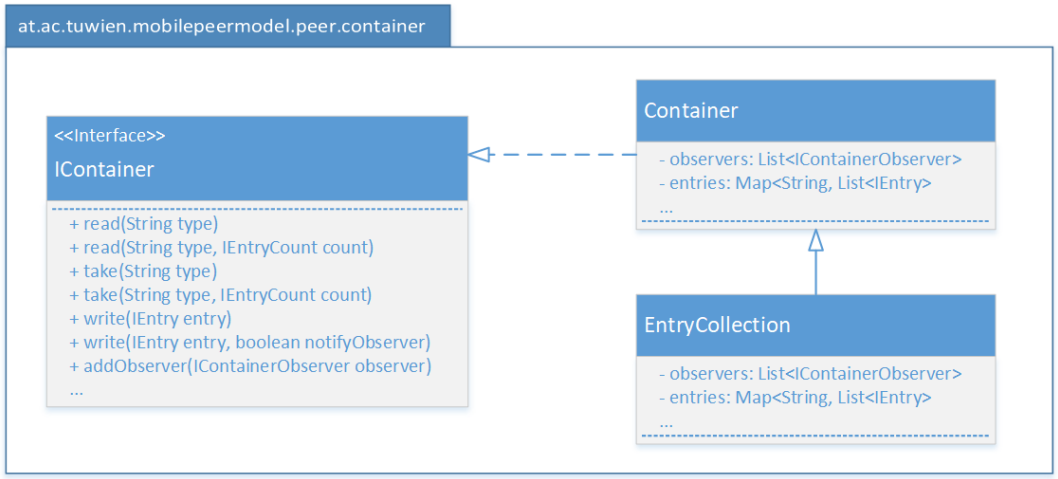

Figure 6.3: The class diagram of the Container implementation. The *EntryCollection* is a special Container that is used to store and transport Entries between parts of a Wiring (Guard -> Service -> Action).

order. However, this coordination principle is not guaranteed when fetching Entries in a Guard (READ or TAKE), because Entries might be written back after an unsuccessful Guard execution. By the definition of the MPM, the Entries don't have to be returned in a specific order anyway.

For one Container several *ContainerObserver* objects can be registered that are notified on each WRITE operation. This concept is used to inform a Peer about the modification of its PIC so that the potentially waiting thread of that Peer can be continued.

When an Entry with a non-negative Time-to-start (TTS) property is written to a container, a *Timer*[13] is created that will notify all observers of that container on the point in time this Entry becomes valid. As it is the case for the TTL property, a TTS property is an absolute timestamp, as relative timestamps are not usable when an Entry is cached on the server for some time. When an Entry with an expired TTL property is written to a container, the Entry gets wrapped into an Exception-Entry and is then automatically forwarded to the Exception-Peer with the help of a predefined Wiring (see Sections 5.5.6 and 6.3.6). During a read or take operation on a container the TTL property of all affected Entries are checked. Again, in the case that the Entry has expired, it is wrapped into an Exception-Entry and forwarded to the Exception-Peer.

The reference Container implementation can be replaced with another implementation quite easily, as the interface *IContainer* is used where possible. If a more sophisticated solution is desired in a future profile of the MPM, for example the support for different

[13] `https://docs.oracle.com/javase/7/docs/api/java/util/Timer.html` accessed 2017-04-24

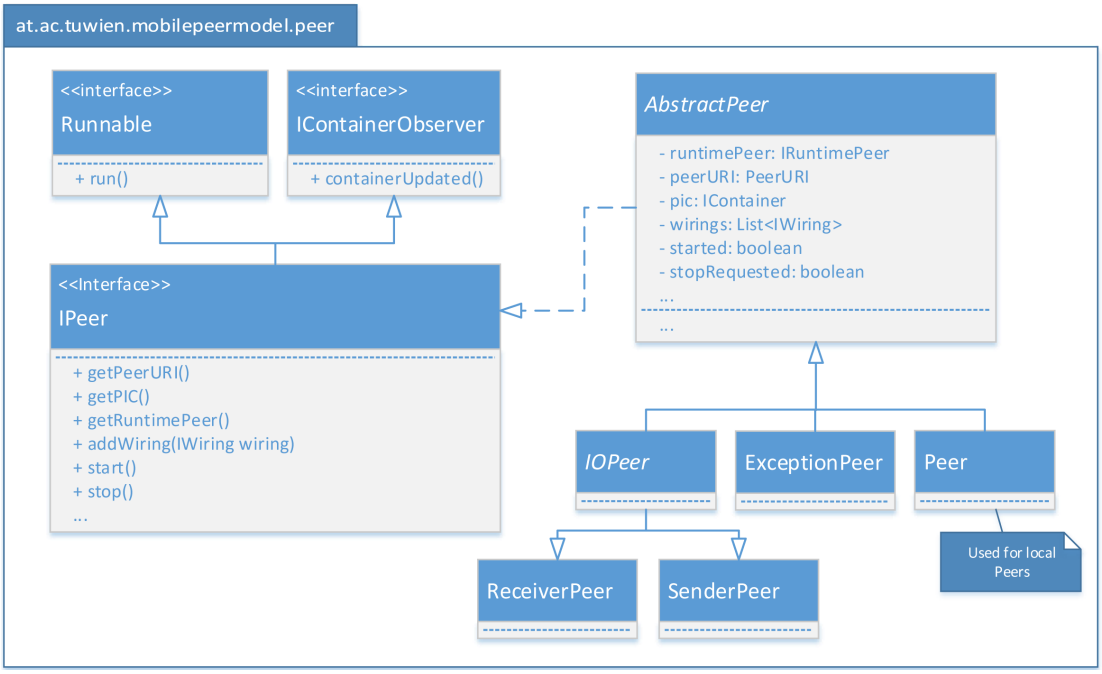

Figure 6.4: The class diagram of the Peer implementation.

coordination principles, changes on the IContainer interface as well as on the concrete implementation of the Guards and Actions have to be performed.

6.3.2 Peer

In the design chapter the scope of a Peer and the execution flow of Wirings have already been described (see Section 5.5.3). Here some details on the implementation are presented. Figure 6.4 illustrates the inheritance hierarchy of classes and interfaces of the package *at.ac.tuwien.mobilepeermodel.peer*. The class *Peer* is a concrete subclass of the abstract class *AbstractPeer* and is used if an application developer calls the *IRuntimePeer.createPeer(String peerName)* method. On the created Peer instance the developer can then add Wirings to model the desired coordination behaviour. The three system Peers (Receiver-Peer, Sender-Peer and Exception-Peer) also extend the *AbstractPeer* class, where the necessary functionality to act as a Peer in the system is implemented. In addition, predefined Wirings are added, to define the specific task of that Peers. Details about these preconfigured Wirings are described in the subsequent Sections 6.3.4, 6.3.5 and 6.3.6.

Implementing the Runnable Interface

In the Java programming language a class has to implement the special *Runnable* interface, if its instances shall be executed in an own thread. The interface defines the single no-

arguments method *run()*[14]. As each Peer in the MPM shall run in an own thread, the Peer has to implement this run() method.

On a successful RTP start, all assigned Peers are also started by calling the Peer's *start()* method. In this method each Peer creates a new thread and passes the *this* reference as argument, forcing the *run()* method of the Peer to be executed. The *AbstractPeer* class is defining the default implementation of the *run()* method, forcing all user-added Peers and system Peers to be executed in the same way.

The *run()* method is immediately entering an infinite loop, whereas in each iteration all Wirings are executed one after the other. If in one loop pass no Guard of a Wiring has been successful, the thread is suspended by calling the *wait()* method on the Peer object. The thread will remain in this state until there is a container modification in the Peer's PIC. Each Peer is implementing the *IContainerObserver* interface and registers itself as an observer for container updates of its PIC. On each *WRITE* operation, the method *containerUpdated()*, defined in the interface, is called, which will continue the waiting thread by calling the *notify()* method on the Peer's object.

6.3.3 Wiring

Figure 6.5 shows the class diagram of a Wiring and its three fundamental components. The two important implementation parts of a Wiring are the constructor and the *execute(IPeer peer)* method.

The constructor of the form *public Wiring(final IGuard guard, final IService service, final IAction action)* assigns the passed parameters to final, private member variables of the class in order to prevent manipulation of the variables after initialization. During the creation of a Wiring object also a preconfigured Action-Link with Entry count >= 1 and Entry type *exception* is added to the list of Links in the Action. This Link assures that all Entries that expire during the Wiring execution (in Guard, Service or Action) are forwarded to the Exception-Peer's PIC. Furthermore, the *ArrayLists* of Links in the Guard and Action are transformed into unmodifiable lists[15] so that the behaviour of a Guard or Action can not be changed after the Wiring's creation.

To enable reusability of Wiring objects, a Wiring is not associated with a specific Peer and can be used by different Peers at the same time. Therefore, on each execution of the method *execute(IPeer peer)* the Peer reference has to be passed as parameter so that the Wiring is executed in the right context.

Guard

The most important methods and variables of a Guard are visualized in Figure 6.5. Each Guard can have one or more Guard-Links, where the Link-Operation is restricted to

[14] `https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html` accessed: 2017-04-23

[15] `https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableCollection(java.util.Collection)` accessed: 2017-05-02

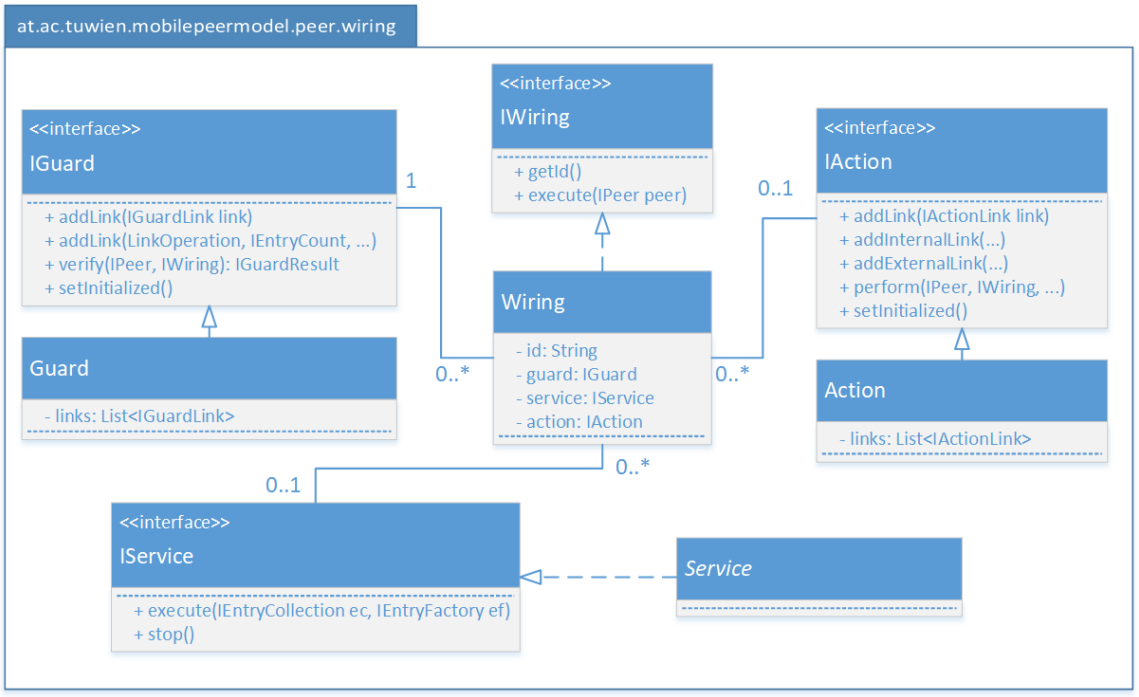

Figure 6.5: The class diagram of the Wiring implementation.

READ or *TAKE* and to be valid at least one *TAKE* operation must exist. The *count* can be defined with one of the predefined static methods of the class *EntryCount* (*exactly(int number)*, *largerEquals(int number)* or *lessEquals(int number)*). The *entryType* can be an arbitrary non-empty String. However, Guard-Links can only be added to a Guard until the Guard is added to a Wiring to avoid changes in the behaviour at runtime.

A Guard-Link is independent of a Guard and can therefore be used in different Guards. The variables of the Guard-Link are declared as *final* and can not be changed after creation to prevent modification or manipulation of Links at runtime. Furthermore, a Guard is a self-contained component and may be used in different Wirings. No reference to a Wiring or Peer is stored in the object.

The method *verify(IPeer peer, IWiring wiring)* tries to satisfy all associated Links of the Guard and returns an object of type *IGuardResult*. If a Guard can be successfully satisfied, the result delivers the boolean value true, all taken Entries (which are needed for a persistence related task) and the newly created EC containing all Entries that have been read or taken by the Guard. The EC is actually a special kind of container, similar to the PIC of a Peer, with the only difference that no observer is registered that is notified on container updates. If one Guard-Link can not be fulfilled, the result returns an empty EC and the boolean value *false*.

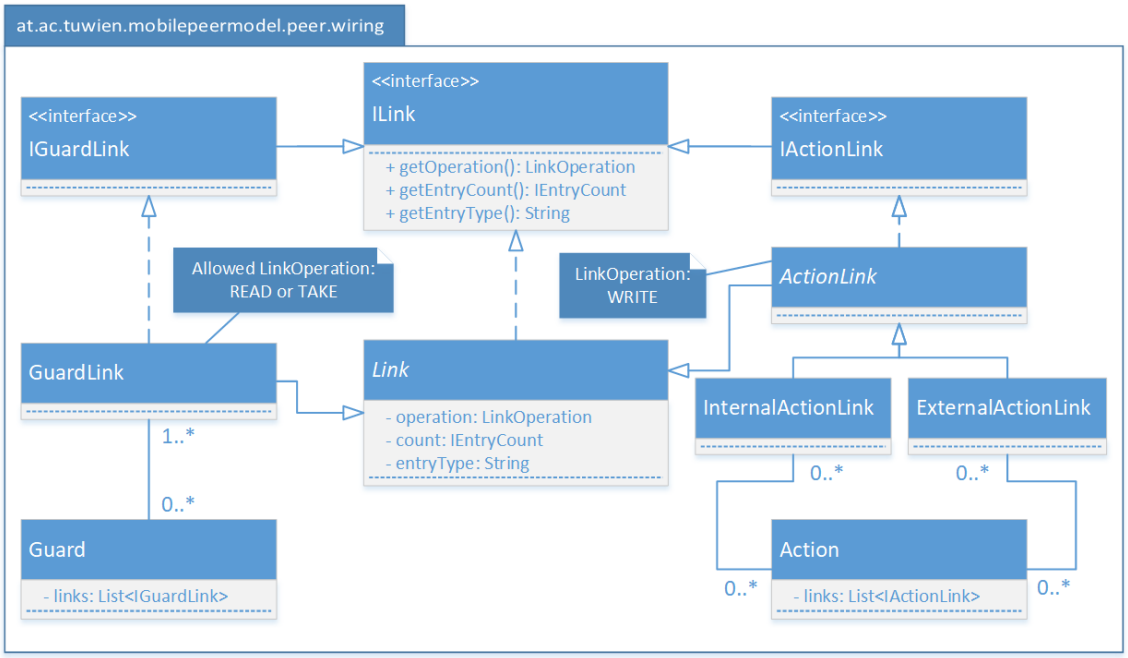

Figure 6.6: The inheritance hierarchy and relationship of Links.

Service

The *IService* interface defines the two methods *execute(IEntryCollection ec, IEntryfactory ef)* and *stop()*. If the Guard could be satisfied, the Wiring calls the *execute()* method of the Service and passes the created EC and an Entry-Factory as arguments. The provided EC may be changed by the application developer during the Service execution, because it is used as basis for the Action afterwards. The second parameter (Entry-Factory) can be used to create new Entries in the system and assures that newly created Entries are initialized with correct properties. To be exact, the factory defines the property values for *origin* and *from* with the Peer-URI of the Peer the Service is executed in. Apart from the *injectData()* method of the RTP the usage of an Entry-Factory is the only possibility to create new Entries. There is no public constructor defined for the class *Entry* to assure that no Entries exist with undefined or wrong properties. Furthermore, and similar to the Guard, each Service object is independent of any Peer or Wiring to facilitate reusability.

The execution of a Service is surrounded by a *try-catch* block to circumvent any failure or unexpected behaviour in the system. In case of an error, the thrown exception is caught, the stack trace is printed and the execution of the Wiring is continued. However, the subsequent Action might not be fully performed if the EC does not contain all required Entries after the execution of the Service. A rollback mechanism does not make sense in such a situation, as Entries that are written back to the originating PIC of the Wiring would most likely trigger the same Wiring in the next Peer's execution loop again.

As depicted in Figure 6.5, an application developer may implement the *IService* interface or extend from the provided *Service* class to define a concrete Service. When using the *Service* class the *stop()* method is already implemented so that the developer can focus on the *execute()* method. The *stop()* method of the Service is invoked from the Service's Peer just after the RTP has been stopped if the Service is currently running. In this case, the application developer should ensure the immediate termination of the Service (which is not always possible though). After the Service has been stopped, the Wiring's Action is executed as usual and available Entries are inserted into their destination containers. However, it can not be guaranteed that Entries are also sent to a remote Peer, because the server connection has been most likely closed at this point in time. Then, on a subsequent restart of the RTP and after the connection has been successfully established again, the pending Entries are sent over the wire. If the persistence layer is enabled, the state of all containers and therefore also potentially not sent entries would be reinitialized also after a complete device shutdown (see Section 6.6.5).

Action

The interface definition of an Action is shown in Figure 6.5. Similar to a Guard, an Action can have one or more Action-Links. To add a new Link, one of these three methods can be used: *addLink(IActionLink link)*, *addInternalLink(ContainerType container, IEntryCount count, String entryType)* or *addExternalLink(IEntryCount count, String entryType)*. An internal link has to be used if one or more Entries (specified with the *IEntryCount* parameter) shall be written into the PIC of the Peer the Action is executed in. For external links, again the count and the Entry type has to be defined, the actual destination Peer of a local or remote host must be specified in the concrete Entry's *dest* property. Entries of internal links are directly inserted into the PIC of the designated Peer, Entries of external links are forwarded to the Sender-Peer's PIC. The class hierarchy of Links and their relationship with the Action is visualized in Figure 6.6.

As it is the case for the Guard, the list of Action-Links is exchanged with an unmodifiable view of the list, as soon as the Action is added to a Wiring. Just before this finalization step (performed by the Wiring), a special preconfigured Action-Link is added as last element to the list. It is defined as

```
addExternalLink(EntryCount.largerEquals(1), EXCEPTION_TYPE);
```

and will take all Exception-Entries (with type *exception*) from the EC and forward them to the PIC of the Exception-Peer. With this setup, it does not matter at which point in time an Entry expires (in a Peer's PIC or during a Wiring execution), it will eventually end up in the PIC of the Exception-Peer.

Moreover, an Action-Link does not belong to one single Action and additionally each Action is a self-contained component so that they can be reused for different Actions/ Wirings.

The *perform()* method of an Action iterates through all Action-Links and constructs a list of Entries that can be actually inserted into the desired containers. If persistence is enabled, those Entries are first written in the persistence layer and simultaneously all taken Entries are removed from the persistence to avoid any inconsistencies. Only then, the Entries are actually written into the desired containers.

Simple and understandable API

In contrast to the domain-specific language (DSL) of the Peer Model, defined in [KCJ+13], the API of the Mobile Peer Model reference implementation seems clearer and easier to use. Some error prone situations, like the usage of references for containers, can be avoided. However, one reason for that is the reduced feature set of the MPM in contrast to the original one.

In the case that the application designer/developer is using the currently developed graphical modeling tool of the PM [Sch17b], the coordination-related code (creation of Peers, Wirings and their relationships) can be generated automatically. In this case, a more complicated API would not mean a considerable disadvantage in the first place. Nevertheless, simplicity and better readability can improve maintenance tasks. The modeling tool could also generate some necessary configurations, like the registration of mappings between coordination-types and Java class types. More details about the need for this data type registration process and the concrete implementation is described in [Sch17a].

6.3.4 Receiver-Peer

Figure 6.4 shows the class and interface hierarchy of Peers in the reference implementation. The *ReceiverPeer* class extends the abstract classes *AbstractPeer* and *IOPeer*. Details on the implementation of the *AbstractPeer*, which is the super type of all other Peers, was already introduced in Section 6.3.2. Therefore, the Receiver-Peer acts in the same way as any other Peer concerning execution behaviour. The only difference is the automatically added preconfigured Wiring, illustrated in Figure 6.7. The Guard of the Receiver-Peer's single Wiring takes one Entry of type *START_RECEIVER* and executes the Service *ReceiverService*. One Entry of that type is inserted into the PIC of the Receiver-Peer on the first Runtime-Peer start to activate the ReceiverService once.

ReceiverService

The ReceiverService only registers a message processor for incoming Entries on the communication layer. The message processor callback is then called for each incoming Entry. The implementation of the callback function is simple: check the *dest* property of the Entry and then insert it into the specified local Peer. As simple validation step it is checked if the addressed Peer has at least one Guard-Link registered in any of its Wirings with the type of the received Entry. If no such Link exists or if no local Peer with the desired destination Peer-URI is available in the list of local Peers, currently just a log

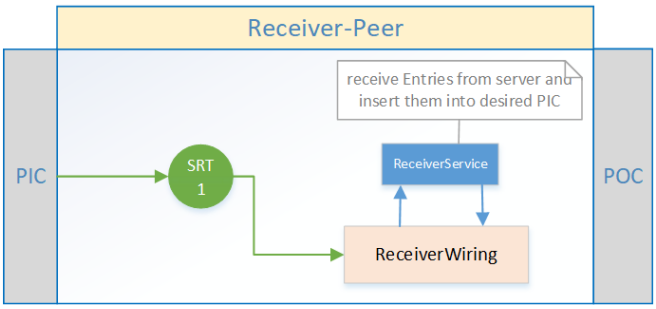

Figure 6.7: The graphical notation of the Receiver-Peer with its single Wiring. The only objective of the Wiring is to start the *ReceiverService* once, which registers a message listener for incoming Entries.

message (WARN) is printed and the received Entry gets ignored. This application layer security feature has been introduced to prevent memory overflows in peer-in-containers if an attacker tries to flood a host with arbitrary Entries that would never be processed by a Guard.

6.3.5 Sender-Peer

Similarly to the Receiver-Peer, the Sender-Peer has a preconfigured Wiring with one Guard and one Service, see Figure 6.8. The special characteristic of the single Guard-Link is that no Entry type is defined, because the SenderService shall be able to send every Entry to a remote or local Peer, regardless of its Entry type and data payload. The count of the link is defined as *GREATEREQUALS(1)*, meaning that in one Wiring execution all Entries of the Sender-Peer's PIC will be processed.

SenderService

The Service iterates over all Entries that are delivered via the EC and decides if the Entry can be sent internally, by just writing the Entry to the PIC of a local Peer, or if the *send()* method of the communication layer has to be called to actually send the Entry over the wire. Before that, it is checked if the *dest* property is correctly set and formatted. If notifications are enabled, a further Entry is transmitted to the Notifier-Peer, immediately after an Entry is sent to a remote host, in order to wakeup the receiving device.

Send exceptions

If an Entry cannot be sent (i.e. because no active internet connection is available), it is cached until there is a call to the RTPs *onNetworkConnected()* method. In this case the RTP notifies the *Sender-Peer* about the re-established internet connection and the Sender-Peer forwards this information to the SenderService, which, after all, resends the

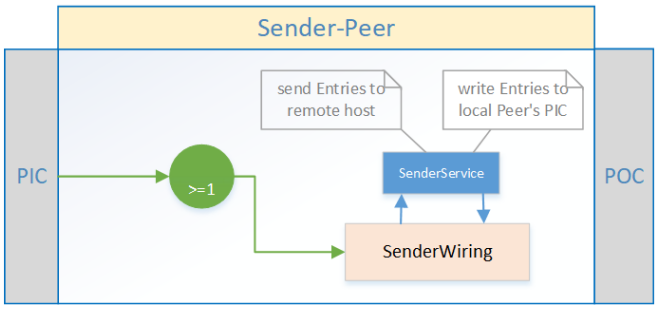

Figure 6.8: The graphical notation of the Sender-Peer with its single Wiring. The responsibilities of the SenderService are the insertion of Entries to the PIC of a local Peer (if the Entry's destination Peer is within the current runtime) and the transmission of Entries to remote hosts.

pending Entries. This special behaviour is not modeled via a dedicated Wiring, although it would be possible of course.

In Section 5.6.7 the procedures to guarantee data consistency and how to recover data in the case of system failures are described. To ensure data consistency in every failure situation (also during the send process), distributed transactions would be necessary. As illustrated in more detail in the design section with Figure 5.12, data consistency during the send process is not guaranteed, but failures immediately before or after the transmission can be detected. The application developer can then decide how to proceed (i.e. resend the possibly not sent Entry on the next RTP start or ignore the failure situation). Such an exceptional situation is very unlikely and can only happen if the application is killed unexpectedly or the device has to perform an abrupt shutdown during a low battery condition.

6.3.6 Exception-Peer

The Exception-Peer is the third automatically provisioned system Peer that comes with a preconfigured Wiring. In the design chapter the concept of the Exception-Peer and the scope of an *Exception-Entry* have already been introduced. Also the two currently supported exception types *TTL_EXCEPTION* and *POTENTIAL_SEND_EXCEPTION* were presented.

Interface of the Exception-Peer

To allow an application dependent exception handling for developers, the interface of the Exception-Peer is extended with two methods. The first method (*setExceptionService(IService service)*) defines the *Service* of the Exception-Peer and the second one (*setExceptionServiceAndAction(IService service, IAction action)*) sets the *Service* and the *Action*. The Guard of the single Wiring of the Exception-Peer is already defined and cannot be changed - exactly one Entry of type *exception* is taken from the PIC. If

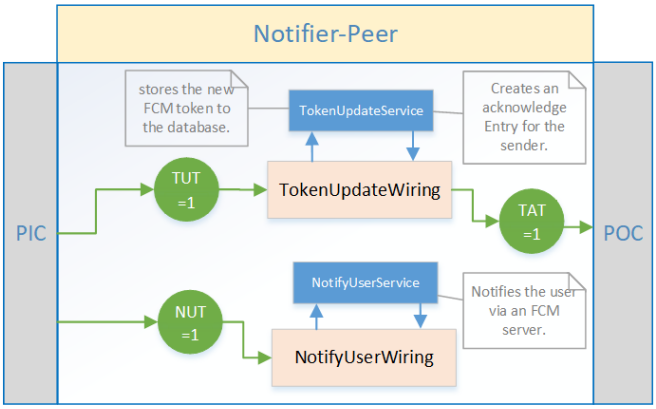

Figure 6.9: The graphical notation of the Notifier-Peer with its two Wirings.

no Service or Action is specified by the developer, a default Service is used that is just doing nothing (NO-OP). Consequently, the Entries are removed from the PIC, because it would not make sense to maintain them in the PIC if there is no applications specific exception handling anyway. Also this avoids the overflow of the PIC in very error-prone applications.

The Wiring of the Exception-Peer is the only Wiring in the system that has no automatically added Action-Link of type *exception*. This Action-Link is responsible for forwarding Exception-Entries, that were created i.e. because of a TTL-expiration, to the Exception-Peer. This restriction prevents an infinite cyclic loop in the Exception-Peer if an exception occurs during the Wiring of the Exception-Peer. Apart from that, Exception-Entries cannot expire by definition.

6.4 Notifier-Peer

The notification approach used in the MPM framework was introduced in Section 5.3. The so-called *Notifier-Peer* component that is shipped together with the MPM-Core notifies other MPM hosts in the system about incoming messages. The Notifier-Peer is built upon the MPM framework and has one single local Peer with two dedicated Wirings and Services. The first Wiring is responsible for storing/updating the FCM token of a host and the second one for sending notification requests to the Firebase server (see Figure 6.9). In the reference implementation the *Spring framework* is used to facilitate the work with the FCM token database. More details about the Notifier-Peer and instructions on how to correctly set up and run the *Spring Boot* application are provided in Section 7.1.1. The second step of the notification infrastructure setup process, i.e. how to create an FCM project via the Google FCM console and which steps are necessary to configure the Android application, is described in Section 6.6.3.

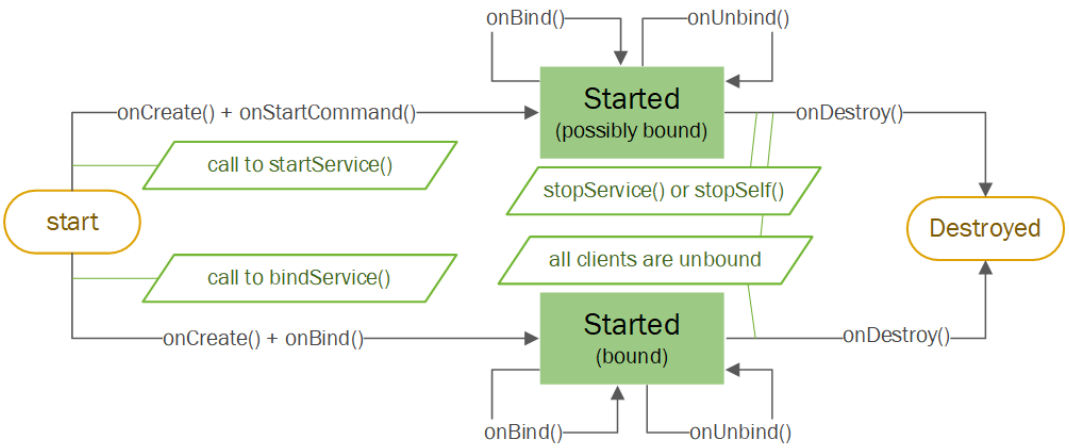

Figure 6.10: The lifecycle of an Android Service. A Service can either be started explicitly with a call to *startService()* or implicitly when a component wants to bind to the Service.

6.5 Registration

The registration component is implemented as a Java Spring web application that exposes some REpresentational State Transfer (REST) endpoints. The black arrow R1 in Figure 5.1 of the design chapter illustrates a REST-call (POST) to register a new user. The web method associated with that call then inserts the new username and the password into the XMPP user database if no such user exists. If the registration succeeded, the user will get the HTTP-Status 201 (CREATED) as result. Consequently, the user can establish a connection to the XMPP server.

6.6 Mobile design considerations

In Section 5.6 important insights into the Android platform and its application components have been given. In the following subsections the provided Runtime-Service and the two FCM Services are described in more detail, including a description of the interfaces and which steps have to be performed by an application developer in order to use the provided components correctly.

6.6.1 Android Services

This section introduces the lifecycle of an Android Service and gives insights into concrete API calls and important callback functions.

Each application component that has a valid reference to a Context object can start a Service. If desired, this is even possible by components from other applications. To interact with the Service a component can also set up a two-way binding. Figure 6.10 shows the two possible lifecycles of an Android Service. It can be either started by a call

to *startService()* or *bindService()*. Both methods are available on object references of type *Context* (remember that each Activity and Service extends from the abstract class *Context*).

The way in which the Service is started has influence on the method callbacks that are executed on Service startup and also affects how the Service can be terminated. If a Service is started with the *bindService()* method, the Android system will bind the Service to a specific component (usually an Activity or an immediate member variable). This also allows communication between the component and the Service. Once created, further components can bind to the same Service and it will run as long as there is at least one component bound. To close the connection to a Service the method *unbind()* must be called.

A Service that is started with the explicit *startService()* method will run until there is a call to *Context.stopService()* from outside or to *Service.stopSelf()* from within the Service implementation. Moreover, the Service can be *bound* to various components, as it is the case for a Service that was started with *bindService()*. If bound, the Service cannot be stopped until all components have closed their connections again. However, in this constellation a call to *stopService()* or *stopSelf()* is still needed, because the Service was started explicitly.

6.6.2 Runtime-Service

The RTP can only run in a reliable way in the background if it is executed in the context of an Android Service. This process is only stopped by the Operating System (OS) if resources are very limited and they are needed by other foreground Activities or Services.

Interface of the Runtime-Service

Figure 6.11 shows the class diagram of the provided Android Runtime-Service and some of its related components. In this section some important methods are described and the necessary steps of an application developer to use and configure a concrete subclass of the Service in an Android app are demonstrated.

The abstract class *MPMRuntimeService* extends the system-defined component *Service* that comes with predefined callback methods to control its execution and state.

The *onCreate()* method called immediately after the Service object is instantiated by the OS and is the proper place to initialize important member variables. The method is comparable with the constructor of a normal Java class, however, Android components (like an Activity or Service) should not be created with their constructors. Instead, the Android system forces application developers to use methods of the Android *Context* object (see Section 5.6.2) to create components, in order to correctly register it in the system. Important methods on the Context object are for example *startActivity (Intent intent)*, *startService (Intent intent)* and *bindService(Intent intent, ...)*. In the *onCreate()* method, the provided *MPMRuntimeService* class initializes a *BroadcastReceiver* with

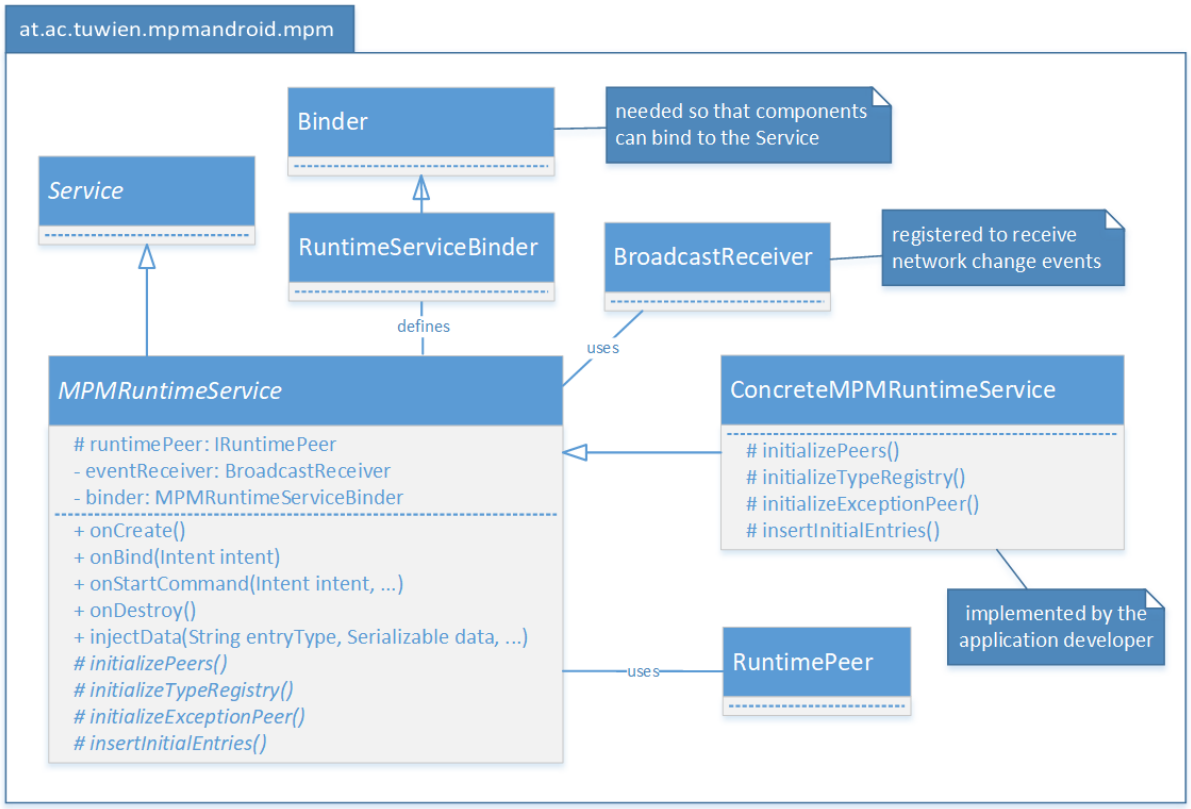

Figure 6.11: The class overview of the abstract *MPMRuntimeService* and its related components.

the two *IntentFilter*[16] actions *ConnectivityManager.CONNECTIVITY_ACTION* and *MPMConstants.PREF_FCMTOKEN_UPDATED*. The first is emitted by the system on network change events and the second is sent out by the provided *MPMFirebaseInstanceIDService* when the unique FCM token got rotated by the FCM engine (see Section 6.6.3). Also the *IBinder* is created at this point. If a Service allows clients to interact with it, the method *onBind(Intent intent)* has to be defined, which specifies how other components can communicate with the Service. As the *MPMRuntimeService* allows client communication, it returns an object that extends the system class *Binder* with the method *getService()*. This method returns an object of type *RuntimeServiceBinder*, which defines the methods bound components can call, for example *injectData(...)* (see Figure 6.11). Finally, the *onCreate()* method registers a concrete Android PersistenceManager for the MPM framework.

Further important callback functions of a Service are the methods *onStartCommand()* and *onBind()* - both are called by the system immediately after *onCreate()* has returned.

[16] `https://developer.android.com/guide/components/intents-filters.html` accessed: 2017-05-10

As already discussed in the previous section, there are two different ways how a Service can be started. This also influences the lifecycle of the Service and the methods that are called during its creation (see Figure 6.10).

The method *onStartCommand(Intent intent, ...)* is executed if the Service is created explicitly via a call to *startService()*. This method will actually start the Runtime-Peer and afterwards returns an integer value that indicates the behaviour of the system if the Service had to be terminated unexpectedly. In the provided *MPMRuntimeService* implementation the returned value is *START_STICKY*, which will force the system to restart the Service as soon as possible if it was terminated due to a resource bottleneck. In such a case, the state of all containers in the MPM is preserved (if persistence is enabled) and the RTP will continue its work immediately after the restart of the Service. The method *onBind()* is called every time a component wants to bind to the Service and the Service will run only as long as there is at least one component bound to it.

Both methods (*onStartCommand()* and *onBind()*) receive an *Intent*[17] parameter, which can be supplied with an optional username and password property, so that the RTP can authenticate against the server. This data is stored in the Android's *SharedPreference* store, so that subsequent Runtime-Service starts do not need username or password arguments.

As next step, either within *onStartCommand()* or *onBind()*, the Runtime-Peer singleton is created and the username and password is set. Then, some abstract methods of the *MPMRuntimeService* are called which have to be overridden by application developers in a concrete subclass of the *MPMRuntimeService*. Those methods are:

- **initializeTypeRegistry():** Remember that an Entry may have an arbitrary payload, which means that any user defined Java class may be used. To enable a functioning and performing serialization and deserialization of Entries, each possible coordination-type of an Entry has to be mapped to a concrete data-type, representing the Java class of the Entry's payload object. More details about the type registry and the serialization and deserialization mechanism are shown in the thesis of Jörg Schoba. Listing 7.3 in Chapter 7 demonstrates the usage of the *MPMRuntimeService.initializeTypeRegistry()* method.
- **initializePeers():** Here all coordination-related code must be inserted by the application developer. In practical terms, that means that all additional Peers and Wirings shall be created here and have to be added to the RTP. The RTP instance is a protected member variable of the *MPMRuntimeService* class and can therefore be used in any subclass with a usual variable reference. Again, in Chapter 7 an example implementation of this method is shown (see Listing 7.4).
- **initializeExceptionPeer():** This protected method is the right place to add a Service and Action to the internal Exception-Peer. An application developer

[17] `https://developer.android.com/guide/components/intents-filters.html` accessed: 2017-05-10

does not have to override the method, as it is not declared as *abstract* in the *MPMRuntimeService* class. If the method is not overridden and no Service is added to the Exception-Peer, a default Service just takes all Exception-Entries from the PIC and does nothing.

- **insertInitialEntries():** Entries that shall be inserted into a specific Peer on the very first RTP start must be defined in this method. This might be necessary to trigger particular tasks only once. If the persistence layer is enabled, these Entries are not inserted again after the first RTP start, even after a complete device restart. This does not work with persistence disabled, though.

To receive FCM push notification messages (see details about the needed Android Services in Section 6.6.3), each RTP host has to transmit its current unique FCM token to the Notifier-Peer. In the protected method *initializeTokenUpdatePeer()* the Peer with name *TOKENUPDATEPEER* is created and two Wirings necessary for the token update process are defined. Specifically, one Wiring with Guard, Service and Action is responsible for sending the FCM token to the Notifier-Peer and the second one takes an Entry with type *tokenupdate_ack* from the PIC that is received from the Notifier-Peer after a successful token update and executes the Service *TokenAckReceivedService*. This Service simply removes a boolean value in the Android SharedPreferences that determines if the FCM token has been updated by the FCM engine, so that the token update Entry won't be sent again to the Notifier-Peer on the next RTP start.

How to use the Runtime-Service

The *MPMRuntimeService* class is delivered via the MPM-Android library (see Figure 6.1 in Section 6.2 on the left side). An application developer has to perform the following steps to use the Service:

- **Add library dependency:** In the *build.gradle* file the dependency to the name of the MPM-Android library has to be defined, so that the library can be downloaded by the gradle build tool. When using the recommended Integrated Development Environment (IDE)[18] *Android Studio* or *IntelliJ IDEA*, the right *build.gradle* file is located in the *app* module in the Android project. The single line an application developer has to add to the *dependencies* section is for example: *compile 'tuwien.ac.at:mpmandroid:2.0.0'*. The library will be publicly available in at least one of the popular repositories *maven*[19] or *jcenter*[20], which are used to store artifacts and dependencies of varying programming languages.
- **Extend the abstract MPMRuntimeService:** The delivered abstract *MPMRuntimeService* class has to be extended in order to create a concrete Android

[18] `https://developer.android.com/studio/index.html` accessed: 2017-05-19
[19] `https://search.maven.org/` accessed: 2017-05-19
[20] `https://bintray.com/bintray/jcenter` accessed: 2017-05-19

Service. All three abstract methods, *initializeTypeRegistry()*, *initializePeers()* and *insertInitialEntries()* have to be implemented. The developer may also override the methods *initializeExceptionPeer()* and *initializeTokenUpdatePeer()*.

- **Register the MPMRuntimeService:** The concrete Service class has to be declared in the file *AndroidManifest.xml* in order to let the system instantiate it at runtime (see Chapter 7 for a sample code).

- **Start the RuntimeService:** Basically, the concrete Runtime-Service can be started in two different ways, by calling the *startService(Intent intent)* method or by binding to the Service in order to communicate with it (*bindService(Intent intent)*).

- **Stop the RuntimeService:** Depending on how the Service was started, it must either be stopped explicitly by calling *stopService()* or it is automatically stopped by the system when all bound components have closed the binding (*unbind()*). It is the responsibility of the application developer to determine the right lifecycle, which depends on the concrete use case. Sometimes the Runtime-Service only has to be started while a visual application component is shown, where binding to the Service is surely the better approach. Other applications, however, might require the Service to run long-term in the background. In this situation the developer should start and stop the Service explicitly. Nevertheless, a developer always has to bear in mind that a running Runtime-Peer might slow down the whole system and drain the battery unnecessarily, among other things because a permanent connection to the server has to be maintained. Also starting and stopping the Runtime-Peer for several times successively might not be the ideal behaviour. That is because on each Runtime-Peer start several new threads are created, implying a considerable additional resource consumption.

Additionally, the already implemented callback functions of the *MPMRuntimeService* can be overridden by the developer, for example the *onBind()* method to return another interface in order to call further methods on the Service or the *onCreate()* to initialize additional resources used in the Service. In such a case, the developer has to ensure that the *super()* method is called, which initializes mandatory functionalities (like the registration of system event listeners).

Further necessary configuration steps, especially regarding push notifications, can be found in Section 6.6.3. Demonstrated with the example of a P2P messenger, Chapter 7 illustrates some descriptive sample code of coordination logic. This sample application will be provided in the MPM repository and can be used easily as starting point for new Android applications. Then, only the coordination part and the server configuration have to be replaced.

Further information

The Runtime-Service sends a local broadcast message that informs the user about the success or failure of the Service (and RTP) startup. If the RTP could be started successfully, the delivered Intent[21] has an *Extra* with type *MPMStartResult.SUCCESS*. Further possible result types that might be sent out during a Runtime-Service start are: *AUTHENTICATION_EXCEPTION*, *COMMUNICATION_EXCEPTION* and *RUNTIME_EXCEPTION*. Application developers can register for those messages and react as desired. For example in a messaging app, the user should be notified about wrong credentials (AUTHENTICATION_EXCEPTION is thrown) on the login page. Those exceptions have to be emitted via a broadcast message, because Android Service callbacks shall not be directly called by an application developer and exceptions that occur during such a callback can not be caught via the standard Java try-catch mechanism.

The lifecycle of the RTP is also shown in Figure 5.8 of the design chapter. Also noteworthy at this point is that the Service's callback functions are called on the main UI-thread. Resource consuming initializations and tasks would slow down the whole user experience drastically, because the UI might not react to any user input in the meantime. Therefore, the RTP is created and initialized in an own thread. The startup of the RTP must be performed in a background thread anyway, because the Android system does not allow network operations on the main thread and would throw a *NetworkOnMainThreadException*.

Another possibility to decrease the probability that a Service is terminated by the operating system on low memory, is to transform the Service into a foreground Service, as described in the Android developer reference[22]. While such a Service is running, the user is aware of the execution, because a notification is shown in the Android notification bar. The system will very unlikely stop such a Service, as it is seen as very important to the user (see also the Android process importance hierarchy in Section 5.6.1). To configure a foreground Service the method *startForeground()* has to be called from within the Service. Therefore, if desired, the *onCreate()* or *onBind()* method has to be overridden by the application or framework developer in order to use this feature.

6.6.3 FCM Services

In the design chapter in Section 5.6.5 the background knowledge and the need for push notifications were introduced. In order to save processing resources by stopping the Runtime-Service if it is not needed, but still getting notifications about incoming messages, the MPM-Android library comes with a predefined notification approach that requires a minimal configuration effort for application developers. Firebase Cloud Messaging (FCM)[23] simplifies the delivery of push notifications to Android, iOS and web applications.

[21] `https://developer.android.com/guide/components/intents-filters.html` accessed: 2017-05-10

[22] `https://developer.android.com/reference/android/app/Service.html` accessed: 2017-05-10

[23] `https://firebase.google.com/` accessed: 2017-05-20

Furthermore, this approach is very battery-efficient, because the Android device only has to maintain one single connection to a server, which can be used by different applications.

Google delivers an AAR library project containing all necessary classes and Android Services that are needed to use FCM. The two mandatory Android Services (*FirebaseInstanceIdService* and *FirebaseMessagingService*)[24] an application developer has to implement, are described in the following sections.

FirebaseInstanceIdService

The first Service that has to be implemented for an Android app that makes use of FCM notifications must be a subclass of the *FirebaseInstanceIdService*[25] from the package *com.google.firebase.iid*. This Service is an extension of a basic Android Service with the method *onTokenRefresh()*, which is called by the system each time the FCM token changes. This will happen on application installation, if the app data is cleared or the FCM SDK decides to rotate the token. The token itself is a unique ID to identify each client app instance.

Usually, the just generated token is sent immediately to an application server for later notification activities. However, this is not possible in the MPM framework, because the Notifier-Peer, where the FCM messages are stored, might not be reachable from the current device (remember the Notifier-Peer is a special host in the system also built upon the MPM framework). This is because the *hostname* and *password* might not to be set for the Runtime-Peer at the moment of token creation and a connection establishment to the XMPP server and consequently message delivery to any host is not possible. Therefore, only a key-value pair with name *PREF_FCMTOKEN_UPDATED* is set to *true* in the Android SharedPreference store. On each successful Runtime-Peer start, this preference is evaluated and if the value is *true*, the token is delivered to the Notifier-Peer. To achieve this, a special Entry with coordination-type *tokenupdate* and the token as payload-data is injected into the RTP. The *dest* property is set to the automatically added local Peer *TOKENUPDATEPEER* that is responsible for transferring the token to the Notifier-Peer. In the *onTokenRefresh()* method of the MPM implementation also a broadcast message with an *Intent-Extra*[26] *PREF_FCMTOKEN_UPDATED* is sent out. A running Runtime-Service has registered for such broadcast messages (in the *onCreate()* method, see Section 6.6.2) and would immediately initiate the token update process in that case.

[24] `https://firebase.google.com/docs/cloud-messaging/android/client` accessed: 2017-03-20

[25] `https://firebase.google.com/docs/reference/android/com/google/firebase/iid/FirebaseInstanceIdService` accessed: 2017-03-19

[26] `https://developer.android.com/guide/components/intents-filters.html` accessed: 2017-05-10

FirebaseMessagingService

The second mandatory FCM Service actually deals with receiving FCM notifications, as described in the official Firebase Reference[27]. In principle, there are two different types of messages that can be sent from an FCM server to a client: *notification* messages and *data* messages[28]. A simple *notification* message contains up to 2 KB payload with a set of predefined keys and is automatically displayed as an Android notification on the user's device. When receving that kind of notification message, the callback function *onMessageReceived()* is only executed if the application is currently in the foreground. A *data* message may deliver up to 4 KB of data (custom key-value pairs) and forces the *onMessageReceived()* method to be invoked. For that type of notification message the developer is responsible for taking further steps. For example, the developer may decide to display a message in the Android notification bar if a specific property is set in the payload of the received message or the received data is simply stored in the local app storage for later usage. Note, that iOS apps in the background will only receive the data message immediately, if the *content_available* property of the FCM notification message is set to *true*.

The predefined implementation of the *onMessageReceived()* callback method is simple, by just starting the Runtime-Service. In Section 6.6.2 the method *onStartCommand()* was described that requires an integer return value. If the Runtime-Service is explicitly started from the user outside of the *FirebaseMessagingService*, the method will return the system-defined value *START_STICKY*, forcing the Runtime-Service to restart immediately after it has been killed by the operating system. When the Service is started automatically after an FCM push notification retrieval in the method *onMessageReceived()*, the predefined integer *START_NOT_STICKY* is returned meaning that the Service will not be recreated if killed by the OS. The start of the Runtime-Service will automatically start the RTP, which in turn will establish the XMPP connection to receive the pending message. The credentials for the XMPP connection had been already set before, because otherwise the FCM token registration on the Notifier-Peer would not have been possible. In the case that the Runtime-Service is currently running and an FCM notification is received, the *startService()* call will just have no effect. The Runtime-Service will ignore consecutive Service starts.

How to use the FCM Services

Both Services (*MPMFirebaseInstanceIdService* and *MPMFirebaseMessagingService*) are delivered with the *MPM-Android* library artifact and app developers only have to perform the following simple steps to use the FCM notification approach[29]

[27] `https://firebase.google.com/docs/reference/android/com/google/firebase/messaging/FirebaseMessagingService` accessed: 2017-03-20

[28] `https://firebase.google.com/docs/cloud-messaging/concept-options` accessed: 2017-03-22

[29] `https://firebase.google.com/docs/cloud-messaging/android/client` accessed: 2017-05-21

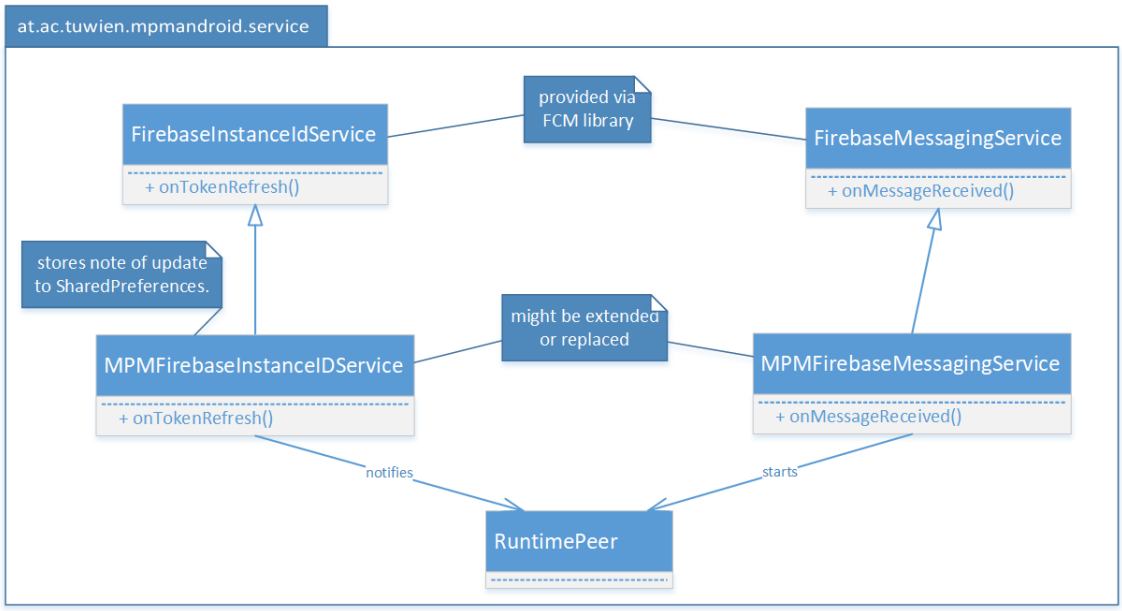

Figure 6.12: The class overview of the provided FCM Services.

- **Create Firebase project:** There is a good description on the official website on how to add *Firebase* to an Android project[30]. The easiest approach is to simply use the pre-existing wizard in the Android Studio. Otherwise, a Firebase project has to be created online in the Firebase Console[31]. The thereby generated *google-services.json* file has to be downloaded and copied into the Android projects *app* module.

- **Add FCM dependency:** The newest version of the FCM messaging library has to be added to the dependency section of the app-level build.gradle file: *i.e. compile 'com.google.firebase:firebase-messaging:10.2.6'*

- **Add Services to AndroidManifest.xml:** The two provided Services from the MPM-Android library have to be registered in the *AndroidManifest.xml* configuration file of the Android app (see Listing 7.9).

After performing those three steps, the MPM RTP will be successfully notified on any incoming message, presuming that a correctly configured Notifier-Peer instance is deployed.

[30] `https://firebase.google.com/docs/android/setup` accessed: 2017-05-21
[31] `https://console.firebase.google.com/` accessed: 2017-05-21

Overridable Services

With the steps described in the previous section, an application developer has a working notification mechanism for the MPM. However, it may be necessary to change the provided FCM Services. For example, the developer might want to use the FCM messages also for additional use cases in the application or the developer may decide to first show a notification in the Android notification bar and only if the user clicks on it, the Runtime-Service or the app is started (in order to let the user control the background Service execution). In such a situation the *MPMFirebaseMessagingService* can be exchanged or extended.

Another possibility why a developer might not want to use the FCM notification approach is because of data protection reasons, as Google is hosting the FCM servers and can record the whole data flow of the Peer Model. To use a different notification approach, the FCM Services simply should not be registered in the *AndroidManifest.xml* and the developer has to implement an own mechanism. Furthermore, the Notifier-Peer is build upon the MPM framework and the FCM-Services of the preconfigured Wirings can be easily exchanged, so that the different approach can be implemented without much additional effort.

Further information

When an Android Service is started, the fully qualified class name of the Service has to be specified. For example, to start a concrete Runtime-Service with name *MyRuntimeService*, defined in package *at.test.mpm*, the Service must be started by invoking the method *startService(new Intent(this, at.test.mpm.MyRuntimeService.class))*. As the developer has to extend the abstract *MPMRuntimeService* class at development time, the concrete subclass has to be called by the *MPMFirebaseMessagingService* to start the concrete Service. However, the name of the concrete subclass is not known when compiling the MPM-Android library. Therefore, the class name is stored to the Android SharedPreference store when the Runtime-Service is created at the first time. On incoming FCM messages the fully qualified class name of the Runtime-Service is loaded from the local store and the Service can be started.

Unfortunately, this approach has one drawback, namely, if the user wipes the local data storage of the app, the concrete class name is also removed from the Preference store. Consequently, when receiving subsequent FCM notifications the concrete Service can not be started from the *MPMFirebaseMessagingService*, because the class name is not available. In such a case, the application has to be started again explicitly by the user to initialize the class name again. Otherwise, to avoid this behaviour, the app developer has to override or exchange the *MPMFirebaseMessagingService* and start the Runtime-Service with the concrete class name. The reason why this approach is still implemented in that way is, because the developer does not have to concern about the FCM implementation at all and only has to register the Service in the manifest.

6.6.4 User interaction

Peers and Wirings have to be added to the RTP before it is started. At that point in time, GUI components, which have to be notified about Entries that are being processed during the MPM execution, might not have been initialized. Therefore, and as already described in Section 5.6.6, one possibility to send data from an MPM Service to a GUI component is the usage of so-called *broadcasts*[32].

This communication mechanism is similar to the *publish-subscribe* messaging pattern. Data packages are transported within an *Intent*[33] that is identified by a well-defined String identifier, similar to a *topic* in the publish-subscribe pattern. In order to receive some desired data packages, components can register themselves on a *BroadcastManager* with a specific String. For both actions, send and receive, a *Context* is needed. Therefore, an MPM-Service that wants to communicate with a GUI component via broadcasts must have a reference to a *Context* object. Because the Runtime-Service extends from the system class *Context*, this object can be easily passed to each MPM-Service via its constructor. The Context object is also necessary to make use of other application- or system-specific features inside an MPM-Service, like i.e. storing data to a local database.

If broadcasts are not intended to go beyond the application boundaries, a *local* broadcast should be performed. It is more efficient, because no inter-process communication is needed and, even more important, security considerations can be neglected, as local broadcasts are only sent out to components of the current application. When using the *LocalBroadcastManager* to send messages, its method *sendBroadcast(Intent intent)* returns a boolean value, determining if there was at least one receiving component. In the case that the method returned *false*, meaning that no component was receiving the message, an application developer can realize that the application is not in the foreground at the moment and could store the Entry (or its data) to a local file or database for later retrieval. When using this mechanism it is very important, though, to *unregister* a previously registered *BroadcastReceiver* to avoid losing messages on outdated receivers.

The code snippet in Listing 6.1 is taken from the method *onTokenRefresh()* of the delivered *MPMFirebaseInstanceIDService* that sends out a broadcast message to inform the possibly running Runtime-Service about the rotation of the FCM token. In this example no payload is added to the Intent, because the FCM token can be obtained via a call to the static method *FirebaseInstanceId.getInstance().getToken()* at the receiver, which has been registered in the Runtime-Service.

```
Intent newTokenIntent = new Intent(MPMConstants.FCMTOKEN_UPDATED);
LocalBroadcastManager.getInstance(this).sendBroadcast(newTokenIntent);
```

Listing 6.1: Everytime the FCM engine generates a new token, a broadcast message is emitted.

[32] https://developer.android.com/guide/components/broadcasts.html accessed: 2017-03-25

[33] https://developer.android.com/guide/components/intents-filters.html accessed: 2017-05-10

In the *onCreate()* method of the *MPMRuntimeService* a *BroadcastReceiver* is registered (see Listing 6.2).

```
BroadcastReceiver receiver = new RuntimeServiceReceiver();
IntentFilter filter = new IntentFilter();
filter.addAction(ConnectivityManager.CONNECTIVITY_ACTION);
filter.addAction(MPMConstants.FCMTOKEN_UPDATED);
this.registerReceiver(receiver, filter);
```

Listing 6.2: Registration of the BroadcastReceiver in the MPMRuntimeService.

First, the concrete RuntimeServiceReceiver, that implements the method *onReceive(Context context, Intent intent)*, is created. Then, the two topics that this receiver is registering for are defined (ConnectivityManager.CONNECTIVITY_ACTION and MPMConstants.FCMTOKEN_UPDATED). Finally, to actually register the BroadcastReceiver the method *registerReceiver()* has to be called on a Context object. Here, the *this* variable refers to the current *MPMRuntimeService* object, which is a subtype of Context. On an incoming message, the *onReceive()* method is invoked by the LocalBroadcastManager. For example, the *onReceive()* method of the *RuntimeServiceReceiver* is invoked after the code in Listing 6.1 is executed. In that case, a new Entry is injected into the RTP with the FCM token as payload data in order to update the token on the Notifier-Peer host. Of course, inside the *onReceive()* method the actual String identifier, ConnectivityManager.CONNECTIVITY_ACTION or MPMConstants.FCMTOKEN_UPDATED, has to be checked before.

6.6.5 Persistence

A persistence layer, responsible for reflecting the state of the RTP's containers to a non-volatile memory, is a necessary requirement to provide a reliable product. Further reasons for implementing a persistence layer are described in detail in Section 5.6.7.

Interface of the persistence layer

As pointed out above, a persistence solution had to be integrated into the core framework. Particular methods of the persistence layer interface (see Figure 6.13) are called at relevant parts during the RTP execution. However, as sudden terminations of the Runtime-Peer are only expected in an Android environment, a concrete persistence implementation has been only developed for the Android platform at the moment. Applications that only use the core framework (see MPM-Core on the right side of Figure 6.1) have to implement and register an additional persistence implementation to store data permanently on the persistent storage.

Figure 6.13 shows the persistence related packages of the MPM-Core and the MPM-Android library and their relationships. In the following list some important methods are enumerated.

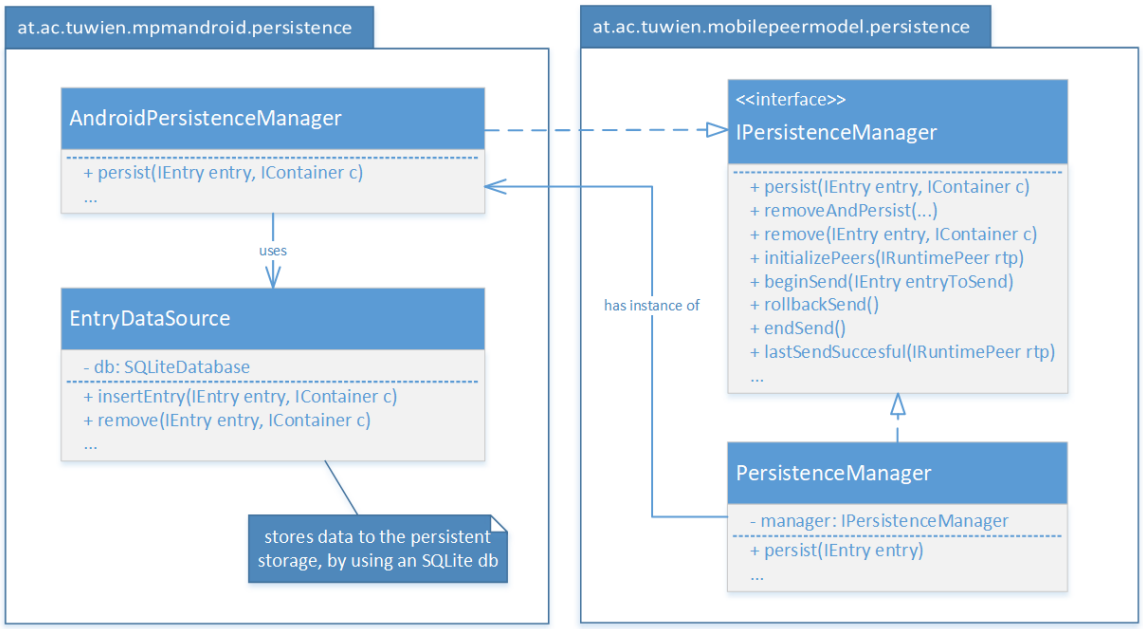

Figure 6.13: On the right side the *IPersistenceManager* interface and the *PersistenceManager* singleton is shown. The left side illustrates the concrete *AndroidPersistenceManager* of the MPM-Android library that makes use of a SQLite database to store Entries to the hard disk.

- **persist(IEntry entry, IContainer c):** If new Entries are inserted into a local Peer, for example if an Entry is received from a remote host or a new Entry is injected via the method *injectData()*, the Entry is also inserted in the persistence layer.

- **removeAndPersist(ContainerUpdate toRemove, List<ContainerUpdate> toPersist):** Figure 5.11 in the design section illustrates the sequence diagram of a Wiring execution with the call to the persistence layer enabled. The method *removeAndPersist()* removes Entries from the persistence that are taken from the PIC by a Guard and inserts all Entries into the persistent storage that shall be written to containers by the Wiring's Action. To guarantee data consistency in the persistence at any point in time, this method has to be executed in a single transaction, meaning that either all container modifications are reflected in the persistence or none. One *ContainerUpdate* object contains a list of Entries and the affected container. The first argument (*toRemove*) holds all Entries taken by the Guard and the second argument (*toPersist*) holds all Entries that are written to (possibly different) containers.

- **beginSend(IEntry entry), rollbackSend(), endSend()**: Those methods are called on the *PersistenceManager* during the send process in the Sender-Service.

Figure 5.12 in the design section illustrates the sequence diagram of the send process including calls to the persistence layer. Also in the design chapter (see Section 5.6.7), all possible exceptional situations that might occur while an Entry is being sent are elaborated and presented.

- **getNotSuccessfullySentEntry()**: If a send transaction could not be successfully committed, a potential send exception should be detected. To achieve this, each Entry is marked in the persistence layer while the Entry is transmitted to the server. If any exception occurs during this procedure, the Entry is still marked on the next RTP start. Therefore, the method *getNotSuccessfullySentEntry()* will return the Entry that might not have been delivered successfully to the server. The RTP then wraps the returned Entry into an Exception-Entry and forwards it to the Exception-Peer, where further actions can be defined via a dedicated Service and Action (see Section 6.3.6).

- **initializePeers(IRuntimePeer runtimePeer):** If the persistence layer is enabled and the RTP is restarted, the state of all containers is restored during the startup. The method *initializePeers()* should therefore iterate over all local and system Peers and write the persisted Entries of the permanent memory to the defined containers of the respective Peer.

The PersistenceManager singleton

Figure 6.13 shows the relationship between the *IPersistenceManager* interface, the singleton *PersistenceManager* and the concrete class *AndroidPersistenceManager*. Here, the *proxy* design pattern is used, where one object (the proxy) holds another object with the same interface [Gra03]. Method calls on the proxy are then forwarded to the concrete object, if certain conditions are fulfilled. In this case, calls are only forwarded from the *PersistenceManager* proxy to the concrete *AndroidPersistenceManager* if persistence is enabled. In the core framework there is no concrete persistence implementation available and therefore the singleton proxy *PersistenceManager* just ignores all method calls.

When using the MPM-Core, an application developer can easily implement a concrete *PersistenceManager* for standard Java applications that stores Entries on the file system of any operating system, like Windows or Linux. The concrete implementation can either be added via a call to *PersistenceManager.setInstance(IPersistenceManager newInstance)* or by specifying the fully qualified class name in the file *general.properties* (i.e. *persistenceImplementation=at.myapp.persistence.ConcretePersistenceManager*). This file must be located under *src\main\resources*. Additionally, the persistence layer can be enabled or disabled in this file (*persistenceEnabled=true/false*).

Android persistence implementation

The AndroidPersistenceManager is provided via the delivered MPM-Android library and is registered for the RTP in the *onCreate()* method of the Runtime-Service. The

reference implementation of the Android persistence layer makes use of an SQLite database. The Android platform provides classes that facilitate the work and management with local databases[34]. The class *EntryDataSource* (see Figure 6.13) acts as Data Access Object (DAO) and provides some convenient methods to insert and remove Entries in/from the local database. Its private member variable *db* of type *android.database.sqlite.SQLiteDatabase* also supports transaction handling. When removing and inserting Entries at the same time, the methods *db.beginTransaction()*, *db.setTransactionSuccessful()* and *db.endTransaction()* ensure that either all container updates are reflected on the disk or none (atomic commit)[35]. Furthermore, SQLite can be safely used in applications with multiple threads[36]. This feature is necessary, as different Peers may use the database concurrently. However, only one thread can write to the database at the same time, which can drastically decrease the overall system performance.

The current database model contains only a single table with 4 columns that is used to store all Entries. The columns are *hash_code*, *container_id*, *entry* and *sending*. The first column contains the hash code of the Entry's object and is used to identify an object during the deletion. The hash code value is obtained via the method *System.identityHashCode(Object o)*[37], which will always return the value that the default method *hashcode()* of the root object *Object* would return. This method is used, because an application developer might override the Entry object and introduce an own hash code function to the subclass, which would possibly corrupt the persistence implementation. Obviously, the *container_id* represent the unique id of the container (PIC) and also contains an information about its Peer.

The *entry* column is used to store the Entry, serialized as String (or byte array). For serialiazing/deserializing, two predefined implementations exist (see Section 5.4.2) that can be activated by setting the concrete class name in the *general.properties* configuration file. More details about the serialization components can be found in Jörg's thesis [Sch17a]. By default, the *Gson* serialization is used, because for that approach no additional *TypeAdapter* has to be developed, which are responsible for transforming a generated *protobuf* message object to a Java first-class object and vice versa. However, application developers can also implement an own serializer for the persistence layer, the class only has to be compliant with the interface *IEntrySerializer* of the serialization layer.

The last column is used to mark an entry as *currently sending*. The value in all rows with a container_id different to the one of the Sender-Peer will always have 0 as value. An Entry that is sent to a remote host at the very moment will have the value 1 assigned, until the Entry is completely deleted after successful transmission. On each RTP start it

[34] `https://developer.android.com/training/basics/data-storage/databases.html` accessed: 2017-05-25

[35] `http://www.sqlite.org/atomiccommit.html#_introduction` accessed: 2017-05-25

[36] `http://www.sqlite.org/threadsafe.html` accessed: 2017-05-25

[37] `https://docs.oracle.com/javase/7/docs/api/java/lang/System.html` accessed: 2017-05-26

is checked if there exists a row with the value 1, and if one exists it is wrapped into an Exception-Entry and forwarded to the Exception-Peer (see 6.3.6). Also at each RTP start, all Entries in the table are fetched and inserted into the specific container to initialize the runtime with its latest state. At the same time, the hash_code column is updated with the hash code value of the newly created (deserialized) Entry.

One could argue that using one single table for storing the Entries is inefficient, because if several threads of different Peers want to write to the database, they have to wait for the previous transaction to finish, as the table is locked. However, SQLite is no database management system comparable with server or desktop solutions, like i.e. *MySQL*[38] or *PostgreSQL*[39], and only allows one thread to write data to the database at once. No concurrency control mechanism and optimization strategies are implemented[40], as this is beyond the scope of the SQLite project and would also mean too much resource consumption. Nevertheless, there exist a lot of features, as it can be seen on the official website[41].

Additional overhead

Reflecting all container updates immediately in the persistence also means a lot of additional overhead. Therefore, the persistence can be disabled by setting the property *persistenceEnabled* in the file *general.properties* to *false*. An application developer should be aware of that limitation and it is recommended that in systems with a high data flow the persistence layer is only activated if it is really necessary. Furthermore, the detection of failures during the send process can be disabled explicitly by the configuration variable *sendTransactionEnabled*. Apart from that, an application developer might enhance the existing implementation or exchange it with an optimized one. As discussed earlier, the concrete persistence implementation can be set directly via the *PersistenceManager* class or by defining the fully qualified name of the Java class in the *general.properties* file.

In Chapter 8 several tests have been performed to measure the additional overhead the persistence layer is generating with the provided solution. Although there is a considerable overhead, it can be shown that the solution with the SQLite database is not much slower as another promising, but more complicated approach, which just appends the serialized Entry to a file (see Section 8.2.3 for more details).

6.7 Tests

First of all, the correct functionality of the core framework is intensively tested. Especially for important components (Peer, Wiring, Guard and Action) a lot of test cases exist. Unit tests for small independent parts as well as integration tests for the interactions

[38] `https://www.mysql.com` accessed: 2017-05-27
[39] `https://www.postgresql.org/` accessed: 2017-05-27
[40] `https://sqlite.org/lockingv3.html` accessed: 2017-05-27
[41] `https://www.sqlite.org/features.html` accessed: 2017-05-27

between several components can be performed and are available in the test package of the project. The integration tests also demonstrate different use cases and can also serve as additional documentation.

The container implementation is exhaustively tested, with around 60 test cases, including high concurrency scenarios with around 1000 threads that are writing and taking entries at the same time.

Also the Notifier-Peer has some test cases, in the first place, to prove the correct behaviour of the two provided MPM-Services (*NotifyUserService* and *UpdateUserTokenService*) via unit tests and in the second place, to completely test the functionality of the Runtime-Peer with all its Peers and Wirings via integration tests.

Finally, the MPM-Android library project contains some test cases in its test package. Unit tests for the persistence layer and integration tests for different example Runtime-Services have been implemented. They can be executed on a Universal Serial Bus (USB)-connected physical device or on an emulator, meaning that Android framework API's and information (as the target app's *Context* object) can be accessed[42]. With the help of *Firebase Test Lab*[43] the tests can also be run simultaneously in a cloud-based infrastructure on several different devices to cope with the wide variety of device configurations. It is even possible to test an application in the *Test Lab*, if not a single test case has been implemented. In this so-called intelligent *robo test*, the system will try to crash the app by automatically interacting with the user interface.

[42] `https://developer.android.com/training/testing/unit-testing/instrumented-unit-tests.html` accessed: 2017-05-28

[43] `https://firebase.google.com/docs/test-lab/` accessed: 2017-05-28

CHAPTER 7

Proof of concept application

In order to illustrate a possible use case for an Android P2P application and to demonstrate the functionality and usability of the created software solution, a sample application built upon the MPM framework is presented in this chapter. All necessary steps that have to be carried out by a developer are described in detail. Therefore, this chapter also serves as a manual for new application developers.

7.1 A secure P2P messenger app with the MPM framework

As proof of concept application, a simple Android messenger app is implemented that makes use of FCM push notifications to notify the receiver about a new incoming message. Additionally, full end-to-end encryption is supported, meaning that not even the server can read the payload of the delivered messages.

In this section, the most important setup steps for the project and fundamental code snippets are presented. The full sources of the sample application will be provided in the publicly available source code repository along with the sources of the MPM-Core, the MPM-Android library, the Notifier-Peer and the web application for registering new users.

7.1.1 Infrastructure setup

Figure 5.1 of the design chapter illustrates all participating entities in an MPM application. Important components, apart from user-added hosts, are the relay server, the Notifier-Peer and the registration web application.

Server setup

In the reference implementation XMPP is used as protocol to communicate with the server. Therefore, any server that supports this protocol can be used. *Openfire* is a well known XMPP server, written in Java, that can be easily and quickly configured[1]. In a nutshell, the server has to be installed on any operating system of choice and a short web-based installation wizard has to be completed. Here the server port, the service name (see Section 2.5.3) and the user database get configured. For experimental tests also an in-memory database can be chosen.

Notification setup

The Notifier *Spring Boot* project has to be checked-out together with the MPM-Core to enable the provided notification approach. The only necessary part of the project for an application developer is the folder *src\main\resources*, where all settings can be configured. First the *Notifier* user has to be created in the XMPP user database. The name can be arbitrary, however, the MPM-Core has configured the name of that system host as *notifier_peer* by default. It can be changed by setting a key-value pair in the *general.properties* file in the MPM project later (i.e. *notifier_hostname=myNotifierName*). The password of the *Notifier* user has to be configured in the file *application.properties* in the Notifier project. Also the database that is used to store the FCM tokens has to be configured there (URL, username, password). The data model of the database can be automatically created by the Spring framework and again there is the possibility to start an in-memory database for testing purposes. As next step, an FCM project has to be created on the official Firebase website[2] and the thereby generated API key must be configured in the file *firebase.properties*. Finally, the *Spring Boot* application can be deployed and the first MPM host should be able to successfully start the Runtime-Peer.

7.1.2 Android project setup

To write Android applications an Android project has to be created in the first place. This can be either done with a wizard in any supporting IDE or by copying a sample application from the online repository. If the latter is used, the next step (adding the MPM-Android dependency) is not needed.

Adding MPM-Android dependency

There are two possibilities how to use the MPM-Android library for an Android project. Firstly, to download the sources of the library and mark it as dependency in the module window of the IDE. Or secondly, and the far better and easier solution, to add the dependency of the library in the app module's *build.gradle* file (see Listing 7.1).

[1] `http://download.igniterealtime.org/openfire/docs/latest/documentation/install-guide.html` accessed: 2017-05-26

[2] `https://firebase.google.com/docs/android/setup` accessed: 2017-05-26

```
dependencies {
   compile 'tuwien.ac.at:mpmandroid:2.0.1'
   ...
}
```

Listing 7.1: The MPM-Android library can be easily used in an Android project by defining a dependency in the gradle build file.

With this single line, the build tool *gradle* will download the library with the specified version from a repository (like *maven* or *jcenter*) and compile the library together with the Android source code.

Configuring necessary properties

Next, the XMPP settings, defined in the previous step, have to be configured in the *xmpp.properties* file under *src\main\resources*. This file might look like Listing 7.2:

```
host=ec2-35-164-176-148.us-west-2.compute.amazonaws.com
port=5222
service=mpmmessenger
security=enabled
```

Listing 7.2: A sample XMPP property file.

Further settings can be configured in the *general.properties* file, i.e.:

```
persistenceEnabled=true
sendNotifications=true
...
```

Additionally, also the *Connection* class of the communication layer and the serialization implementation can be defined in this property file. If those properties are not specified, the default implementations are used. More details about these features can be found in the thesis of Jörg Schoba [Sch17a]. For this proof of concept application, the default implementations are absolutely sufficient. The persistence layer is only needed in the sample messenger application to detect errors during the send process (see Section 7.1.3). Otherwise, this feature would not be necessary, because messages received from other hosts are stored in a local SQLite database. However, they could also be stored in a local Peer and fetched via dedicated Wirings, but this would mean to permanently hold all Entries in the RAM, which is not beneficial. Notifications are enabled by default, here it is just listed for demonstration purpose.

In the case that there shall also be a non-Android host in the system (i.e. another server host like the *Notifier*) or a host running on a desktop computer, the MPM-Core library can be used for a standard Java project instead.

7.1.3 Extending the MPMRuntimeService

The delivered *MPMRuntimeService* that is used as a wrapper around the actual Runtime-Peer is an abstract class. Therefore, a concrete subclass has to be derived from the *MPMRuntimeService* implementing all abstract methods. Section 6.6.2 describes the methods of the *MPMRuntimeService* in detail. In the following, some sample code of the MPM messenger Runtime-Service is shown.

Initializing the type registry

In order to properly serialize and deserialize the data payload of Entries, each coordination-type has to be mapped to a data-type, which is actually the Java class of the payload. In the MPM messenger app there is only one user-defined class necessary for now. The class *Message* simply holds the actual message as *String* and a unique ID as *long*. The ID is needed to identify the message and to send back a confirmation about the successful arrival of the message. As the payload of this confirmation Entry only contains the message ID, there is no need to register it as own type. System types like *String*, *int*, *long* or *boolean* are automatically added to the type registry.

The actual serialization layer implementation shall be configurable by requirement NFR 5 (see Section 3.1.2) and at least one human-readable and one binary implementation, aiming at performance, shall be provided. The code in Listing 7.3 is sufficient for the default JSON-serializer (see Section 5.4.2) that serializes the payload into a human-readable format. If the second provided serialization implementation is used, which converts the data into a non-readable and binary format, the developer has to implement particular type adapters to convert the Java objects into *protobuf*[3] messages. Jörg describes the need for these adapters in more detail in his thesis [Sch17a].

```
@Override
protected void initializeTypeRegistry() {
    DataTypeRegistry.putMapping("message", Message.class);
}
```

Listing 7.3: Code to map the coordination-type *message* to the Java class *Message.class* in order to enable the correct serialization/deserialization of Entries with a payload of that type.

Here the coordination-type *message* is mapped with the user-defined Java class *Message.class*. Every time an Entry with the coordination-type *message* has to be serialized, the concrete serializer obtains the mapped Java class from the *DataTypeRegistry*. This is the case when Entries are sent to a remote host or, if the persistence layer is enabled, also during each Wiring execution.

[3] https://developers.google.com/protocol-buffers/ accessed: 2017-05-27

Initializing local Peers

All coordination-related code must be inserted in this method. More specifically, local Peers and their Wirings shall be created and added to the Runtime-Peer here.

```
@Override
protected void initializePeers() {
  //create Peer
   IPeer peer = runtimePeer.createPeer("messenger_peer");

   //create Wiring to receive messages
   IGuard grd = new Guard();
   grd.addLink(LinkOperation.TAKE, EntryCount.exactly(1), "message");
   IService service = new MessageReceivedService(this);
   IAction action = new Action();
   action.addExternalLink(EntryCount.exactly(1), "message_ack");
   peer.addWiring(new Wiring(grd, service, action));

   //create Wiring to receive message confirmations
   IGuard guardAck = new Guard();
   guardAck.addLink(LinkOperation.TAKE, EntryCount.exactly(1),
       "message_ack");
   IService serviceAck = new MessageAckReceivedService(this);
   peer.addWiring(new Wiring(guardAck, serviceAck, null));
}
```

Listing 7.4: Code that creates a Peer with name *messenger_peer* and two Wirings.

The code in Listing 7.4 should be straightforward and clear. A Peer with the name *messenger_peer* is created. Then, two Wirings (with Guard, Service and Action) are created and added to the Peer. The first Wiring takes one Entry of type *message* from the PIC of the Peer and executes the Service *MessageReceivedService* (see Listing 7.5). After the Service completed, the Action is executed, which will try to send an Entry with coordination-type *message_ack* to a remote host. The second Wiring is responsible for fetching and handling this acknowledgement Entry.

The *MessageReceivedService* of the first Wiring is a bit more complicated (see Listing 7.5). First, the message payload is extracted from the received Entry with type *message*. Then, the message is sent out via broadcast to any component that is registered for the Intent-Action *message_received_action*. The message payload is actually delivered via an Intent-Extra with identifier *message_received_extra* and can be extracted from the Intent in the receiving BroadcastReceiver (see Section 7.1.5). The method sendBroadcast() on the LocalBroadcastManager returns *true* if at least one component could successfully receive the data. In that case, the app is currently shown to the user and the message was delivered to the GUI component. Apart from that, if the method returns *false*, an Android notification is created that shall make the user aware of an incoming message, as known from any messenger app. The full source code for the creation of a notification

can be seen in the source of the sample application or on the Android developer website[4].

```
...

@Override
public void execute(IEntryCollection ec, IEntryFactory entryFactory)
    throws Exception {

  //take the Entry from the EC and get the message payload
  IEntry receivedEntry = ec.take("message");
  Message message = (Message) receivedEntry.getData();

  //put the received message into an Intent and send it via
     broadcast to any registered receiver.
  Intent localIntent = new Intent("message_received_action");
  localIntent.putExtra("message_received_extra", received);
  boolean success = LocalBroadcastManager.getInstance(context)
    .sendBroadcast(localIntent);

   //if no component received the message -> a notification will be
      shown in the Android notification bar.
  if(!success) {
    NotificationCompat.Builder mBuilder =
      new NotificationCompat.Builder(context)
        .setContentTitle("New message")
        .setContentText("You received a new message");

    //define what should happen, if the user clicks on the
       notification Entry (i.e. just open the app).
    ...
  }

  //add Ack-Entry that is sent back to the sender
  IEntry ackEntry = entryFactory.create("message_ack");
  ackEntry.setData(message.getId());
  ackEntry.setDest(new
     PeerURI(receivedEntry.getFrom().getHostName(),
     "messenger_peer"));
  ec.write(ackEntry);

  //here the received message could also be stored to a local
     database for later retrieval.
  ...
}
```

Listing 7.5: Parts of the code of the MessageReceivedService.

[4] https://developer.android.com/training/notify-user/build-notification.html accessed: 2017-05-27

As next step, an Entry with coordination-type *message_ack* is created and added to the EC in order to inform the sending host about the successful arrival of the message. The ID of the message is set as payload and the destination for the Entry is the originating Peer. Finally, all receiving messages might be inserted into a local database so that they can be fetched later on.

The Service *MessageAckReceivedService* is similar to the Service shown above, but much simpler. The message ID is extracted from the Entry and a broadcast is sent out to inform currently shown components that the message with the given ID has been received by the opposite host. Then the column *ack_received* is updated in the local database.

The code in method *initializePeers()* might be completely generated by the modeling tool, currently developed by Matthias Schwayer [Sch17b], whereby the application logic in the MPM-Services has to be implemented always by the developer.

Initializing the Exception-Peer

The developer can decide which action should be taken if any exception occurs during the runtime execution. This might be the expiration of Entries or an unsuccessful transaction during the send process. In the MPM messenger sample app no Entries with a positive *TTL* property are defined, meaning that TTL exceptions can be ignored. However, for the case of a potentially not transmitted Entry (exception-type *POTENTIAL_SEND_EXCEPTION*), the functionality that the Entry is resent on the next Runtime-Peer start is added to the Exception-Peer. Therefore, the code of Listing 7.6 is added in the method *initializeExceptionPeer()* of the Runtime-Service, which registers the MPM-Service *ResendExceptionService* (see Listing 7.7) and an Action at the Exception-Peer.

```
@Override
protected void initializeExceptionPeer() {
  IService service = new ResendExceptionService();
  IAction action = new Action();
  action.addExternalLink(EntryCount.largerEquals(1), "message");
  runtimePeer.getExceptionPeer()
    .setExceptionServiceAndAction(service, action);
}
```

Listing 7.6: Registration of a Service and Action for the Exception-Peer in the method *initializeExceptionPeer() of the MPMRuntimeService.*

The Service of the Exception-Peer is very simple (see Listing 7.7). It takes the Entry from the EC and parses it to an Exception-Entry (a subtype of Entry). Then it is assured that the exception-type of the Exception-Entry is *POTENTIAL_SEND_EXCEPTION*. Actually this verification would not be necessary, because the second currently supported exception with type *TTL_EXCEPTION* can never occur in this application. Finally, the Entry's data payload is written to the EC which is actually the potentially not sent

Entry with type *message*. In this scenario, the developer has to consider though that an Entry might be sent twice. The transaction surrounding the send procedure of the communication layer can only detect that the send process could not be completed, but not whether the Entry could be actually delivered or not.

```
@Override
public void execute(IEntryCollection ec, IEntryFactory entryFactory)
    throws Exception {

  ExceptionEntry entry = (ExceptionEntry) ec.take("exception");
  if(entry.getExceptionType() ==
      ExceptionEntryType.POTENTIAL_SEND_EXCEPTION) {
    ec.write((IEntry) entry.getData());
  }
}
```

Listing 7.7: The MPM-Service of the Exception-Peer, which simply resends a potentially not delivered Entry.

Inserting initial Entries

The last method that can be overridden by application developers is *insertInitialEntries()*. All Entries that shall be inserted at the very first time the application is started can be defined here. This feature is not needed in the sample application and thus, nothing is added in this method's body.

Registering the RuntimeService

As last step, the concrete Service *MyRuntimeService* has to be registered in the *AndroidManifest.xml* file, in order to let the Android system instantiate it at runtime (see Listing 7.8). The attribute *exported=false* ensures that the Service can not be invoked by components of other applications.

```
...
<service
   android:name=".service.MyRuntimeService"
   android:exported="false" />
...
```

Listing 7.8: Registration of the concrete Runtime-Service in the AndroidManifest.xml configuration file.

7.1.4 Registering the FCM-Services

To use the FCM notification approach, the two provided Services of the MPM-Android library have to be added to the app's *AndroidManifest.xml* file (see Listing 7.9). Further-

more, the *google-services.json* file that was generated when creating the FCM Project on the Firebase website (see Section 7.1.1) must be inserted into the Android projects *app* folder.

```
...
<service
   android:name="at.ac.tuwien.mpmandroid
      .service.MPMFirebaseInstanceIDService"
   android:exported="false">
   <intent-filter>
      <action android:name="com.google.firebase.INSTANCE_ID_EVENT"/>
   </intent-filter>
</service>
<service
   android:name="at.ac.tuwien.mpmandroid
      .service.MPMFirebaseMessagingService"
   android:exported="false">
   <intent-filter>
      <action android:name="com.google.firebase.MESSAGING_EVENT"/>
   </intent-filter>
</service>
...
```

Listing 7.9: Registration of the two FCM-Services in the AndroidManifest.xml file.

To recapitulate, the *MPMFirebaseInstanceIDService* is responsible for the update process of newly generated or rotated FCM tokens and the *MPMFirebaseMessagingService* will receive notification messages via its *onMessageReceived(RemoteMessage remoteMessage)* method. Then, simply the Runtime-Service is started in order to receive the pending message from the server.

7.1.5 Building the GUI

The purpose of this section is not to illustrate how a graphical user interface for Android applications can be built, but rather how the communication between GUI components and the Runtime-Service or the MPM-Services looks like. Otherwise, the sample application in the online repository or the Android developer training[5] are very good starting points for how to build the graphical user interface.

Receiving broadcasts from an MPM-Service

The *MessageReceivedService* (see Listing 7.5) is sending out a broadcast message if an incoming message has been successfully transported via a dedicated Wiring from the PIC of the Peer *messenger_peer* to the MPM-Service. In order to receive a broadcast

[5] https://developer.android.com/training/basics/firstapp/building-ui.html accessed: 2017-05-28

message in a GUI component, a BroadcastReceiver has to be registered via the method *Context.registerReceiver(BroadcastReceiver receiver, IntentFilter filter)*. The best place to do this is the *onResume()* method that is called whenever the GUI component (in the sample application a Fragment[6] is used) comes to the foreground. The app will stay in this state until the user decides to leave this component or if any system event occurs so that the focus is taken away from the app. Such a system event may be an incoming phone call or the notification about a low battery state. More details on the Activity (and Fragment) lifecycle can be found in Section 5.6.1 and on the Android developer website[7].

```
@Override
public void onResume() {
   super.onResume();
   LocalBroadcastManager.getInstance(getActivity())
      .registerReceiver(chatReceiver, chatIntentFilter);
}
```

Listing 7.10: Registration of a BroadcastReceiver in the lifecycle method *onResume()*.

The variable *chatReceiver* of Listing 7.10 is a subtype of BroadcastReceiver and implements the method *onReceive()*, which is invoked by the system if a broadcast has been sent out. In the GUI component, the message is added to a list of messages which is managed by a *ListAdapter*[8] (see Listing 7.11).

```
private class ChatReceiver extends BroadcastReceiver {

@Override
public void onReceive(Context context, Intent intent) {
  Message m = (Message) intent
     .getSerializableExtra("message_received_extra");

  //update the list of messages so that the new message is shown
  chatAdapter.add(m);
  chatAdapter.notifyDataSetChanged();
}
```

Listing 7.11: The implementation of the ChatReceiver.

The variable *chatIntentFilter* of Listing 7.10 is of type *IntentFilter* and defines which broadcast messages (Intents) shall be received by a BroadcastReceiver. In the example above only Intents with action *message_received_action* are filtered.

[6] https://developer.android.com/guide/components/fragments.html accessed: 2017-05-28

[7] https://developer.android.com/guide/components/activities/activity-lifecycle.html accessed: 2017-05-28

[8] https://developer.android.com/reference/android/widget/ListAdapter.html accessed: 2017-05-28

As last step, it is very important to unregister the *ChatReceiver* when the app is closed or the focus is taken away from it (see Listing 7.12). The right place to do this is either the *onPause()* or the *onStop()* method. If the receiver is not registered anymore, the method *sendBroadcast(localIntent)* in the MessageReceivedService will return *false* and therefore an Android notification is shown to inform the user about the new message.

```
@Override
public void onPause() {
   super.onPause();
   LocalBroadcastManager.getInstance(getActivity())
      .unregisterReceiver(chatReceiver);
}
```

Listing 7.12: The ChatReceiver is unregistered in the lifecycle method *onPause()*.

Communication with the Runtime-Service

The previous subsection dealt with the communication from an MPM-Service to Android GUI components. Here, the communication from GUI components to the Runtime-Service is shown. In order to allow clients to communicate with an Android Service, the *onBind()* method has to be overridden (see Section 6.6.2). A good place to actually bind to a Service is the *onStart()* callback function (see Listing 7.13). However, this depends on the actual use case, for example sometimes it would be better to bind to the Service after a specific button was clicked.

```
@Override
public void onStart() {
  super.onStart();

  runtimeServiceConnection = new MPMRuntimeServiceConnection();
  Intent intent = new Intent(getActivity(), MyRuntimeService.class);
  getActivity().bindService(intent, runtimeServiceConnection,
     Context.BIND_AUTO_CREATE);
}
```

Listing 7.13: To communicate with the Runtime-Service a *ServiceConnection* has to be established.

The *MPMRuntimeServiceConnection* is responsible for managing the connection between the Service and a component. The *Context.BIND_AUTO_CREATE* indicates that the Service should be created if it is not currently running. The callback function *onServiceConnected(ComponentName className, IBinder service)* is executed by the system if the connection could be established successfully (see Listing 7.14).

```
private class MPMRuntimeServiceConnection implements
    ServiceConnection {

  @Override
  public void onServiceConnected(ComponentName className, IBinder
      service) {

    MPMRuntimeService.RuntimeServiceBinder binder =
        (MPMRuntimeService.RuntimeServiceBinder) service;
    runtimeService = binder.getService();
    serviceBound = true;
  }

  @Override
  public void onServiceDisconnected(ComponentName className) {
    serviceBound = false;
  }
}
```

Listing 7.14: The method *onServiceConnected(...)* is called automatically by the operating system after a successful service connection.

After onServiceConnected() has been invoked, the Runtime-Service can be used to inject new data into the Runtime-Peer. Such a call looks like the following in the MPM messenger example app: *runtimeService.injectData("message", message, new PeerURI(receiver, "messenger_peer"));*

Finally, the connection to a GUI component has to be closed at some point (unbind). A good place to do this is generally the *onStop()* method. If the Service has been created with a call to *bindService()*, the Service will be stopped automatically if the last component has unbound from it.

7.1.6 Activate end-to-end encryption

One important non-functional requirement of the MPM framework is *NFR 3 (Security)*. Besides the optional channel encryption that can be used to prevent the unauthorized interception and eavesdropping of packages by third parties, this requirement demands an optional end-to-end encryption in order to ensure that not even the server can interpret the forwarded data. This requirement was in Jörg's area of responsibility. Therefore, reference is made at this point to Jörg's thesis [Sch17a], which describes the design and implementation of this part of the software.

To use end-to-end encryption, an application developer only has to write the line *e2e_encryption_enabled=true* in the *encryption.properties* file. As it is the case for all resource files, it has to be located in the app-module's folder *src\main\resources*.

7.2 A coordination focussed Android app with the MPM framework

The MPM messenger application described in the previous section does not utilize the whole potential of the provided framework, as the coordination mechanism is almost only used to map incoming messages to a specific MPM-Service. Jörg has developed an Android application that makes use of the master-slave pattern. The master and each slave are represented by an MPM host in the network and their interactions and collaborations are modeled via constructs of the Mobile Peer Model (Peers and Wirings). A sophisticated problem shall be solved by dividing it into smaller sub-tasks so that each slave only has to process a small portion of the problem and the heavy computational workload can be distributed. In the sample application the non-polynomial Travelling Salesman Problem (TSP) shall be solved by using a Genetic Algorithm (GA)-based optimization technique [Sch17a].

CHAPTER 8

Evaluation and critical reflection

This chapter demonstrates on the one hand the benefits of the built software solution and on the other hand provides a critical reflection of the framework, revealing some important considerations for framework and application developers.

Firstly, it is pointed out which additional effort is needed by an application developer to setup the infrastructure and to implement specific use cases of an Android application comparable to the P2P Android messenger application presented in Chapter 7 without the provided MPM framework.

Secondly, the Android persistence layer is evaluated in respect of additional overhead and scalability. Moreover, a second more sophisticated persistence implementation is analyzed and compared with the current one.

Finally, the chapter closes with a critical reflection on the built software solution, including some disadvantages and limits of the framework.

8.1 Benefits of the framework

The MPM framework simplifies and accelerates the development of Android applications that make use of peer-to-peer communication and collaboration tasks to a great extent. The time and implementation effort that can be saved by using the framework is demonstrated in this section.

8.1.1 Server setup

Without the MPM framework, the developer has the choice to select an arbitrary server implementation and protocol. This seems as a huge advantage at the first glance, but as existing background technologies for P2P networks in mobile environments (see Chapters 2 and 3) have already been elaborated thoroughly, the usage of an XMPP relay server is

definitely one of the best existing options. So, unnecessary long research activities can be skipped in good conscience.

8.1.2 Communication layer

After a communication protocol and a concrete server implementation have been chosen, a client library must be found to enable a reliable communication with the server. Then, a communication layer has to be designed and implemented, which can be quite time-consuming, especially if good software engineering patterns shall be applied. Quite important is also that the serialized messages can be interpreted by different operating systems in order to support all current mobile platforms. Jörg describes in his closing words that the effort for the platform-independent serialization implementation was relatively high.

If the MPM framework is used, the only thing an application developer has to bother about is the deployment of an XMPP server (i.e. Openfire[1]) and the registration of important configuration values in the file *xmpp.properties* under *src\main\resources* (i.e. server URL, port, etc.).

8.1.3 Error handling

Mobile environments also stand for restricted availability and therefore error handling is a very important topic with regard to sending data. If the message cannot be transmitted to the server because of a missing internet connection, the message must be cached and after a successful re-connection (in Android the system broadcasts an event with action *android.net.conn.CONNECTIVITY_CHANGE*) the message can be resent. In the case that the message could not be sent and the application (already in the background) is killed by the operating system because of a memory shortage or if the battery is empty, the cached message also have to be persisted so that it does not get lost.

The MPM framework handles all those error cases out of the box and will eventually deliver the message to the server, even if the application was killed or the device was completely restarted (when the optional persistence layer is enabled). Furthermore, the MPM framework detects any irregularities during the send procedure of the communication layer and will forward this information to the Exception-Peer on the next Runtime-Peer start so that the message can be resent, if desired (see Section 7.1.3).

8.1.4 Push notifications

In Section 5.6.5 the need for push notifications in mobile environments is explained in order to stay connected with other hosts while saving processing power. The main advantage of this approach is that no constant connection has to be held with the server.

[1] `https://www.igniterealtime.org/projects/openfire/` accessed: 2017-06-09

Without the MPM framework developers first have to thoroughly research the different messaging capabilities of Google's Firebase Cloud Messaging (FCM) or a similar notification techniques. In the case that FCM has been chosen, first an FCM project has to be created on the official FCM website. Then, a web application has to be created that stores all unique FCM tokens of end devices and which communicates with Google servers in order to notify a concrete device. This application certainly needs a database or something similar to map users to their devices, which has to be designed and constructed. In the Android project the Firebase client library has to be included in the project and at least two Android Services (described in Section 6.6.3) must be implemented to get the notification approach running.

To use this feature in the MPM framework, only the FCM project must be created online, the Notifier-Peer (see Section 6.4) project has to be deployed on an application server and the two FCM-Services provided via the MPM-Android library must be registered in the AndroidManifest.xml file. No further configuration settings have to be performed, even the data model of the token database for the Notifier-Peer application can be automatically generated via the Spring Data JPA plugin.

8.1.5 End-to-end encryption

By using the optional channel encryption of the XMPP protocol only eavesdropping of data packages by third parties can be prevented. In this scenario the XMPP relay server could still intercept data packages, because only the connection between client and server and vice versa are encrypted. Jörg has worked also on an effective and exchangeable end-to-end encryption implementation [Sch17a] (see also Figure 5.2 in Section 5.4). The OTR protocol was chosen after comparing several possible end-to-end encryption candidates. As described in Section 7.1.6, when using the MPM framework a single line in the file *encryption.properties* is sufficient to enable a secure end-to-end encryption between all participating peers in the network.

8.1.6 Further security countermeasures

In the course of Jörg's thesis, different known security vulnerabilities in internet-scale applications and entworks are described [Sch17a]. The MPM framework supports counter measures against eavesdropping, data modification, replay attacks, identity spoofing, MITM and DOS attacks, as well as spam. In order to implement these features in an application from scratch, obviously a lot of effort has to be made.

8.1.7 Coordinating the data flow

The sample application of Chapter 7 does not exploit the full potential of the provided coordination mechanism of the MPM framework. However, with a simple and intuitive API application designers and developers can easily control the data flow and the execution of specific business logic.

8.1.8 Summary

To recapitulate, in order to use the MPM framework for an Android project, only the library has to be added to the *build.gradle* file in the app-module, the provided Services of the library have to be registered in the AndroidManifest.xml and the abstract *MPMRuntimeService* class has to be extended to configure the coordination-related tasks. Of course, the XMPP server and the notification peer (if desired) have to be deployed and the configuration values have to be registered in the appropriate files.

Apart from that, with the proof of concept messenger application one can see that the MPM framework does not only facilitate the development of distributed collaboration tasks that need a sophisticated underlying coordination mechanism. Also applications that simply need a reliable and secure P2P communication can profit from the framework. If an Entry has been added to the Runtime, eventual transmission to its destination is guaranteed.

8.2 Runtime-Peer and persistence performance

The Android operating system may kill a running background thread without any prior notice, even an Android background Service can be terminated by the system if available resources are low or the Service has been running for too long in the background (see Section 5.6.1). Therefore, a persistence layer can be optionally activated at compile time to reflect the current state of data containers on a permanent memory. In the end of Section 6.6.5, it was already mentioned that a noticeable additional overhead has to be expected, when the persistence layer is activated.

Several test cases have been implemented to demonstrate the additional computation time that is required if persistence is enabled. As only the AndroidPersistenceManager (see Section 6.6.5) has been developed at the moment, concrete evaluation results are only available for the Android platform. For a desktop and server environment, a more sophisticated solution might be appropriate that makes us of a database server.

8.2.1 Infrastructure

Three devices have been used to measure the additional calculation time of the persistence layer, two emulators and one real device:

- Device 1: emulated device, quad core CPU 64 bit, 1536MB RAM, Android 7.1.1
- Device 2: emulated device, duo core CPU 64 bit, 1024MB RAM, Android 5.1
- Device 3: real device, quad core CPU 64 bit, 2048MB RAM, Android 6.0.1

The emulated devices have been hosted on a Windows 10 desktop computer with an Intel Core i7-6700K 4x4.40 GHz processor and 16GB DDR4-RAM. These specifications are

mentioned, because they probably have an impact on the performance of the emulated Android devices, as they perform much faster than a real smartphone (see the results in Section 8.2.4).

8.2.2 Test cases

No real use-cases are needed to measure the additional overhead of the persistence layer. Therefore, two simple Runtime-Service implementations are used in order to achieve this objective.

- EchoService 1: Holds a Runtime-Peer with one Peer and a single Wiring that takes one Entry of type "message", executes a Service (which appends a short String to the Entry's String payload) and performs an Action that sends the Entry back to its sender.
- EchoService 2: From the outside, this Runtime-Service acts identically as the EchoService 1 by just echoing a received message with a small additional information back to its sender. However, this Runtime-Peer contains three Peers, where each of them has one Wiring that forwards the Entry to the next Peer. The last Peer then sends the Entry back to the sender. The flow of an Entry inside this Runtime-Peer can be illustrated more clearly by listing the Peers that have to be passed: *Receiver-Peer –> Peer1 –> Peer2 –> Peer3 –> Sender-Peer.*

The second Runtime-Peer constellation is used to analyze the performance of the persistence when concurrently accessed. To reach such a situation (when three Peers want to persist Entries at the same time) a lot of Entries have to be available in the runtime. Therefore, also the time between each sent Entry can be configured:

- Single mode: An Entry is sent to the Runtime-Peer, gets processed and is returned to the origin. Then the next Entry is sent.
- Flooding mode: All Entries are sent simultaneously to the Runtime-Peer.

Altogether, always 100 Entries are sent so that a meaningful average execution time can be calculated. For the first test, the method *removeAndPersist()* (see Section 6.6.5) of the PersistenceManager is measured, which is called during a Wiring execution and removes the Entry from the current Peer's PIC and persists the new Entry into the Sender-Peer's PIC (or the PIC of Peer2 or Peer3 in EchoService2). Furthermore, the payload size of each Entry can be configured (0 KB, 10 KB, 100 KB or 1000 KB).

All tests have been performed with the provided *Gson*-Serializer that does not need any additional type adapter implementations for the Entry's payload (see thesis of Jörg [Sch17a]). By using the more performant protobuf implementation, the additional overhead of the persistence layer could be reduced, but this was not tested in the course of this evaluation.

Finally, also the overall execution time of each Runtime-Service (EchoService 1 and EchoService 2) for 100 incoming Entries is measured with persistence enabled and disabled.

8.2.3 Alternative version

As already mentioned in the implementation part of the persistence layer (see Section 6.6.5), the SQLite database used in the AndroidPersistenceManager only allows one single writer at a time[2]. Therefore, and to compare the provided solution with another one, a second experimental approach has been tested in the course of this evaluation chapter.

In the alternative implementation, all container updates are just appended as information to a file on the persistent storage. For example the deletion of an Entry with hash code *1234* from container with id *cont6* would just append the following line to the file: *remove;cont6;1234*. Similarly, if an Entry with hash code *5678* shall be persisted to container *cont7* for example the following line would be appended to the file: *persist:5678,cont7,<serialized entry>*. With this approach no database transaction handling and locking have to be performed and the data could also be written concurrently, if several files are used.

However, this approach is more complicated. In fact, the developer has to implement an own transaction handling to ensure that all container updates have been actually persisted to the file or none. Furthermore, the file has to be processed in order to stay in a consistent and manageable state. This is because the file is growing over time and might waste too much memory, which is usually extremely scarce anyway on mobile devices. This processing step should take place in the background at a point in time when the mobile device is not doing any intense computation tasks to not unnecessarily slow down the whole system. Moreover, this approach would lead to longer Runtime-Peer startup phases, as the whole file content has to be processed first in order to get the current valid Entries of all containers.

This approach has been tested in an experimental implementation, where the *removeAndPersist()* method is just appending the current container update to a file instead of using the SQLite database. This method is used during a Wiring execution and the execution times are measured in the same way as described above (100 times in single mode and 100 times in flooding mode). The comparison of the two approaches can be found in the next section. Note that only the method *removeAndPersist()* has been implemented and the described issues (i.e. transaction handling and continuous maintenance tasks) have not been realized.

[2] `https://sqlite.org/lockingv3.html` accessed: 2017-06-21

8.2.4 Results

Overhead of one Wiring execution

The following two Figures 8.1 and 8.2 illustrate the additional overhead of one Wiring execution when persistence is enabled. All three devices proportionally show the same results. Surprisingly, the emulated device 2, which has a slightly weaker hardware configuration and an older version of the Android operating system than the emulated device 1, is the fastest tested device. However, the difference between the two emulated devices is very low and therefore only the diagram of the slightly faster device 2 is shown.

Although both of the emulated devices have less RAM and a similar Central Processing Unit (CPU), they perform much better than the real device. The reason for that might be the quite performant execution environment of the emulators, which acts as underlying resource provider (see Section 8.2.1).

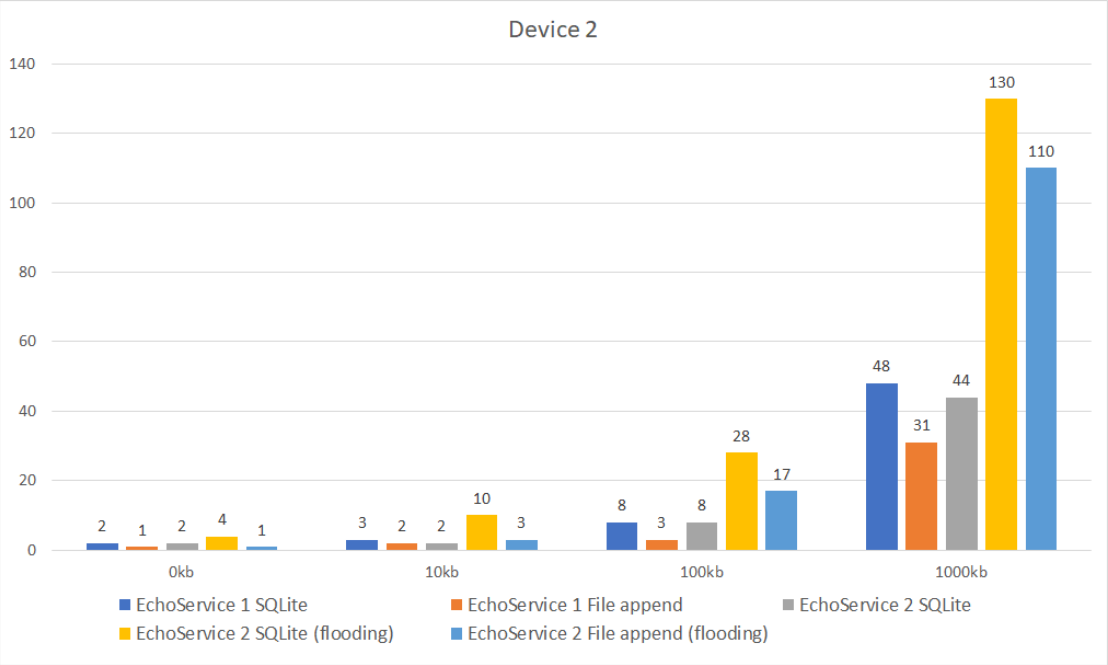

Figure 8.1: The additional overhead of one Wiring execution in milliseconds when persistence is enabled, measured on device 2.

As expected, one can see that the provided approach with the SQLite database is performing not that good when concurrent write operations are performed - see the yellow bar (EchoService 2 in flooding mode). This is especially the case if smaller payload data is used (0 KB to 10 KB), with payloads greater than 100 KB the two tested approaches (SQLite and file append) converge, most likely because the storage medium reaches its limit. The alternative approach clearly performs better than the implemented

version with the SQLite database. However, as already described in the section above, the file appending approach comes with other problems, for example a maintenance task that has to be performed in the background when the system is not fully utilized at the moment or the implementation of an own transaction handling. In addition, by the design of the MPM, not that many concurrent Peers are expected to write data concurrently, as one Peer can only execute one Wiring at a time. If, however, a lot of different Peers need to perform tasks at the same time and persistence is really needed, the developer can implement another approach by defining an own PersistenceManager.

The real device is performing very badly with large data payload, the test cases took longer than 200 ms with a data payload of 1000 KB. In the flooding mode (yellow and light blue), the Runtime-Peer even threw a CommunicationException, as the XMPP library could not manage the 100 MB of data in such a short period of time.

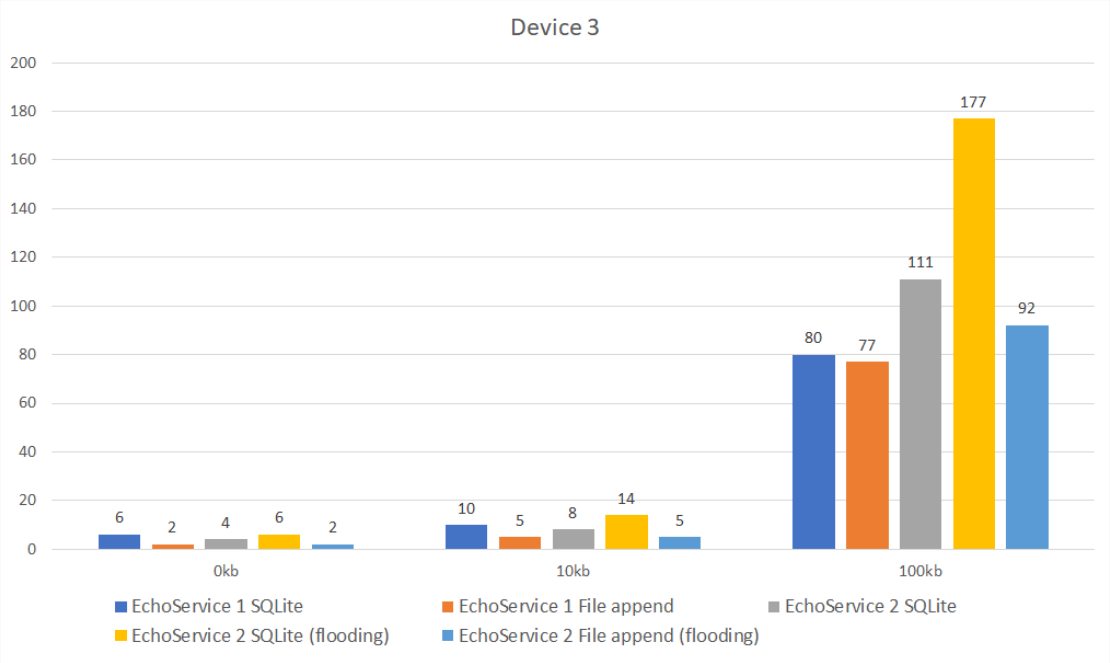

Figure 8.2: The execution time of the persistence layer of a single Wiring on a real device (Samsung Galaxy A5 2016). Entries with payload of 1000 KB exceeded the 200 milliseconds limit, the precise execution time is therefore not shown to provide a clear visualization of the other results.

Total overhead

Figure 8.3 shows the different execution times (in seconds) of the two tested Runtime-Peers when 100 Entries are sent in *flooding mode* to the emulated Android device 1. The tests have been performed 10 times and the average execution time is depicted in the figure. Again, different payload data sizes were used.

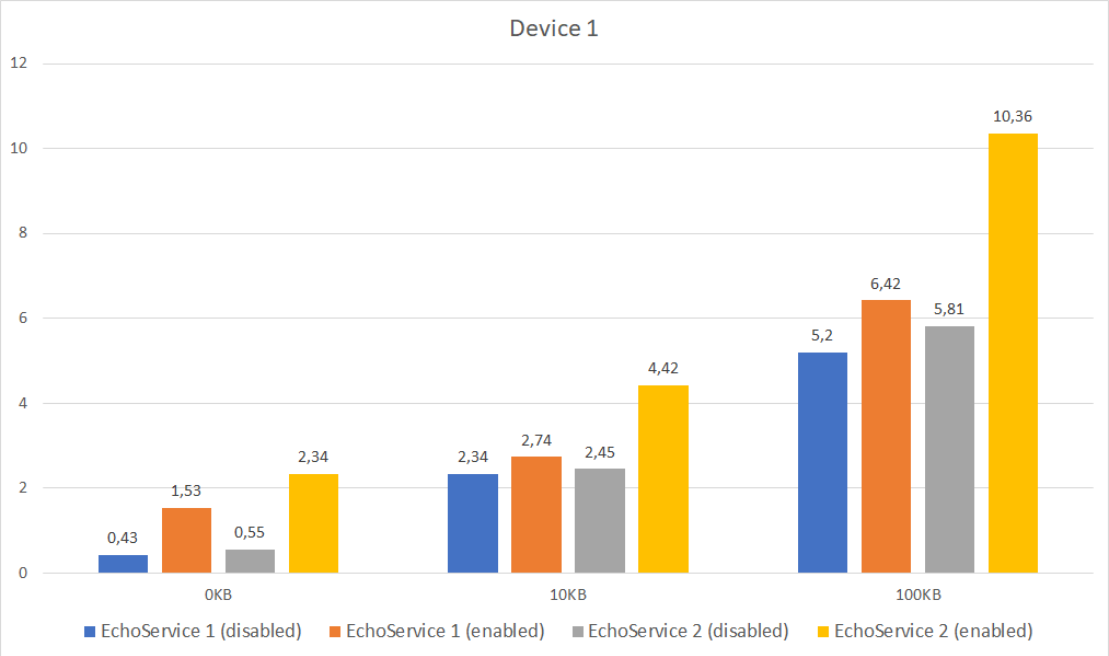

Figure 8.3: The overall execution time of the two tested Runtime-Peers when 100 Entries are sent with and without persistence, measured on device 1.

One can see that the elapsed time between EchoService 1 (one Peer) and EchoService 2 (three Peers) is almost the same if persistence is disabled. This is, because during a Wiring execution only the object reference of the payload data has to be added to the newly created Entry. However, if persistence is enabled, after each Wiring execution the whole Entry has to be persisted in order to correctly reflect the state of the Peer's container. As it could be seen in the previous two diagrams, the test with EchoService 2 is performing significantly worse than the one with EchoService 1.

With bigger data payload, the execution time of EchoService 1 (one Peer) with and without persistence enabled is almost the same, because the transmission and the deserialization of the incoming Entries are approximately as expensive as the persistence overhead. Especially EchoService 1 with payload data of 10 KB is only marginally slower, if persistence is enabled.

Again, the communication layer threw a *CommunicationException* when the test was performed with 1000 KB of payload data. Therefore, this data size is not shown in Figure 8.3.

Overall throughput

As last performance benchmark, the amount of Entries that can be processed in one minute by a Runtime-Peer on a real device (Device 3) is measured. A RTP with only

one Peer and one Wiring (EchoService 1) could receive, process and send back 40593 Entries without data payload in 60 seconds, if persistence is disabled. The test has been performed five times and the average number of processed Entries was used.

If the persistence layer is enabled, the RTP crashes after receiving around 700 Entries, because too many threads that are spawned in the communication layer (for each received Entry a new thread is created) try to write the serialized Entry into the database. The SQLite database can not write data concurrently and is not able to handle so many threads that have to wait for inserting the data. Therefore, a timeout of 12 milliseconds has been added between sending Entries to the device. In this constellation the RTP could process 4320 Entries in 60 seconds after all.

Summary

As the overhead for the persistence layer on real devices can become very high, especially if Entries with huge payload data are used or several Peers are working concurrently, the persistence should be only activated if really necessary. In addition to the higher execution time, of course also the battery will be drained considerably faster.

The test cases can be found in the test package of the MPM-Android library project. However, the exact test results might vary on a different execution environment.

8.3 Open issues

The provided MPM framework has been carefully tested with automated unit as well as integration tests (see Section 6.7) and two proof of concept Android applications have been developed to prove the correctness and the usability of the solution. This section, on the other hand, tries to reveal some limits of the framework.

The MPM RTP is running in the background and might consume a considerable amount of resources. In the first place, this is depending on the coordination-related code of the concrete application and its MPM-Services. Therefore, application developers have to take this into account when designing and developing their solutions for a mobile device in order to not negatively influence the overall system performance.

The current Android Runtime-Service gets automatically restarted by the operating system if it was killed due to a resource bottleneck or when it was running for too long in the background (see the return value *START_STICKY* in Section 6.6.2). Unfortunately this feature, the automatic relaunch of a Service after it was unexpectedly killed by the OS, can be used only in Android versions prior to 8. According to a statement on the official Android developer website[3], the new Android version (Android O - expected release date in August 2017) imposes some limitations regarding the execution of background services. Many running background services with resource-intensive tasks might decrease the user

[3] `https://developer.android.com/preview/features/background.html` accessed: 2017-05-07

experience drastically. Therefore, in the new version of the Android operating system, the Service is killed after a few minutes by an automatic call of the *Service.stopSelf()* method. With the chosen approach in the MPM framework, data consistency can still be ensured, but the application developer has to have the knowledge about these limitations for future Android versions.

The iOS system is also quite restrictive regarding the execution of background services and allows them to live only a relatively short period of time until they are killed by the OS[4].

In addition, the life span of an Android background Service is varying from device to device. The current state of the device including the amount of apps installed and the number of currently running apps and services, the hardware capabilities and the version of the operating system have direct influence on the point in time when the operating system may kill a background Service.

The communication between an MPM-Service and the GUI components of an Android app is not very intuitive in the first place. However, as GUI components might not have been initialized when the coordination-related Peers and Wirings are being created, the approach with the Android broadcasts is the most promising one.

Troubleshooting or debugging might also be quite difficult at first. The error might originate from a misconfigured Wiring or the incorrect implementation of an MPM-Service. Especially if a number of concurrent Wiring executions have to be debugged, error tracing can become fairly complex.

One feature that an Android application developer might desire is the immediate notification about the non-availability of the internet connection. Usually, developers are used to build a try-catch block around the code responsible for a simple internet request and expect an *IOException* on any communication failures. If an application developer tries to request an information with the constructs of the MPM, but the internet connection is not available, the injected request (Entry) is cached in the MPM as long as there is no connection (if no TTL property is specified). The developer does not know if the Entry was successfully sent or not, only the eventual delivery of the Entry is guaranteed by the design of the model.

One exceptional situation is currently not implemented, namely the automatic re-establishment of the XMPP connection if the server was temporary down. However, the cached Entries are resent if the network connection on the mobile device changes or if another Entry could be transmitted successfully.

A potential bottleneck of an MPM distributed system is the Notifier-Peer. On each incoming notification request, the database is queried to fetch the particular FCM token

[4] `https://developer.apple.com/library/content/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/BackgroundExecution.html` accessed: 2017-06-29

of the receiving peer. The tokens should be cached in order to increase performance and to prevent failure of the component.

8.4 Fulfilment of imposed requirements

In this section a short assessment of whether the imposed requirements of Section 3.1 are fulfilled by the provided framework is made.

The Peer Model [KCJ+13] has been chosen as underlying coordination model and by implementing the features of this model (including the RTP, Peers and Wirings) requirement FR 1 (Coordination) could be fulfilled. An Android library is provided that contains an Android Service that acts as a wrapper for the RTP so that it can be used to build applications that are executed solely in the background of a mobile device (FR 2). FCM is used to notify an application about an incoming message, which means that requirement FR 3 is met. By using the RTP for standard Java programs or the RTS for Android applications requirement FR 4 (decoupling) is fulfilled. FR 5 (Connectivity with local and mobile carrier networks) can be fulfilled by using an XMPP relay server, which enables a reliable communication in every network constellation, especially if a host is located behind a local W-LAN network that uses NAT.

The provided MPM framework and all additional software artifacts (see Section 6.2) will be published under a Copyleft license (NFR 1). Requirements NFR 2 and NFR 3 (scalability and security) have been implemented and evaluated by Jörg, see his thesis for more details [Sch17a]. The API of the framework has been kept as simple as possible and is intuitive (NFR 4), a reason for this is certainly the reduced feature set of the mobile version of the Peer Model. The provided code is well documented and furthermore several unit and integration tests exist that can be seen as an additional part of documentation (NFR 5). Important layers of the implementation can be exchanged easily by providing the class name of the concrete implementation in a configuration file (NFR 6). The coordination-related code is structured and can be generated by a modeler (NFR 7). As required by NFR 8, the framework is available for the Android platform and two proof-of-concept applications have been developed. In Section 8.1 of this chapter, the benefits of the framework have been discussed in detail (NFR 9). Resource efficiency (NFR 10) is a very important requirement for mobile devices in general, but is even more important for services that can run in the background. This requirement could be fulfilled on the first hand by elaborating a reduced mobile version of the Peer Model so that a resource-saving implementation is possible (i.e. no concurrent execution of Wirings or no transactions on the container level) and on the second hand by the usage of a notification approach in order to let an application switch into an idle state, but still be able to get notified about incoming messages. The last requirement (reliability, NFR 11) could be met by designing and implementing a persistence layer that is reflecting the state of the RTP at any point in time on a persistent storage.

CHAPTER 9

Conclusion

This final chapter shortly recapitulates the benefits and the achievement of the resulting MPM framework. Finally, some possible ideas on future work are given.

9.1 Summary

The presented MPM framework facilitates rapid development of P2P applications, especially in mobile environments. With the help of the integrated coordination model the data flow and the execution of business logic can be controlled easily in a distributed system. The infrastructure setup as well as the configuration and deployment of important system components can be completed very quickly.

A detailed analysis about coordination and communication frameworks as well as protocols has revealed the need for a relay server in a mobile environment. In the MPM reference implementation XMPP as protocol and Openfire as relay server have been chosen. To fulfil further requirements regarding mobile limitations, a reduced version of the selected underlying coordination model (the Peer Model) has been elaborated.

In the course of this thesis, the core framework has been designed and implemented, which consists of several components. Firstly, a core module is provided that can be used for standard Java applications, deployable on any OS that runs a Java VM with version 1.7 or higher. Secondly, an Android AAR module is delivered, which comprises an Android Service that can be used out-of-the-box for Android applications to guarantee the reliable execution of the RTP in the background. Furthermore, a notification approach has been implemented to notify a RTP about incoming data. Also the server part for the implemented push notification approach and a web application for registering new hosts in the system are provided. Because a mobile operating system might terminate a background service without any prior notice, a persistence layer has been introduced, which is responsible for reflecting the state of all containers of a RTP to

a persistent memory. For now, the framework is only available for the Android platform, but cross-platform considerations have been taken into account from the beginning of the software development lifecycle. As proof-of-concept a P2P messenger application has been implemented with the MPM framework.

The benefits of the provided framework have been pointed out in the evaluation chapter, according to which a software developer can save a lot of research and development effort. A performance evaluation showed the relatively high throughput of the RTP in a mobile test application. Furthermore, the persistence implementation has been analyzed in respect of additional execution time. It has been shown that persisting all Entries permanently to a persistent storage is notably slower on a mobile device, but that enabling the persistence layer is certainly applicable for applications with small data payloads.

9.2 Future work

As last part of this thesis, some possible extensions and improvements of the framework shall be provided.

First of all, the coordination profile of the MPM that was elaborated in the course of this work might be extended. In particular, one student that is already using the MPM framework for his master thesis requested the flow functionality (see Section 4.1.5) to simplify the development of a coordination-related problem. It is planned to implement this feature in the next version of the framework.

Secondly, the functionality of the container implementation (see Sections 5.2.2 and 6.3.1) might be enhanced to also allow different selection principles, like for example a *FIFO*, a *Last-In-First-Out (LIFO)* or a *Key-Selector*.

No persistence implementation for standard Java applications in a server or desktop environment is currently available and this might be an important future work for highly secure applications.

In addition, the level of battery consumption or CPU load for different use cases could be measured (with and without persistence enabled).

Some open issues were already described in Section 8.3. For example a caching functionality in the Notifier-Peer and automatic reconnections to the XMPP server after a temporary server shut down could be implemented.

One focus of Jörg's thesis was the analysis and elimination of security vulnerabilities in an MPM application in respect of the communication part of the system. In addition, security concepts on the application layer might be elaborated and integrated in the framework, i.e. to restrict the size of a container in order to circumvent overflow errors.

Finally, porting the framework to the iOS platform would be a conceivable future work. However, this would mean a lot of additional effort, as the MPM-Core and also the

wrapper around the RTP (assuming that the architecture of the framework is the same as the current one) have to be implemented in a new programming language.

As described in Section 8.3, some additional constraints regarding the execution of Android Services are being introduced with the new Android version O (planned release date in August 2017), see the official statement on the official Android website[1]. In order to ensure that the MPM framework is working properly also in future Android versions, the framework has to be maintained steadily and new features and restrictions have to be considered.

[1] `https://developer.android.com/preview/features/background.html` accessed: 2017-07-10

List of Figures

List of Listings

List of Tables

Acronyms

AAR Android Archive. 69, 89, 127

AMUSE Agent-based Multi-User Social Environment. 17

AOT Ahead-Of-Time. 68

API Application Programming Interface. 25, 34, 35, 41, 71, 82, 99, 117, 126

ART Android RunTime. 68

BSD Berkeley Software Distribution. 25

CAN Content Addressable Network. 6, 7

CPU Central Processing Unit. 121, 128

DAO Data Access Object. 97

DEX Dalvik EXecutable. 68

DHT Distributed Hash Table. 8, 9, 13–15, 27

DNS Domain Name System. 9, 18

DOS Denial Of Service. 25, 117

DSL Domain-Specific Language. 33, 78

DVM Dalvik Virtual Machine. 68

EC Entry Collection. 36, 40, 42, 49–52, 54, 71, 75–77, 79, 107

ejabberd Erlang Jabber Dameon. 22

FCM Firebase Cloud Messaging. 43, 45, 59, 60, 81, 82, 84, 86, 88–94, 101, 108, 117, 125, 126

FIFO First-In-First-Out. 34, 41, 71, 128

FIPA Foundation for Intelligent Physical Agents. 29

GA Genetic Algorithm. 113

GNU General Public License. 25

GUI Graphical User Interface. 61, 93, 105, 109–112, 125

HTTP hypertext transfer protocol. 20, 45

ID identifier. 8, 89

IDE Integrated Development Environment. 86, 102

IETF Internet Engineering Task Force. 17, 20

IM Instant Messaging. 22

IO Input/Output. 52

IP Internet Protocol. 7, 18, 46

IRC Internet Relay Chat. 20

JADE Java Agent DEvelopment framework. 15, 29, 30

JID Jabber ID. 18

JSF Jabber Software Foundation. 20

JSON JavaScript Object Notation. 67, 104

JVM Java Virtual Machine. 68

JXTA Juxtapose. 17, 29

LAN Local Area Network. 6

LEAP Lightweight and Extensible Agent Platform. 16

LGPL GNU Lesser General Public License. 17

LIFO Last-In-First-Out. 128

LRU Least-Recently-Used. 57, 58

MIT Massachusetts Institute of Technology. 8, 25

MITM Man In The Middle. 7, 117

MPM Mobile Peer Model. 6, 23, 25, 28, 31, 39, 41–43, 45, 47, 51, 57–62, 65, 67–72, 81, 84–96, 99, 101–104, 107–109, 111–113, 115–118, 122, 124–129, 133

NAT Network Address Translation. 14, 17, 25, 27–29, 126

OS Operating System. 83, 90, 124, 125, 127

OTR Off-the-Record Messaging. 21, 117

P2P Peer-To-Peer. 5, 6, 8–10, 12–15, 17, 24, 26–28, 30, 31, 46, 55, 87, 101, 115, 118, 127, 128, 136

P2PSIP Peer-to-Peer Session Initiation Protocol. 29

PGP Pretty Good Privacy. 21

PIC Peer-In-Container. 35, 36, 40, 41, 49–54, 61–63, 71, 72, 74–77, 79–81, 86, 95, 97, 105, 109, 119

PIER P2P Information Exchange Retrieval. 7

PM Peer Model. 31, 71, 78

POC Peer-Out-Container. 35, 41, 42

RAM Random-Access-Memory. 57, 58, 62, 103, 121

REST REpresentational State Transfer. 82

RPC Remote Procedure Calls. 14, 28

RTP Runtime-Peer. 35, 40, 43, 45, 47–49, 51, 52, 54, 55, 58–64, 69, 70, 74, 76, 77, 79, 80, 83, 85, 86, 88–91, 93, 94, 96–98, 123, 124, 126–129

RTS Runtime-Service. 59, 126

SASL Simple Authentication and Security Layer. 18, 21

SDK Software Development Kit. 68, 89

SIP Session Initiation Protocol. 22, 29, 30

SQL Structured Query Language. 34

TCP Transmission Control Protocol. 18, 46

TLS Transport Layer Security. 18, 21, 46

TSP Travelling Salesman Problem. 113

TTL time to live. 40, 53, 54, 72, 81, 107

TTS time-to-start. 72

UDP User Datagram Protocol. 14, 28

UI User Interface. 57, 88

URI Uniform Resource Identifier. 35, 41, 42, 76

USB Universal Serial Bus. 99

UTC Universal Time Coordinated. 40

VM Virtual Machine. 47, 68, 70, 127

VOIP Voice Over IP. 22

W-LAN Wireless Local Area Network. 11, 25, 29, 126

WADE Workflows and Agents DEvelopment framework. 17

WS-BPEL Web Service - Business Process Execution Language. 33

XAP eXtreme Application Platform. 71

XEP XMPP Extension Protocols. 20

XML Extensible Markup Language. 17

XMPP Extensible Messaging and Presence Protocol. 17, 19, 21, 29–31, 43, 45, 46, 51, 54, 55, 59, 70, 82, 89, 90, 102, 103, 115–118, 122, 125–128, 131

XSF XMPP Standard Foundation. 20

XVSM eXtensible Virtual Shared Memory. 41

Bibliography

[Alt16] Marion Altschach. Classification of space based computing systems. Master's thesis, TU Wien, 2016.

[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 Comput. Surv.*, 36(4):335–371, 2004.

[BCG14] Federico Bergenti, Giovanni Caire, and Danilo Gotta. Agents on the move: Jade for android devices. In *Procs. Workshop From Objects to Agents*, volume 2, 2014.

[BCP+03] Fabio Bellifemine, Giovanni Caire, Agostino Poggi, Giovanni Rimassa, and A Jade. Jade - a white paper. *Telecom Italia EXP magazine Vol*, 3, 2003.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Jade–a fipa-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London, 1999.

[CC10] C.-F. Michael Chan and S.-H. Gary Chan. Distributed hash tables: Design and applications. *Handbook of Peer-to-Peer Networking*, pages 257–280, 2010.

[CDG+02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314, 2002.

[CGK+05] Krzysztof Chmiel, Maciej Gawinecki, Pawel Kaczmarek, Michal Szymczak, and Marcin Paprzycki. Efficiency of jade agent platform. *Scientific Programming*, 13:159–172, 2005.

[CJK15] Stefan Craß, Gerson Joskowicz, and Eva Kühn. A decentralized access control model for dynamic collaboration of autonomous peers. In *Security*

and Privacy in Communication Networks – 11th International Conference (SecureComm), pages 519–537, 2015.

[CKS09] Stefan Craß, Eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *International Database Engineering and Applications Symposium (IDEAS)*, ACM International Conference Proceeding Series, pages 301–306. ACM, 2009.

[CMM02] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving dns using a peer-to-peer lookup service. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 155–165. Springer-Verlag, 2002.

[DGKW10] Krishna Dhara, Yang Guo, Mario Kolberg, and Xiaotao Wu. Overview of structured peer-to-peer overlay algorithms. *Handbook of Peer-to-Peer Networking*, pages 223–256, 2010.

[Die08] Tim Dierks. The transport layer security (tls) protocol version 1.2. `https://tools.ietf.org/html/rfc5246`, 2008. accessed: 2016-10-22.

[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.

[Gra03] Mark Grand. *Patterns in Java: a catalog of reusable design patterns illustrated with UML*. John Wiley & Sons, 2003.

[HA02] S. Hazel and Wiley B. Achord. A variant of the chord lookup service for use in censorship resistant peer-to. peer publishing systems. In: Proc. of the 1st Int'l Workshop on Peer-to-Peer Systems (IPTPS 2002). Cambridge, 2002.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 321–332. VLDB Endowment, 2003.

[JEA+07] Diane Jordan, John Evdemon, Alexandre Alves, Assaf Arkin, Sid Askary, Charlton Barreto, Ben Bloch, Francisco Curbera, Mark Ford, Yaron Goland, et al. Web services business process execution language version 2.0. *OASIS standard*, 11(120):5, 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, 2006.

[JYF07] Yuh-jzer Joung, Li-wei Yang, and Chien-tse Fang. Keyword search in dht-based peer-to-peer networks. *IEEE Journal on Selected Areas in Communications*, 25(1):46–61, 2007.

[KCH14] Eva Kühn, Stefan Craß, and Thomas Hamböck. Approaching coordination in distributed embedded applications with the peer model DSL. In *40th EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA)*, pages 64–68, 2014.

[KCJ98] Lars M. Kristensen, Soren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[KCJ+13] Eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In Rocco De Nicola and Christine Julien, editors, *15th International Conference on Coordination Models and Languages (COORDINATION), held as part of the 8th International Federated Conference on Distributed Computing Techniques (DisCoTec)*, volume 7890 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2013.

[KCS15] Eva Kühn, Stefan Craß, and Gerald Schermann. Extending a peer-based coordination model with composable design patterns. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 53–61, 2015.

[KK07] Byungryong Kim and Kichang Kim. Keyword search in dht-based peer-to-peer networks. In Hai Jin, Omer F. Rana, Yi Pan, and Viktor K. Prasanna, editors, *Algorithms and Architectures for Parallel Processing: 7th International Conference*, pages 338–347. Springer Berlin Heidelberg, 2007.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663. ACM, 1997.

[Küh17] Eva Kühn. Flexible transactional coordination in the peer model. 7th IPM International Conference on Fundamentals of Software Engineering (FSEN). Springer, 2017. (to appear).

[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, 2005.

[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, 2005.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65. Springer-Verlag, 2002.

[Nie06] Pin Nie. An open standard for instant messaging: extensible messaging and presence protocol (xmpp). In *TKK T-110.5190 Seminar on Internetworking*, pages 1–6, 2006.

[otr] Otr | off-the-record messaging. `https://otr.cypherpunks.ca/`. accessed: 2016-10-23.

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

[PRR99] G. C. Plaxton, R. Rajaraman, and W. A. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350. Springer-Verlag, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[RWBB05] T. Reidemeister, P. A. S. Ward, K. Bohm, and E. Buchmann. Malicious behaviour in content-addressable peer-to-peer networks. In *3rd Annual Communication Networks and Services Research Conference (CNSR'05)*, pages 319–326, 2005.

[SA05] P. Saint-Andre. Streaming xml with jabber/xmpp. *IEEE Internet Computing*, 9(5):82–89, 2005.

[SA11] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. `https://tools.ietf.org/html/rfc6120`, 2011. accessed: 2016-10-22.

[SASTT09] Peter Saint-Andre, Kevin Smith, Remko Tronçon, and Remko Troncon. *XMPP: the definitive guide*. O'Reilly Media, Inc., 2009.

[Sch17a] Jörg Schoba. Mobile Peer Model: A mobile peer-to-peer communication and coordination framework - with focus on scalability and security. Master's thesis, TU Wien, 2017. (in preparation).

[Sch17b] Matthias Schwayer. Towards a visual design and development environment for the peer model. Master's thesis, TU Wien, 2017. (in preparation).

[SM02] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 261–269. Springer-Verlag, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160. ACM, 2001.

[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.

[UTG08] M. Ughetti, T. Trucco, and D. Gotta. Development of agent-based, peer-to-peer mobile applications on android with jade. In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM '08. The Second International Conference on*, pages 287–294, 2008.

[UVG05] Y. Upadrashta, J. Vassileva, and W. Grassmann. Social networks in peer-to-peer systems. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, pages 200–209, 2005.

[XM12] Bai Xuefu and Yang Ming. Design and implementation of web instant message system based on xmpp. In *2012 IEEE International Conference on Computer Science and Automation Engineering*, pages 83–88, 2012.

[ZHS+04] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, 2001.

[ZO09] X. Zheng and V. Oleshchuk. Improving chord lookup protocol for p2psip-based communication systems. In *2009 International Conference on New Trends in Information and Service Science*, pages 1309–1314, 2009.