FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Enabling Scalable Collaboration by Introducing Platform-Independent Communication for the Peer Model

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Mag.iur. Stephan Cejka, BSc
Matrikelnummer 00925492

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 29. Jänner 2019

Stephan Cejka   Eva Kühn

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Enabling Scalable Collaboration by Introducing Platform-Independent Communication for the Peer Model

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Mag.iur. Stephan Cejka, BSc
Registration Number 00925492

to the Faculty of Informatics

at the TU Wien

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

Vienna, 29th January, 2019

Stephan Cejka Eva Kühn

Erklärung zur Verfassung der Arbeit

Mag.iur. Stephan Cejka, BSc
Ostmarkgasse 31/8, 1210 Wien

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

Wien, 29. Jänner 2019

Stephan Cejka

Acknowledgements

First, I want to thank Eva Kühn for the supervision of this thesis and for letting me work on interesting projects in her team. For the co-supervision, as well as for numerous helpful discussions on the implementation, I want to thank Stefan Craß. As the Java implementation was already used by other students for their theses during its finalization, I want to thank Stefan Zischka for his helpful comments and improvement suggestions.

I want to thank my parents Amanda and Theodor for making my studies possible, and my brothers Julian and Benjamin for accompanying me on my way.

Special thanks need to be addressed to Matthias for our collaborations during nearly our whole studies. Representing all my colleagues from Siemens, I want to thank Albin, also for bringing me into Siemens.

I want to thank Alexander Hanzlik, who was the first one, who read parts of this thesis. Furthermore, I want to thank Magdalena for the joint diploma thesis writing sessions. Finally, I want to thank Saskia for motivating me near the end of writing the thesis.

Without all of them, the thesis would never have been finished.

Kurzfassung

Trotz der enormen technischen Entwicklungen der Computerhardware und den dadurch ermöglichten Fähigkeiten in den letzten Jahrzehnten werden auch weiterhin viele Aufgaben nicht von einer Maschine alleine bewältigbar sein. Die Integration von Systemen zu einem verteilten System zur kooperativen Bewältigung einer Aufgabe ist jedoch durch die Heterogenität der Maschinen, beispielsweise bei Hardware oder Betriebssystem, eine komplexe Angelegenheit.

Diese Diplomarbeit stellt die Enterprise-Java-Implementierung des Peer Models vor, das von der Space Based Computing Group am Institut für Computersprachen an der Technischen Universität Wien erfunden und spezifiziert wurde. Das Peer Model ist ein datengetriebenes Modell für die Koordination von heterogenen Systemen, welches auf den Konzepten der Timed und Colored Petri-Netzen basiert. Eine der wichtigsten Anforderungen für die Kollaboration zwischen Maschinen ist die Kommunikation zwischen diesen. Die Diplomarbeit evaluiert daher einige Serialisierungs- und Kommunikationsformate, um plattformunabhängige Mechanismen für das Auffinden von Peer-Model-Instanzen im Netzwerk, sowie den Datenaustausch mit diesen zu ermöglichen. Dies erlaubt die Entwicklung von skalierbaren und verteilten Lösungen, wobei die involvierten Peer-Model-Instanzen nicht notwendigerweise in der selben Programmiersprache entwickelt sein müssen. Zusätzlich können mit dieser Implementierung während der Laufzeit der Peer-Model-Instanz dynamisch Komponenten hinzugefügt und entfernt werden. Die Implementierung wird in eine – im Aufbau befindliche – Toolchain integriert, die verschiedene Werkzeuge und Systeme im Peer-Model-Kontext bereitstellt. Mehrere Diplomarbeiten und eine Dissertation verwenden oder erweitern diese Implementierung bereits für deren Anwendungsfälle.

Abstract

Despite enormous technical developments on computer hardware and the resulting abilities in the last decades, many applications are still not and will never be computable by one machine alone. For collaboration, various machines are thus assembled to distributed systems; however, as these machines may be heterogeneous in hardware and operating systems, this integration shows to be a challenging task.

The contribution of this diploma thesis is an enterprise Java implementation of the Peer Model, a data-driven model for collaboration of heterogeneous systems based on the concepts of Timed and Colored Petri Nets, introduced and specified by the Space Based Computing Group of the Institute for Computer Languages, TU Wien. One of the primary requirements of collaboration between machines is to enable communication between them. To that end, the thesis evaluates several serialization and communication formats and defines a platform-independent mechanism of instance discovery and data exchange between Peer Model instances. Thus, it allows to build up scalable and distributed solutions while not requiring the Peer Model instances executed on the collaborating machines being implemented in the same programming language. Furthermore, the implementation allows to add and remove entities during the runtime of the Peer Model instance and thus enables dynamic adaptions while being executed. The implementation is integrated into a developing toolchain composed of various tools and systems in the Peer Model context. Furthermore, a few diploma theses and one PhD thesis are already using or extending this implementation for their use cases.

Contents

CHAPTER 1

Introduction

Technical revolutions in the last decades led to increasing processor capabilities and available space on one machine. Along with these comes an enormous decrease of costs of systems in comparison to their technical possibilities. In 2007, *Tanenbaum/Van Steen* showed a performance per price gain of machines of 10^{13} in the past half-century [57]. Technical evolutions have not stopped since, now focusing not on the further increase of one processor's capabilities but on the use of multicore systems. Furthermore, today mobile devices can handle tasks for which a few years ago a high-end desktop system was required. Their processor capabilities are not far behind desktop computers, some of them already including octa-core processors.

Despite these revolutions, many applications are not and will never be computable by just one machine. The collaboration of hundreds of systems may be required to accomplish a joint task; hence those distributed systems have gained high importance over the last decades. One requirement to foster the composition of various systems is to use high-speed communication between machines – both in local and in wide area networks. The collaboration of such heterogeneous systems, following different design decisions and using uncountable different implementations, is a challenging task. Thus, a middleware layer (Figure 1.1) between the local (operating) system and the application was introduced [57] that allows different applications on higher layers to communicate over a distributed system and hides heterogeneous lower layers that differ in hardware and/or operating system. Furthermore, it provides an abstraction layer for coordination logic.

1.1 The Peer Model

The Peer Model [38, 39] is a data-driven model based on the concepts of Timed [12] and Colored [51] Petri Nets. As being located on the middleware layer, it can be used for the collaboration of various heterogeneous systems. While components of the Peer

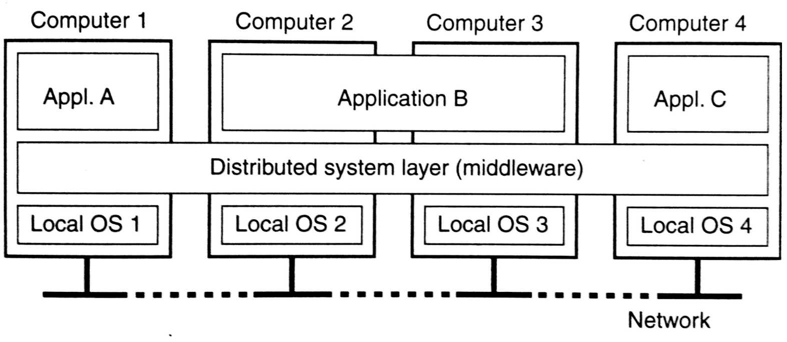


Figure 1.1: Middleware layer [57]

Model handle the coordination logic, the application logic is strictly separated and implemented by the developer itself using so-called *services*. They are written in the respective programming language of the implementation and executed by the Peer Model once the defined preconditions are met.

The Peer Model was invented and specified by the Space Based Computing Group of the Institute of Computer Languages at TU Wien. Several diploma theses that are already finished or are currently in progress specialize on one component of the system. These components will eventually form a toolchain pictured as a honeycomb (introduced in [29]) in Figure 1.2.

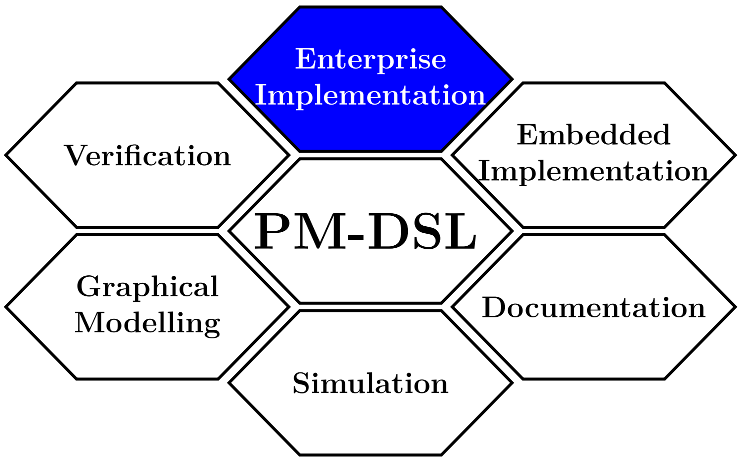


Figure 1.2: Overview of the Peer Model Toolchain [29]. Blue parts are covered in this thesis.

For understanding the concept of the Peer Model, some basic terms need to be introduced briefly: The **peer** is the central and eponymous component of the Peer Model. Peers are running on a Runtime Peer (RTP), in context of this work also called the **PM Instance**. Each peer provides **containers** that serve as a temporary store for **entries**. An entry encapsulates information and as such is transferred between peers. A peer consists of **wirings** that are executed if the specified pre-conditions are met and that move entries between peers and/or containers. Wirings contain **services** that encapsulate business logic to process entries or create new ones. A more detailed description of the Peer Model will be provided in Chapter 3.

1.2 Contributions and Tasks

Some implementations of the Peer Model with different focuses had already been introduced by other students in previous theses [29, 31, 52, 54, 58]. The Java implementation developed in hand with this thesis is an enterprise implementation of the specification (cf. Figure 1.2) using a multi-threaded runtime, allowing extendability by using a modular architecture and providing a user-friendly API. In contrast to the mentioned earlier implementations, the Java implementation for the first time allows the features described in the following subsections.

1.2.1 Dynamic adding/removing of entities during runtime

One goal is to enable adding and removing of resources without restricting functionality. Concerning the Peer Model, it shall be allowed to add and remove services, wirings, peers and even whole instances.

The Java implementation allows the dynamic adding and removal of those resources during its runtime and an unlimited number of containers in peers. It thus supports the proposed meta-model [20] as both wirings and (sub-)peers are treated as special entry types utilizing special containers.

1.2.2 Language-independent remoting

Distributed system components can only collaborate if they know each other; they need to communicate with each other by use of a network. Figure 1.1 shows that the systems required to communicate may be heterogeneous; they may differ in hardware and software components. In the Peer Model context, it shall thus be supported to communicate with remote Peer Model instances that may be implemented in another programming language.

The requirement introduces restrictions on the use of the serialization and message exchange formats, limiting those to platform-independent options. Therefore, this thesis first evaluates several possible serialization and communication formats. It specifies a method for discovery of other instances and the following communication between these instances, i.e., the exchange of *entries*. This communication protocol is used in the Java implementation such that collaboration with instances on other platforms – implementing the specification – is not restricted.

1.3 Methodology

In a first step, existing literature concerning the Peer Model including previous diploma theses have been evaluated. This included a literature research on models that are comparable to the Peer Model.

As the Peer Model specification has evolved during the time this thesis was written, it was required to decide which features are to be implemented and which ones will be part of future work. In that way, the functional and non-functional requirements have been defined next.

The primary focus of this thesis is the introduction of a platform-independent collaboration method between Peer Model implementations to allow building a distributed system and use cases requiring a horizontally scalable solution. Thus, communication options and serialization formats that seem to be suitable for this use case were then evaluated, and a decision on the communication method was made based on the results.

Next, the enterprise Java version of the Peer Model was implemented including the use of the decided platform-independent formats for the discovery of and communication with other instances.

A continuous evaluation of the implementation was possible due to various diploma theses and one PhD thesis already using this implementation for their use cases. Therefore, several helpful comments and feature requests could successfully influence this work during the implementation phase.

1.4 Expected Results

The expected outcome of this work is an enterprise implementation of the Peer Model in Java that allows the development of use cases requiring communication between various instances.It supports the features introduced in [38, 39], such as sub-peers, flows, and timing properties (TTS, TTL). Furthermore, this implementation is the first one that supports the Peer Model meta-model [20], i.e., all wirings and peers are particular kinds of entries. As such, dynamic changes of the model are possible as entries of these types come and go during the runtime.

It was defined that Peer Model instances running on different machines shall be able to collaborate on a common task. For such functionality, these instances are required to communicate with each other either over a local network or the Internet. The thesis does not deal with any networking issues; thus for simplification, all Peer Model instances are running in the same subnet.

As mentioned before, some diploma theses and one PhD thesis are using this implementation for their use cases, including the implementation of security concepts using the meta-model [20]. Therefore, it is expected that the implementation will be adjusted and extended based on future use cases' needs. It thus requires extensibility and maintainability by using modules that can easily be exchanged.

1.5 Outline

The remainder of the thesis roughly follows the structure that was given when discussing the methodology (Section 1.3). It is organized as follows:

In **Chapter 2**, related work is presented. The findings of previous diploma theses concerning the Peer Model are summarized first. Then the Peer Model is compared with a few frameworks using comparable concepts.

The introduction included only a brief description of the Peer Model. **Chapter 3** deals in-depth with the Peer Model concepts as specified in previous scientific contributions.

Afterward, **Chapter 4** introduces the functional and non-functional requirements.

In **Chapter 5**, design decisions for enabling remoting such as addressing and discovery are made. One of the primary goals of the thesis is to enable the communication between Peer Model instances that may be implemented in different programming languages. Thus, options for a suitable serialization format to enable language-independent collaboration are evaluated.

The Java implementation for enterprise use cases that was developed in hand with this thesis and follows the evaluated approaches is described in **Chapter 6**.

The implementation is then evaluated by comparing it with related work in **Chapter 7**. Furthermore, the fulfillment of the functional and non-functional requirements is described, including benchmarks for performance and scalability – generally and concerning communication.

Chapter 8 shows proposals for future work and concludes this thesis with a summary of the results.

CHAPTER 2

Related Work

As the central part of this thesis is the platform-independent communication between Peer Model instances, the evaluation with related work frameworks will be focused on coordination and remote communication.

First, Section 2.1 shows related work in the Peer Model area, i.e., it shows the contributions of previous theses and publications.

Four frameworks have been identified for comparison: The Actor Model was identified by previous diploma theses [52, 53] as being best comparable to the Peer Model approach. It is described in Section 2.2, including its Java implementation Akka. Sharing a high level of similiarities with it and also with the Peer Model, vert.x and Gridlink are introduced in Section 2.3. Next, WS-BPEL is described in Section 2.4 due to its wide spread in distributed coordination. Furthermore, Gigaspaces XAP is introduced in Section 2.5 as a representative of space based computing products due to their similarities with the Peer Model and its high number of features.

Finally, the chapter is concluded with Section 2.6, where the chosen frameworks are evaluated and the missing features to be handled in this thesis are identified.

2.1 Peer Model

The Peer Model is a novel approach for modeling concurrent activities and coordination developed and published by the Space Based Computing Group at the Institute of Computer Languages, TU Wien [38]. It provides a higher-level programming abstraction than Petri nets, on which the Peer Model is based on, inspired by asynchronous message queue or tuple space communication, event-driven architecture and data-driven workflow [38]. Further publications that extend the abilities of the original publication are [39] and [37].

Kühn et al. [38] compare the Peer Model with Colored Petri Nets (CPN) and Reo regarding the scalability of modeling. It shows that while quite powerful, high-level abstractions are missing in CPN and they do not scale. The Peer Model is neither domain-specific nor intended for a specific use case only. The initial leading use case for the team was the wireless propagation of events about approaching trains at railway crossings [39, 37]. A visual simulation of the use case showed a train moving and a visual representation of messages transmitted during the scenario [19]. This simulation was developed in Java, initially without using a Peer Model implementation, but stating its use as future work.

The implementation delivered with and described in this thesis is an entirely novel and general purpose implementation in Java, focusing on the communication between instances, including the ability to communicate with instances that may run on another platform. In the future, a toolchain, which can be imaged as a honeycomb picture (Figure 2.1), will be formed by the shown tools in the Peer Model environment. The figure extends the introductory figure in Section 1.1 by providing more details as well as references to other parts of the toolchain. Some of these tools are already finished by other diploma theses (colored in grey), some tools are currently in work (colored in green), and some are still in the planning phase (colored in white).

This thesis introduces the Java implementation, one kind of an enterprise Peer Model implementation. In the same category falls the C# implementation of the Peer Model (PeerSpace.NET) that was earlier finished by *Rauch* [52]. This implementation was the first general purpose implementation of the Peer Model; however, as the specification had not been finished at that point, the implementation most probably is not entirely inline with it. Thus, *Rauch's* implementation does not allow interoperability between different programming languages, and the afterward introduced dynamics in [20].

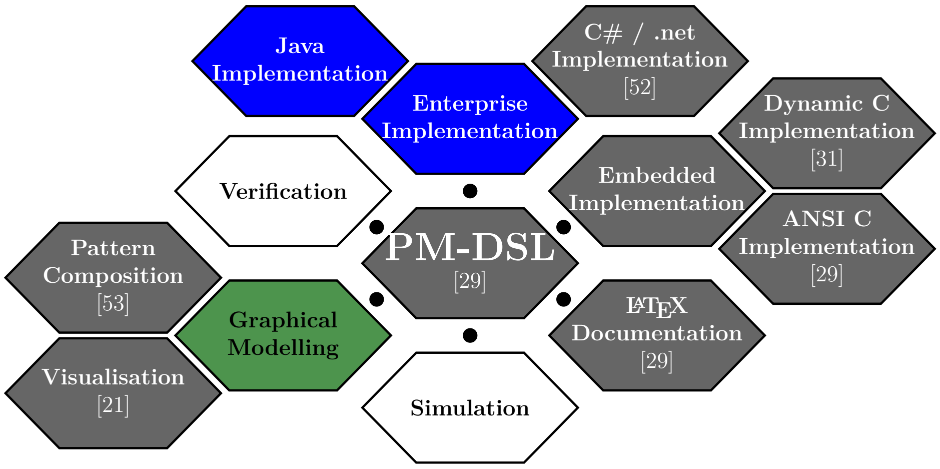


Figure 2.1: Extended Peer Model Toolchain [29]. Blue parts are covered in this work, green parts are work in progress, grey parts are already available from other works, and white ones are planned.

Various already finished and current diploma theses aim at implementations with different focuses. *Hamböck*'s thesis [29] defines the first toolchain components: (1) the domain-specific language to describe a Peer Model use case (PM-DSL), (2) an embedded implementation in ANSI C for heterogeneous wireless sensor network nodes as needed by the requirements of the motivating use case and (3) automatic LaTeX documentation generation. Naturally, the embedded version has some restrictions and assumptions that are the result of limits in these platforms. *Hamböck*'s proposed domain-specific language (PM-DSL) is primarily intended for his embedded implementation of the Peer Model; however, enhancements to enterprise versions were planned. Therefore the PM-DSL is designed to be extensible for features that are either not required or restricted in the embedded C version. The PM-DSL allows the automatic generation of the source for different embedded platforms by translation of the declarative model to source code [37] as well as the generation of the human-readable documentation in LaTeX. Those components are the foundations for an embedded development toolchain [37]; thus the PM-DSL holds the central point in the honeycomb picture on purpose. Another embedded implementation was done by *Holasek* in dynamic C [31].

Schermann [53] compared Reo, the Actor Model, Petri Nets, UPPAAL, WS-BPEL, BPMN and the Peer Model in his diploma thesis with a focus on flexibility, scalability, and the ability to reuse components. His thesis focused on the extension of the Peer Model with composable design patterns. A visual monitoring tool was created by *Csuk* [21] including a definition of a logging format to allow post-mortem monitoring of a Peer Model run. *Csuk* also compared the same mentioned modeling tools regarding an interactive, visual monitoring tool. *Schermann* [53] concluded that the Actor Model is the one modeling tool most comparable to the Peer Model. As there also exists a well-known Java implementation of this model (Akka), the thesis at hand will compare its communication features with the ones of the Peer Model. *Rauch* [52] focused on the evaluation of API usability and used the Xcoordination Application Space[1] for communication. Another current diploma thesis focuses on a graphical modeler for automatic code generation.

A specification for communication between Peer Model instances in the same or in different languages will be part of the thesis at hand; it will however not be possible to communicate with *Rauch*'s implementation without an extension that complies with the communication format. Communication to *Hamböck*'s implementation in C is expected to be not reasonable because of different use cases in mind. As the specification of the Peer Model has evolved (and is still evolving), a collaboration with these earlier implementations cannot be supported. A reduced implementation of the Peer Model specification for mobile devices was developed for the Android operating system by *Schoba* [54] and *Tillian* [58]. It aims for limited system ressources, especially for running in the background as Android service. They designed their implementation in a way that communication method can easily be exchanged as at the time of writing their theses, the preliminary decision for a format was already made, and a draft of the according chapter of this thesis was already available.

Latest contributions in the Peer Model area are the inclusion of a security concept using the techniques the Peer Model offers itself [20]. To that end, a meta-model is introduced,

[1]`http://www.xcoordination.org/application_space`

treating wirings and (sub-)peers as regular Peer Model entries. The handling of these entries is achieved by introducing new types of containers that allow a specific behavior for these entry types. The Java implementation was designed to support these entry types as well as the specific containers; *Craß* thus implemented the security concept in the context of his PhD thesis using this implementation. This Secure Peer Space implementation has already been extended by *Lettmayer* [43] for using a Public Key Infrastructure (PKI) in a Public Resource Computing use case. Furthermore, *Zischka* [60] already used the Java implementation for his use case of evaluating routing algorithms in unstructured peer-to-peer networks.

During the last years, the Peer Model specification has further evolved [35, 36, 40]. New concepts, such as variables and transactions, have been introduced that are out-of-scope of this thesis. Further work will be required to extend the Java implementation to address these concepts.

2.2 Actor Model

The Actor Model aims to provide a higher level of abstraction for concurrent and distributed systems to ease the writing of correct distributed, concurrent, fault-tolerant and scalable applications [44]. Thus, it avoids the necessity for the developer to deal with locking and thread management as far as possible.

Actors were first defined by *Hewitt et al.* in 1973 [30]. Subsequently, the Actor Model gained popularity with the growth of parallel and distributed computing platforms, such as multicore architectures and cloud computers [33]. Various actor languages and frameworks have been developed in the last decades, among them the Erlang language [44]. Famous examples nowadays using actor systems are the Facebook chat system, Twitter and WhatsApp [8], all having in common being systems coping with a high number of concurrent transactions and thus requiring a scalable system.

The definition from the *Encyclopedia of Parallel Computing* [33] describes the Actor Model by defining actors as

> "a model of concurrent computation for developing parallel, distributed, and mobile systems. Each actor is an autonomous object that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, and updating its own local state. An actor system consists of a collection of actors, some of whom may send messages to, or receive messages from, actors outside the system."

Programs are decomposed in smaller components called actors, which are autonomous, interactive, asynchronously operating components [33] having their own memory and communicating with other actors by exchanging messages [8]. Each actor contains its state and particular behavior. Messages are put into the receiver's mail queue, specified by its unique mail address. If the mailbox is empty, the actor blocks and waits for the next message to process [33].

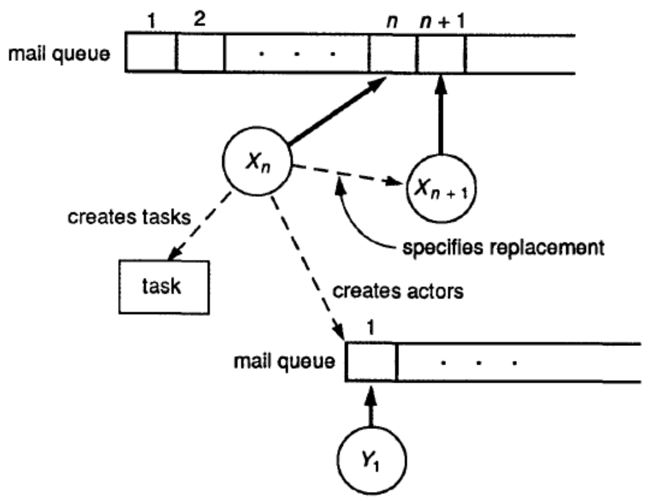


Figure 2.2: Actor behavior at message processing [6]

Figure 2.2 shows the fundamental behavior of an actor. At process step x_n the n^{th} message of the mail queue is handled. In reaction to the message, the process uses the following three primitives [9]:

1. `create`
   creates a new actor.
2. `send to`
   transmits a message to another actor. This message will eventually be delivered, and appended to the receiver's mail queue until processed. Since everything works asynchronously, the call of this primitive returns immediately, i.e., the actor is not blocked by waiting for a response.
3. `become`
   alters the state and/or behavior of the actor for step x_{n+1} that will process the next message $n + 1$.

A possible timeline of events is shown in Figure 2.3, where some (child) actors are created at certain points in time and communication between the actor and other actors is shown.

Location transparency is supported in the Actor Model. Actors can run on the same core, on the same CPU, or on another node in the network, while the actual location of the actor does not affect its naming [33]. The mentioned concepts are not tied to any programming language, they have however been directly integrated into some languages like Erlang and Scala. The next section will deal with Akka, a Java/Scala implementation of the

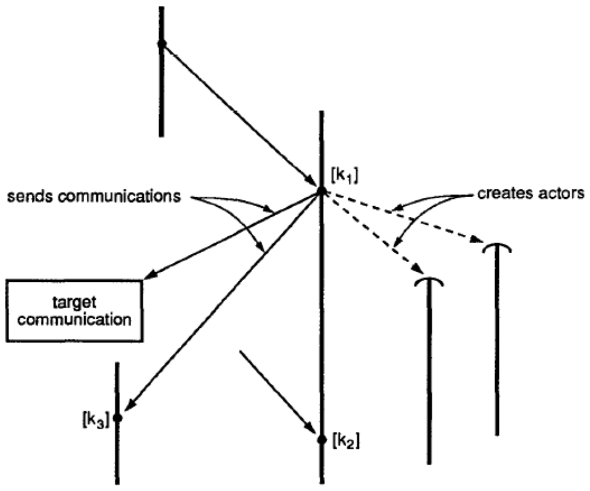


Figure 2.3: Actor timeline [7]

Actor Model's concepts. As just mentioned, Scala previously included actors directly in the language; however, with Scala 2.11.0, the Scala Actors library was deprecated [32] and replaced by Akka.

Akka

Akka[2] is an open source Java/Scala implementation of the Actor Model concepts using an asynchronous, non-blocking, highly performant message-driven programming model [44]. As the primary focus of this diploma thesis is the communication between peers on different Peer Model instances, this brief introduction to Akka will only describe functionalities within the Akka framework that are relevant for comparison. Extended functionalities, e.g., Akka TCP, Akka I/O, the event bus or streams were intentionally left out as not relevant for this purpose. Further details can be found in the detailed Akka documentation [44].

Each actor encapsulates a state and a defined behavior, defining the actions to be taken in reaction to a received message. Actors are hierarchically structured, such that each actor has one root actor and may create child actors that it will supervise. Supervision is most relevant whenever a failure in the child actor occurs. In such cases, the child actor suspends itself and all of its own child actors. It then notifies the parent actor which may either

1. resume the child actor, keeping its internal state.

[2]`http://akka.io`

2. restart the child actor, resetting its internal state.
3. stop the child actor permanently.
4. escalate the failure to its own supervising actor and hence, fail itself.

Actor Reference Types

Actors are referenced using the following types:

- **Purely local actor references**, used by actor systems not configured for being used in a network. Thus, these actor references will not work if sent across a network connection to a remote JVM.
- **Local actor references**, used when remoting is enabled. Those references represent actors within the same JVM; it is – however – possible to send them to other network nodes by including protocol and remote addressing information.
- **Remote actor references** represent actors reachable by remote communication. On the transmission of messages to these actors, they get serialized and sent to the remote JVM. Internally, Akka utilizes the netty server[3] for remoting.
- Special types of actor references include the `DeadLetterActorRef`, where messages are routed to whose destination is not or no longer existent (hence they are called dead letters).

Actor Reference Naming

As mentioned, actors are created hierarchically. For the naming of actors, the parent-child links give a sequence of names forming a URI. The actor path consists of the address component, describing the protocol and location by which the corresponding actor is reachable, followed by the names of the actors in the hierarchy:

- `akka://my-sys/user/service-a/worker1`
  represents a purely local reference
- `akka.tcp://my-sys@host.example.com:5678/user/service-b`
  represents a remote reference. Remote hosts are obtained by the naming pattern:
  `akka.<protocol>://<system>@<host>:<port>/<actorpath>`

Akka also allows to create child actors on a remote node. The remote system places the new actor on a special path for this purpose. The supervising actor will be the remote actor reference to the actor that triggered the creation, which is shown in Figure 2.4.

[3] `https://netty.io`

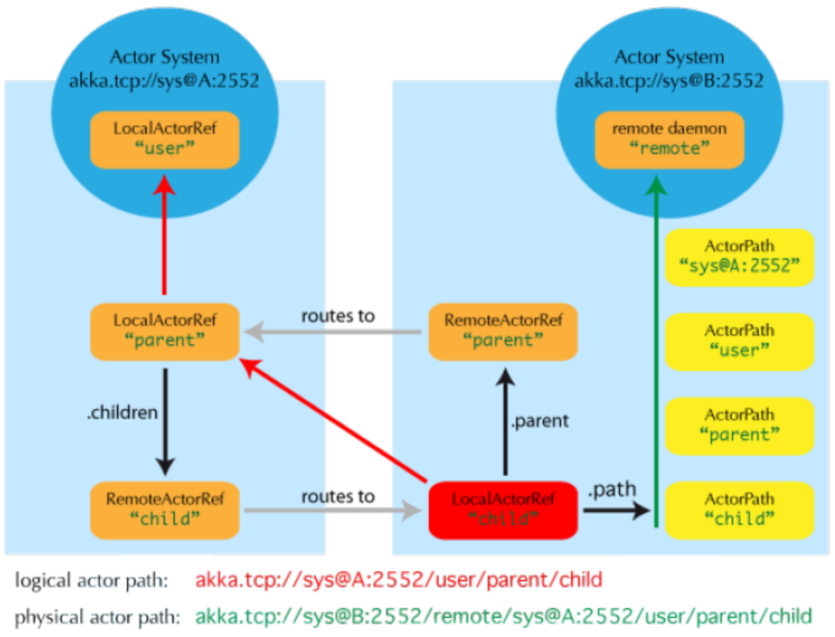


Figure 2.4: Remote child actors in Akka [44]

Communication between actors

The interaction of actors is achieved by exchanging messages. Communication between actors works the same way whether they are on the same local JVM or remote communication is involved. Consequently, location transparency is provided. The latency and reliability, however, cannot be guaranteed to be equal in both cases due to dependencies on the underlying network. Entirely reliable networks cannot exist, and latency cannot be zero whenever communication over a network is involved (cf. one of the eight pitfalls in developing distributed systems described by *Peter Deutsch* [57]).

Messages are delivered in an at-most-once kind, such that no guaranteed delivery can be assumed. If they cannot be delivered, they are put to the previously mentioned dead letter actor reference `"/deadLetters"` – actors can subscribe to this special address to get informed on failed message deliveries. Such dead letters are generated at the node determining that the send operation has failed, which can either be the local system (e.g., if no network connection can be established) or the remote one (e.g., if the actor the message is being sent to does not exist).

Send messages Messages can be sent to another actor by one of two possible methods:

1. `tell`
   sends a message asynchronously and returns immediately. As such it implements a fire-and-forget manner of message transmission.

2. `ask`
   sends a message asynchronously and returns a `Future` for a possible reply message by the recipient.

Serialization Akka uses two built-in serializers, the Java default serializer and the Google Protocol Buffers serializer. Furthermore, it is possible to develop an own serialization implementation.

Receive messages Each actor possesses exactly one mailbox, in which messages destined for that actor are transmitted to and enqueued. The default mailbox implementation uses an unbounded queue; therefore it introduces FIFO behavior for message processing. Different implementations of mailboxes allow altering this behavior, e.g. by utilizing a prioritization of messages. At each time, the thus defined next message in the queue is processed; Akka does neither allow to scan through the mailbox nor to process more than one message at once. When a message is received, the `onReceive` method of the actor is called with this message. As described before, the recipient can reply (answer) to a message if the message issuer used the `ask` method.

Life cycle of an actor

There are various methods to terminate an actor, which are also performed asynchronously. An actor may decide to stop itself. It is also possible to stop its child actors or send special types of messages to other actors to request (graceful stop message) or force them to stop themselves (*PoisonPill* message, *Kill* message).

2.3 Vert.x and Gridlink

Vert.x[4], hosted by the Eclipse Foundation, provides a distributed message bus using *Hazelcast*[5] for clustering. Applications – in vert.x terms called *Verticles* – within one subnet form a cluster of known instances during their execution, by default being discovered using multicast messaging [4]. These verticles communicate over the event bus by either using direct addressing (i.e., request-response pattern) or a publish/subscribe mechanism. One of its main features is its polyglot design, i.e., it allows cross-platform applications with current support for Java, JavaScript, Groovy, Ruby, Ceylon, Scala, and Kotlin languages.

It shares some similarities with the Actor Model in concurrency, scaling and deployment; vert.x does, however, not claim to be a strict implementation of the Actor Model [2]. A verticle is thus comparable to an actor; there are, however, neither hierarchies in naming nor parent-child relations between verticles. Similar to Akka, everything in vert.x is executed asynchronously and developed using its non-blocking API. Vert.x's API is

[4]`http://vertx.io`
[5]`http://hazelcast.org`

mainly event-driven; handlers are registered to get informed whenever an event occurs, e.g., on receiving a message. One designated thread – the event loop – delivers events to registered handlers as they arrive [2]. By this, vert.x shows to be scalable, allowing several verticles to run concurrently on one node. Vert.x extensions, including numerous additional libraries to create web services, servers, or connections to various databases shall not be covered here.

Based on vert.x, the Gridlink [17, 27, 16], which provides an extended API for Java for service-based environments, was developed. It improves modularity by introducing new features like a service registry and application provisioning, such that applications can be managed (deployed/updated/removed) during runtime.

A node is a bundle of modules running on a JVM (Java Virtual Machine). A module is a Java program that (1) implements a dedicated functionality, (2) takes over one or more roles, (3) is addressable and reachable over the event bus via one (or more) role address(es), and (4) provides functions to other modules [17]. Figure 2.5 shows some modules running concurrently on one node, using a NodeManager module to dynamically load applications within one JVM. It manages multiple other Gridlink modules within its JVM even though these modules may be installed, updated, and uninstalled during the runtime of the node and other modules.

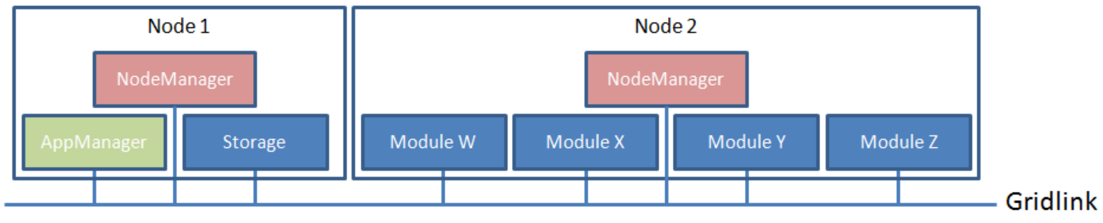


Figure 2.5: Nodes and modules in Gridlink [16]

Referring to the vert.x terminology, a Gridlink module is a verticle. A module registers itself for roles, where it is then accessible via the communication network, i.e., where it can receive and handle messages. Thus, the receiver of a message is not determined by its module address, but by the role address. Modules serving as message producers only (i.e., they are not intended to receive any requests from other modules) do not need to have any role assigned. They are not addressable and thus cannot receive or react to any messages from other modules. Roles are claimed by an intended receiver module, as shown in Figure 2.6, where a storage module takes over – beside others – the role storage, by which it is reachable by other modules.

Services are addressed by roles and service names. A role provides handlers for serving one or more services (e.g., a service handler for adding a measurement to the data storage) and can hence be seen as a container grouping related services. Figure 2.6 shows a module *StorageModule* registered for role *storage*, providing several services (e.g., *createDataPoint*).

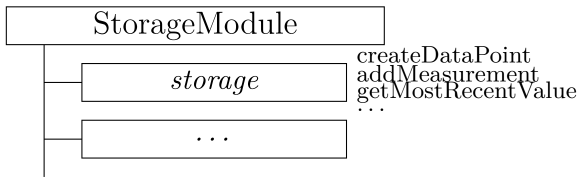


Figure 2.6: Modules, roles, and services in Gridlink [17]

In mapping the concepts to the Actor Model, it can be seen that a role is comparable to an actor. In Gridlink, a module is a vert.x verticle grouping various roles. Gridlink thus shifts the compatibility of concepts, as in vert.x the verticles were described as comparable to actors.

Communication between modules

The communication between modules is handled by messages that are transmitted over the vert.x bus. Three types of message transmission are supported:

- `send`
  The message is transmitted to one module which holds the role specified in the request/event.
- `sendWithTimeout`
  The message is transmitted to one module which holds the role specified in the request/event. On the reception of a reply within a maximum specified time, the reply handler is called asynchronously at the sender of the original message.
- `publish`
  The message is transmitted to all modules which hold the role (topic) specified in the request/event.

If more than one module is registered for the same role, only one of them is chosen for delivery using round-robin fashion when using `send` or `sendWithTimeout`.

vert.x and Gridlink do not require specific message formats, as implementations of requests, replies and events are use-case-specific. Messages in Gridlink consist of a type and an optional payload (e.g., the data point) and are transmitted (per default) as JSON messages to the specified recipient. Before a message is transmitted, it can be modified using configured proxies to do additional message processing steps, including the modification of a message (e.g., for encryption) [27, 16]. Respective proxies are executed at the message's recipient. Afterward, the registered service handler for the received message type is called to process the message, which includes issuing a reply message when using `sendWithTimeout`.

2.4 WS-BPEL

WS-BPEL (OASIS Web Services Business Process Execution Language) is a standardized XML-based language for connecting business processes implemented as web services [47]. It contains a model and a grammar for describing the behavior of business processes and the interactions between them. Processes can either be described as abstract or as executable – the latter ones are fully specified and thus can be executed. The core of WS-BPEL is the peer-to-peer interaction between services described in the Web Services Description Language (WSDL). A process definition defines *PartnerLink* relations for incoming and outgoing messages (Figure 2.7). Messages between interacting parties are exchanged by use of the exposed web services, and serialized into the WSDL message format in XML.

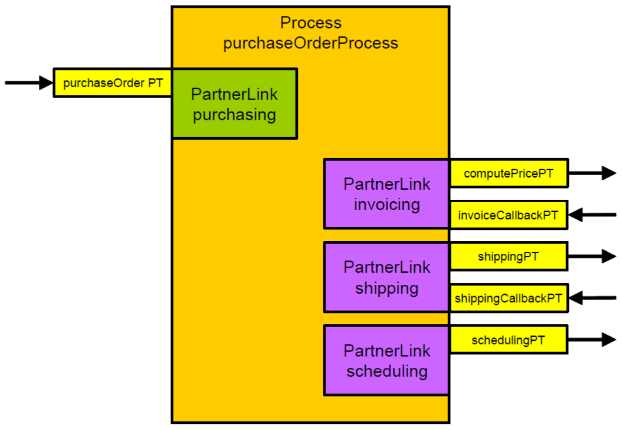


Figure 2.7: WS-BPEL PartnerLink overview [47]

Processes are exposed as WSDL services and written in XML. They consist of activities, that are divided into basic activites describing elemental steps of the process behavior, and structured activities for control-flow logic containing basic and/or structured activities. Examples for basic activities include

- the `<receive>` activity waiting for a matching message to arrive,
- the `<assign>` activity to update values of variables,
- the `<reply>` activity to send a message in reply to a received message,

Examples for structured activities are

- the `<sequence>` activity defining a collection of activities to be executed sequentially,
- the `<flow>` activity defining a collection of activities to be executed concurrently,

- the `<if>` activity to choose one of the children activities to be executed,
- the `<while>` activity to repeat the child activity,
- the `<pick>` activity to wait for one of a set of possible messages to arrive,

Services are addressed and called by use of their WSDL service name. Processes are instantiated when one of the start activities (e.g., `receive`, `pick` – they are also known as inbound message activities) are triggered.

2.5 GigaSpaces XAP

Due to sharing some similarities with the Peer Model, from a high number of tuple space implementations[6], GigaSpaces eXtreme Application Platform (XAP)[7] was chosen for the evaluation. It is a commercial project, where an – although feature-limited – open source variant is also available. It provides a high-end, scalable, high-performance application server where applications can either be written in Java, .NET, or C++. Thus GigaSpaces XAP allows limited language interoperability; however, it is one of the features not included in the open source version.

Messages can be retrieved from the space by using either a `READ` query (non-consuming), a `TAKE` query (consuming), or a `CLEAR` query (deletes the entry but does not return it). There are three types how queries can be built:

- Query by ID
- Query by Template
- Query by SQL

It is possible to request either one (e.g., `take`) or multiple (e.g., `takeMultiple`) entries, in the latter case the number of entries can be limited to a maximum number. All operations usually block until they can be successfully executed, but there are also methods (e.g., `takeIfExists`) that are non-blocking and methods that are executed asynchronously (e.g., `asyncTake`). In summary, the main supported operations of the space are:

- Write objects into the space (`writeXXX`)
- Change objects in the space (`changeXXX`)
- Reading objects from the space (`readXXX`)

[6]e.g., as listed by example in `http://procol.sourceforge.net/dokuwiki/doku.php?id=list-tuplespaces`

[7]`https://www.gigaspaces.com/product/xap`

- Removing objects from the space (`takeXXX`)

Objects can have a Time-To-Live (TTL) set. Some of the space operations can generate notifications (e.g., `write`, `take`) when they are executed to inform other entities attached to the space. Supported features for SQL queries are limited, for example, neither multiple tables select nor joins are possible.

Gigaspaces XAP uses Jini for addressing, lookup, and discovery[8]; the general format of the space URL is `<protocol>://<lookup_service_hostname>:<port>/<space_container_name>/<space_name>?<properties>`. For serialization in GigaSpaces XAP, by default the integrated serializer of Java is used. This can be changed by declaring the class of the objects that are written into the space as `Externalizable`. By implementing its methods `writeExternal` and `readExternal` an own serialization mechanism can be integrated (for example, the GigaSpaces documentation[9] shows how to integrate the Kryo serializer[10]). Limited platform interoperability betweeen .NET and Java is briefly described in the documentation[11], though rules need to be followed and additional steps are required for the serialization of user-defined objects on the .NET side. Various remoting technologies can be used that are yielded by Spring's remoting[12].

2.6 Evaluation

A comparison between the introduced frameworks and the various already existing Peer Model implementations shall be given here. The main focus of comparison – according to the focus of this thesis – shall naturally be laid on the communication features. In consequence, by finding missing features the requirements for this thesis and the Java implementation shall be laid out.

2.6.1 Acting on messages

The most significant difference to the Peer Model is that Akka, Vert.x, and WS-BPEL act on one and only one individual received message. By using `pick`, in WS-BPEL various possible message types to act on can be specified but still it acts on one message. Besides the already existing Peer Model implementations, only GigaSpaces XAP can act on more than one message.

For the comparison, three sample use cases shall be introduced:

Use Case 1: A function shall be executed every time five messages of type A have been received.

[8]`https://docs.gigaspaces.com/xap/14.0/overview/about-jini.html`
[9]`https://docs.gigaspaces.com/xap/14.0/dev-java/custom-serialization.html`
[10]`https://github.com/EsotericSoftware/kryo`
[11]`https://docs.gigaspaces.com/xap/14.0/dev-dotnet/interoperability.html`
[12]`https://docs.gigaspaces.com/xap/14.0/dev-java/space-based-remoting-overview.html`

Use Case 2: A function shall be executed if one message of type A and one message of type B have been received. The message of type A, however, shall not be removed after it has been processed, i.e., as long as the message of type A is still available, the function shall be executed each time a message of type B arrives.

Use Case 3: A function shall be executed whenever no entry of type C is available.

All of these use cases are complicated or even impossible to be implemented using Akka, vert.x, and WS-BPEL. It is neither possible to process more than one message at once nor to process the same message more than once nor to act anyhow if no message is pending. Furthermore, the frameworks do not support multiple preconditions. As each actor only has exactly one mailbox in Akka, no different behavior can be registered for different types of messages. Once a message in Akka is delivered to the mailbox, the actor implementation acts on this one message. A conditional execution is required to be implemented within the actor itself.

GigaSpaces XAP allows to process more than one message at once; however, multiple takes can act on only on message type. Furthermore, only a maximum number of entries to be taken can be specified, but not a minimum number. That renders all of the mentioned use cases to be impossible there. The Peer Model is more powerful in these scenarios: all of the use cases are easily possible to be implemented.

In result, the following enumeration summarizes the possibilities:

take (i.e., consuming read) exactly one entry:
Akka ☑ vert.x/Gridlink ☑ WS-BPEL ☑ GigaSpaces XAP ☑ Peer Model ☑

read (i.e., non-consuming) exactly one entry:
Akka ☐ vert.x/Gridlink ☐ WS-BPEL ☐ GigaSpaces XAP ☑ Peer Model ☑

act on non-available entry:
Akka ☐ vert.x/Gridlink ☐ WS-BPEL ☐ GigaSpaces XAP ☐ Peer Model ☑

take more than one entry of the same type:
Akka ☐ vert.x/Gridlink ☐ WS-BPEL ☐ GigaSpaces XAP ☑ Peer Model ☑

take more than one entry of different types:
Akka ☐ vert.x/Gridlink ☐ WS-BPEL ☐ GigaSpaces XAP ☐ Peer Model ☑

2.6.2 Interoperability

While communication with instances using the same implementation is possible in all evaluated frameworks (note that this is a central requirement for any distributed system), communication with instances using other implementations are rarely supported. Of all evaluated frameworks, only vert.x can convince with platform-independent communication to instances in Java, JavaScript, Groovy, Ruby, Ceylon, Scala, and Kotlin. Note that the Gridlink extension – however – is restricted to Java. Gigaspaces XAP at least allows communication between instances in Java, .NET, and C++; however, additional effort for coding is necessary for serialization. Earlier Peer Model implementations do not allow any communication with instances in other languages.

Communication with implementations in another language
Akka ☐ vert.x/Gridlink ☑ WS-BPEL ☐ GigaSpaces XAP $\boxed{\sim}$
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐

2.6.3 Discovery

Most of the compared frameworks do not include any discovery mechanism; they require remote instances to be known and addressable. Vert.x and GigaSpaces XAP use multicasting for discovery; vert.x by using Hazelcast for building a cluster of all known and reachable instances. In Gigaspaces XAP, a lookup service provides a mechanism for services to discover each other, by default using multicasting. Services register themselves in the lookup service such that they can be found by other services. The client gets a service proxy for the requested server; subsequently a direct connection between the parties is used. Earlier Peer Model implementations do not implement any discovery mechanism.

Akka ☐ vert.x/Gridlink ☑ WS-BPEL ☐ GigaSpaces XAP ☑
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐

2.6.4 Dynamic exchange of logics

Naturally, the Peer Model meta-model introduced in [20] cannot be evaluated with the other frameworks in related work. However, in other frameworks there might be other mechanisms to dynamically replace logics during execution of the system.

Though hardly comparable to the Peer Model approach, Akka, vert.x and Gridlink include such a concept. In Akka, there is a parent-child relation between actors; they can start child actors and terminate them during the system's runtime. Vert.x allows to execute several verticles concurrently within one JVM. One of the major added features of Gridlink is provisioning, i.e., the ability to install, update, configure, etc, modules during the runtime and issued from the remote site [17]. The AppManager module is responsible for receiving provisioning requests and their handling; e.g., for the installation of another module, the AppManager downloads the module archive, extracts it and launches it. Using the same mechanism, Gridlink proxies can be added and removed dynamically during the runtime of a module [27]. Using node management (cf. Figure 2.5), multiple Gridlink modules can be managed within one JVM without influencing installation, update and uninstallation during the runtime of the node and other modules [16].

There is currently not yet an implementation of the Peer Model that supports the meta-model. All previous implementations neither have a special entry type for wirings and peers, nor include special containers for their handling.

Akka $\boxed{\sim}$ vert.x/Gridlink $\boxed{\sim}$ WS-BPEL ☐ GigaSpaces XAP ☐
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐

2.6.5 Remote component creation and termination

For scalability use cases it is of interest whether components on other components could easily be created and terminated again. In Akka, child actors can be created on remote nodes, while in vert.x no such mechanism exists – there are no parent-child relations in vert.x. However, the provisioning features of Gridlink (cf. Section 2.6.4) introduce a mechanism to install, update, and configure modules from the remote (operator) side. Such an mechanism is also neither available in WS-BPEL nor in GigaSpaces XAP.

For the Peer Model implementations, the remote component creation and termination functionality is tightly connected with the meta-model. As described, there is currently not yet an implementation of the Peer Model that supports the meta-model.

Akka ☑ vert.x/Gridlink $\boxed{\sim}$ WS-BPEL ☐ GigaSpaces XAP ☐
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐

2.6.6 Exceptions

Communication failures and expiration of messages shall be properly handled. In vert.x, when using `sendWithTimeout` an exception is generated if no reply was received within the timeout [27]. Comparable to exception entries, Gridlink implements the `ErrorReply` type that can be used as reply type. In either case, Gridlink executes an `ErrorReplyHandler` to react on a communication exception or on receiving the `ErrorReply` from the recipient. As the concept of TTL does not exist, there are no exception types for handling of expired messages. Akka uses a comparable concept with the message type `ReceiveTimeout` [44]. The recipient can also reply with the status class `Status.Failure`. WS-BPEL defines the section `faultHandlers` to define activities to be performed in response to faults resulting from invocation of services [47]. They can be generated by using the activity `throw`. In Gigaspaces XAP, a lease time can be defined when writing entries into the space[13]. By using a listener, the sender can get notified on the expiration of the lease.

[13] `https://docs.gigaspaces.com/xap/14.0/dev-java/leases-automatic-expiration.html`

In the Peer Model, exceptions form a special type of entries that are currently implemented only in the Mobile Peer Model. They are generated on an expired TTL as well as on transmission failures. Furthermore, the Mobile Peer Model includes a pattern for automatically resending entries on failures. In the PeerSpace.NET, no exception entry type exists but the developer can add an error callback method to be executed on transmission failures, while in the Embedded PeerSpace no such concept is implemented.

Akka ☑ vert.x/Gridlink ☑ WS-BPEL ☑ GigaSpaces XAP ☑
Embedded PeerSpace ☐ PeerSpace.NET ⊟ Mobile Peer Model ☑

2.6.7 Results and Todos

The previous subsections defined the features that were identified as missing in the evaluated frameworks as well as in the previous Peer Model implementations. The thesis and the Java enterprise implementation that shall be delivered with this thesis shall thus address these missing features.

- The thesis shall define a platform-independent communication format to let instances implemented in different languages communicate with each other. The implementation is required to implement that format (Section 2.6.2).
- Furthermore, the thesis shall define a platform-independent discovery mechanism to let instances implemented in different languages find each other. The implementation is required to implement that protocol (Section 2.6.3).
- The implementation shall include the proposed meta-model and thus shall handle wirings and peers as special types of entries (Section 2.6.4).
- The implementation shall allow peer and wiring entries to be transmitted to remote instances like any other entry. These wirings and peers shall be added to the remote instance and work as intended; wirings shall also be able to remove those entries – and thus the components – later on (Section 2.6.5).
- Furthermore, the implementation shall introduce exception entries as additional entry type, especially to handle expired entries and communication failures properly (Section 2.6.6).

CHAPTER 3

Peer Model In-Depth

The concepts of the Peer Model can be understood best by giving a stepwise introduction. Section 3.1 introduces the concept of **entries** to encapsulate information. In Section 3.2 the eponymous component, the **peer**, is introduced as well as the peer's **containers**. Section 3.3 defines **wirings** to transport entries between containers, and Section 3.4 handles **services** that implement user-defined application logic. In Section 3.5, **links** that are used to specify wirings' and services' invocations and implications are covered in more detail. Section 3.6 briefly introduces the distribution of entries, which will be described in more detail in Chapter 5. In Section 3.7, the dynamic concepts of the Peer Model are introduced. Section 3.8 summarizes the generation of traces in a Peer Model run.

3.1 Entries

An entry encapsulates information by representing – for example – events, requests, answers, exceptions, or data [39]. It is created, modified and transmitted by other components that will be described later. As the example in Figure 3.1 shows, elements of an entry can be divided into three categories:

- the required **basic elements**
- generally optional **co-data**,
  however, they contain properties that are usually required even for the simplest examples
- optional **app-data**

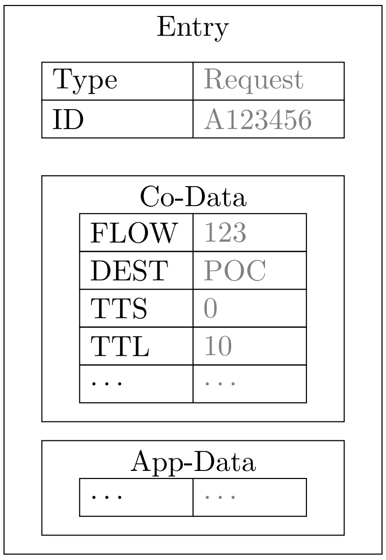


Figure 3.1: An entry of type *Request*

For the rest of this thesis, examples will show entries in a simplified way as a circle containing the type as shown in Figure 3.2.

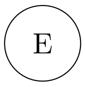


Figure 3.2: A simplified entry of type *E*

3.1.1 Basic Elements

An entry consists of two required fields of information:

- the **Type**, used to distinguish different types of entries for a proper selection in links (Section 3.5) to control the coordination flow in the system. Multiple entries may share the same entry type.
- the **ID**, which is a unique identifier for an entry. The ID is set by the Peer Model implementation when creating an entry and cannot be controlled by the developer. As the type may be shared between more entries, the ID is primarily used for logging purposes, i.e., to see which of the entries was affected.

  When using communication with remote instances, it would not be possible to guarantee the uniqueness of the id when using an integer sequence. However, an entry's ID can be set to the next value of the instance-specific integer sequence when receiving an entry from a remote location. Another option would be the use of a randomly generated 128 bit long Universally Unique Identifier (UUID) [42].

Although not guaranteed to get really unique IDs this option is considered to be acceptable as the probability of generating two identical UUIDs is very low.

3.1.2 Coordination Data (co-data)

These kinds of data are used for system-internal mechanisms like the entry selection [38]. Basic (optional) co-data properties (**system co-data**) are specified as follows:

- **FLOW**
  While the entry's ID is unique, several entries may share the same flow id to mark data that belongs together. The entries' flow ids can then be used for a correlation of these entries.
- **DEST**
  The destination property represents the address to which the entry shall be transmitted to. It may either be a container of the same peer, a sub-peer, a peer in the same instance, or a peer of another runtime which requires the transport of the entry over the network. An entry dispatcher or I/O peer is responsible for the transmission of the entry to the correct destination.
- **TTS**
  The time-to-start property represents the time until which the entry must not be read or taken from a container. Until this point in time is reached, the entry remains invisible to the container. It is accessible only after the TTS is expired. If no TTS is set, the entry is immediately valid once created and can be taken.
- **TTL**
  The time-to-live property represents the lifetime of an entry, i.e., for which time it stays valid. If this time is exceeded and it was not yet taken, the entry becomes invalid and is removed from the container. The handling of expired entries is an implementation issue and can either result in a timeout exception [38, 39] or simply the removal of the entry as in the embedded implementation [37]. If no TTL is set, the entry may stay valid forever if it is never taken.

The user may specify additional coordination data (**user co-data**) that are queryable in guard links (described in Section 3.3 and 3.5).

3.1.3 Application Data (app-data)

These kinds of data are used for the designated functionalities of the application. Therefore, it remains entirely user-specified what is transported in this section. The existence of app-data in an entry is transparent to the Peer Model coordination; consequently, they are not queryable in guard links (described in Section 3.3 and 3.5).

3.2 Peers and Containers

The peer is the eponymous component of the Peer Model. It has a name by which it is addressable, called the *Peer Address*. A peer may itself include sub-peers as child components addressable by their name in conjunction with the parent's peer address. The behavior of a peer is specified by its nested components, namely the sub-peers, the wirings and their associated services.

Each peer consists of one or more containers that serve as temporary storage for entries. A container has a name (*Container Address*) under which – in conjunction with the peer address – it is addressable. As specified in [38], a (default) peer contains two containers:

- a peer-in-container (PIC), intended mainly for entries that are input parameters of the internal components.
- a peer-out-container (POC), intended mainly to store entries that are the outcome of the peer's internal components.

Figure 3.3 shows a default peer having the described containers and integrating one sub-peer.

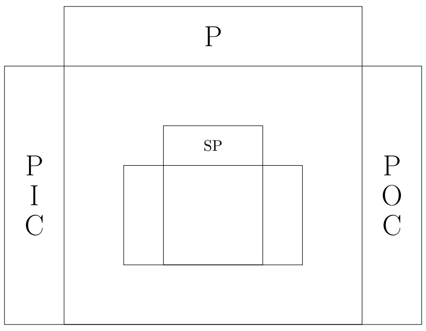


Figure 3.3: A peer having two containers *PIC* and *POC* and a sub-peer *SP*

The basic model was extended in [20] by introducing additional containers for a meta-model that allows to dynamically add and remove peers' sub-components during the instance's runtime. A detailed view on these concepts needs to be postponed to Section 3.7 to allow the introduction of the missing components first.

3.3 Wirings

A wiring is an internal component of a peer intended to transport entries between containers of its peer and direct sub-peers, belonging to exactly one peer and having a name. It extends the concept of Petri net transitions [37].

Guard links (input links) specify which entries are required for the execution of a wiring (pre-conditions). A wiring is only activated if the conditions of all guard links are fulfilled.

The selected entries are (depending on the guard link's type) copied or moved to the *entry collection*, which is the wiring's internal container. The internal logic executes the wiring's nested components, the *services*, described further in Section 3.4. After their execution, entries of the entry collection are eventually moved back to the peer's containers or another specified location by use of *Action Links* (output links). Not all actions need to succeed, action links that cannot be successfully executed are ignored. Entries that would remain in the entry collection after completion of the wiring are discarded [38].

Figure 3.4 shows a wiring with its guard and action links, which will be covered in more detail in Section 3.5.

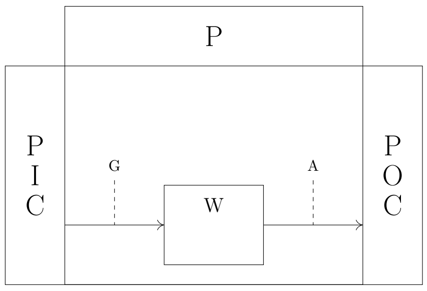


Figure 3.4: A peer *P* having a wiring *W* with one guard *G* and one action *A*

3.4 Services

A wiring incorporates zero or more services that are sequentially executed in order of their specification. Services are the only place for business logic provided by programmers while wirings model the coordination logic [39]. Services are called by a wiring to execute user-defined code.

Comparable to wiring guard links, a *service guard link* defines the entries that are required by the service. The wiring tests whether these entries are contained in the entry collection and moves these entries to the service before its execution. The service can be defined as either required or optional. It is only executed if all service guard links' conditions are fulfilled; in cases of a required execution with at least one service guard's condition not being fulfilled, an exception is thrown. Inside the service code, the user can process these "parameter" entries and may create new entries. Entries may eventually be returned to the wiring's entry collection after the service's execution by use of a *service action link*. The described link specifications are introduced in more depth in Section 3.5.

Figure 3.5 shows a wiring that incorporates one service, showing the service's guard and action links. Service Guard and Service Action Links will be covered in more detail in Section 3.5.

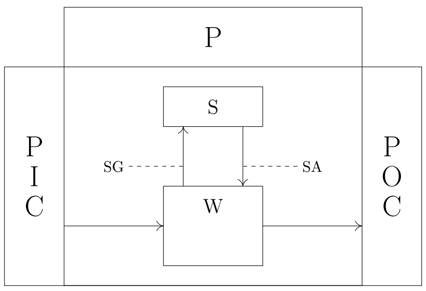


Figure 3.5: A peer *P* with a wiring *W* that incorporates a service *S* with one service guard *SG* and one service action *SA*

3.5 Links

We introduced *Wiring Guards* and *Wiring Actions* in Section 3.3; and *Service Guards* and *Service Actions* in Section 3.4, respectively. All these components are types of links that are covered in depth in this section.

A *Wiring Guard* connects one peer container with the wiring's entry collection determining the conditions in which the wiring can be executed and moving the according entries if all guards are fulfilled. A *Service Guard* works likewise between entry collection and service; entries can then be used in the service in the called user-defined code. After the service's execution, a *Service Action* moves entries back to the wiring's entry collection. Once all services are executed, *Wiring Action*s define which entries are moved to the according peer containers.

A selection of entries is specified on the link in a template-like form. It contains

- an **entry type** (mandatory)
  Entry types were already covered in Section 3.1.1.

- a **link operation** (mandatory)
  Link operations will be further covered in Section 3.5.1.

- a **count specification** (optional)
  The count specification determines the number of entries that are required. It is covered in Section 3.5.2.

- a **query** (optional)
  This concept further restricts the entries that fulfill the requirements as it allows optional queries on entries' coordination properties. It is covered in Section 3.5.3.

- a selection whether entries should be handled in a **flow-dependent** or in a flow-independent (default) way, covered in Section 3.5.4.

3.5.1 Link Operation Type

The link operation specifies in which mode entries are retrieved from the source container. They are conceptually based on operations in space-based middleware like Linda [15]. In Linda, operations are basically “in” for a blocking consuming read, “rd” for a blocking non-consuming read, and “out” for a non-blocking write operation on the shared space. The eXtensible Virtual Shared Memory (XVSM) [41] extended the tuple space approach of Linda by introducing containers as a store for entries. Operations are “take” for a blocking consuming read operation, “read” for a blocking non-consuming read operation, and “write” for a non-blocking write operation. The nomenclature is taken over for link operations in the Peer Model, which however provides extensions to the described operations. According to [38] at least one of the guards needs to specify a consuming get operation (take).

To illustrate the different link operation types, consider the examples shown in the according figures. For all examples in this paragraph, Figure 3.6 shows the states of two containers before the firing of the guard. The left container is termed *source container*, the right container *destination container*, respectively. The enumerated link operation types refer to the operation made on the source container; at the destination container, a write operation is performed in all cases except for TEST and DELETE guards.

SOURCE
E
DESTINATION

Figure 3.6: Contents of the containers before the guard execution

TAKE The effects of the *take* operation on the container’s states in Figure 3.6 are shown in Figure 3.7. It takes (i.e., removes) the entry from the source container and transmits it to the destination container.

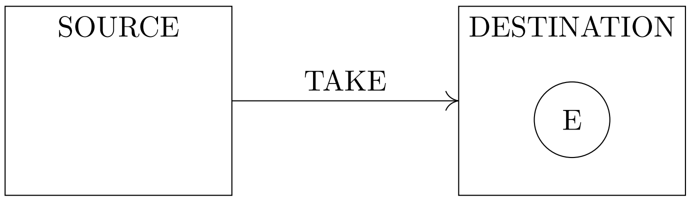


Figure 3.7: Contents of containers after the **take** guard execution

READ The effects of the *read* operation on the container’s states in Figure 3.6 are shown in Figure 3.8. It leaves the entry in the source container, copies it and transmits the copy to the destination container.

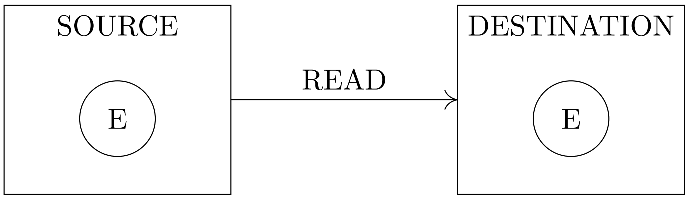


Figure 3.8: Contents of containers after the **read** guard execution

DELETE The effects of the *delete* operation on the container's states in Figure 3.6 are shown in Figure 3.9. It is a specialization of the *take* operation. The entry is removed from the source container, however not transmitted to the destination container.

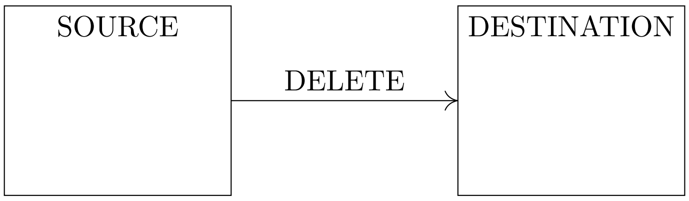


Figure 3.9: Contents of containers after the **delete** guard execution

TEST The effects of the *test* operation on the container's states in Figure 3.6 are shown in Figure 3.10. It is a specialization of the *read* operation. As the figure shows, it may seem that this operation is not meaningful at all as it shows no consequences on the source and destination containers; however, a wiring having a *test* guard (especially next to others) will fire only if the according entries are available at the source container.

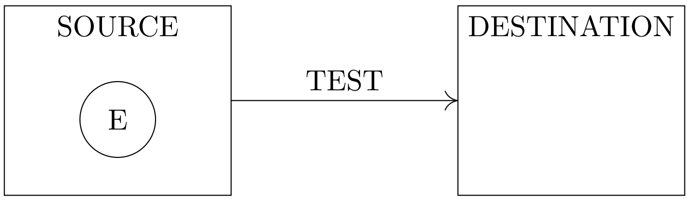


Figure 3.10: Contents of containers after the **test** guard execution

NONE The *none* operation is a specialization of the *test* operation and works differently than the operations described so far. The guard is only satisfied if no according entry fulfills the selection in the container.

CREATE The *create* operation works contrary to the *delete* operation. It does not take any entries from the source container, but generates new entries to be moved to the destination container. Accordingly, this operation cannot be unsatisfied in any link.

3.5.2 Count Specification

The count selection specifies the number of required entries of this type to be fetched. It can be specified to either return a defined number of entries or any number of entries in an interval between a given minimum and a maximum count. In this interval, as many entries as possible are taken. Simplified notations are $< n$, $\leq n$, $= n$, $\geq n$ and $> n$, all having the common meaning. Furthermore, `ALL` is a defined simplification for ≥ 0 and is therefore always fulfilled. Evidently, a `NONE` operation returns exactly zero entries in case of success.

If no count is specified, per default *exactly one* available entry that fulfills the query is fetched. Examples are shown in Figures 3.11, 3.12 and 3.13.

3.5.3 Query Selection

Entries can also be selected by specifying a query using properties defined in user co-data. Queries can also be combined by using `AND`, `OR`, `XOR` or `NOT` operators. Data in app-data are not queryable.

Example. Assuming two entries

$$E(codata : value = 0), E(codata : value = 1),$$

a wiring having the link

$$TAKE\ E(ALL, codata : value > 0)$$

would take only the latter one of the entries.

3.5.4 Flow Dependent Links

As defined in system co-data, the FLOW property can be set to an ID. If the link is flow-dependent, only entries sharing the same flow ID or entries that have none set are retrieved.

Example. Assuming four entries

$$E(flow = 1), E(flow = 1), E(flow = null), E(flow = 2),$$

a wiring having the flow-dependent link

$$TAKE\ E(ALL, flow)$$

would (depending on the container implementation) only take either the first three or the last two entries.

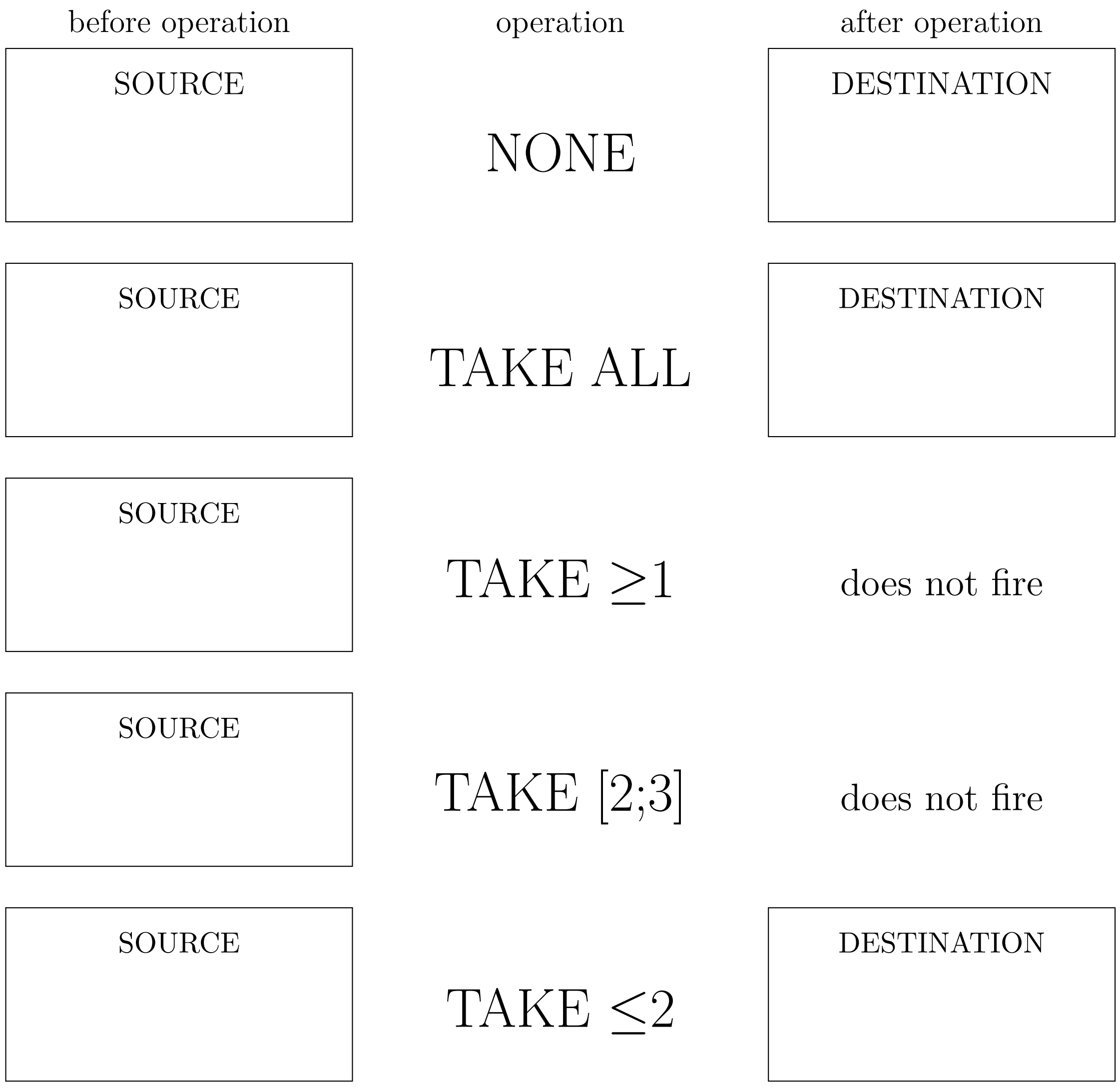


Figure 3.11: Some examples with no entries in the source container

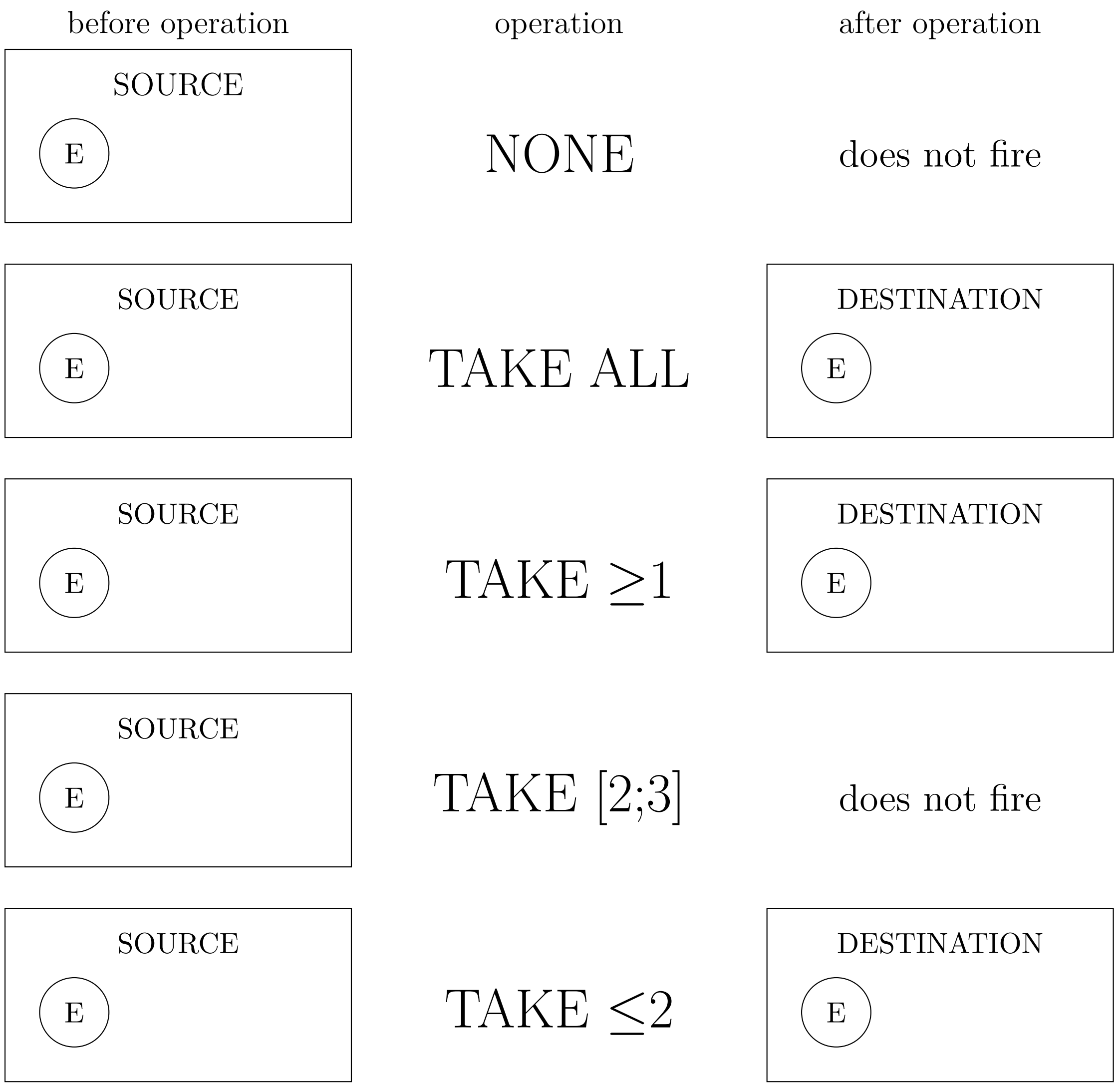


Figure 3.12: Some examples with one entry in the source container

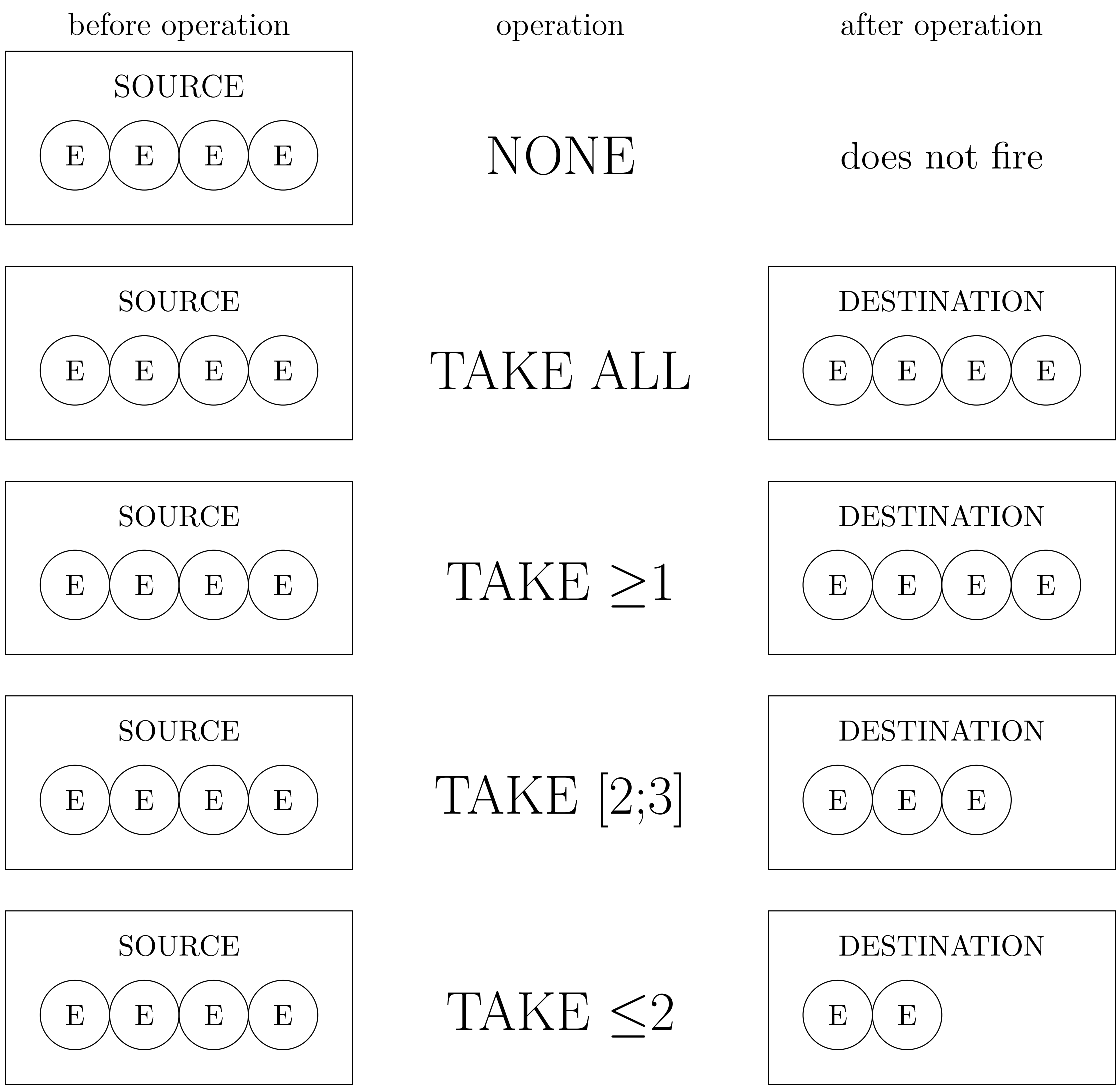


Figure 3.13: Some examples with four entries in the source container

Note, that the FLOW property is evaluated over all links. The first evaluated link specifies which FLOW is tested among all following links.

Example. Assuming two entries

$$E(\mathit{flow} = 1), F(\mathit{flow} = 2),$$

a wiring having the two flow-dependent links

$$\mathit{TAKE}\ E(\geq \mathit{1}, \mathit{flow}), \mathit{TAKE}\ F(\geq \mathit{1}, \mathit{flow})$$

would not fire.

3.6 Distribution

Links handle the delivery of entries from one source to one destination container. The source container can either be a container of the own peer or of its direct sub-peers. Containers of other peers including sub-peers of an own sub-peer are not accessible for selection as source container. The destination container of a link may be any container of the own peer or its direct sub-peers.

It is thus required to allow a transmission of entries over the barriers of one peer. As such, the DEST property of the entry's co-data is used that may contain a local container address, a peer address of another peer or even a remote address, requiring naming specifications that will be introduced in Section 5.2. Transmission of these entries to local or remote destinations is achieved by the concept of an I/O peer. The I/O peer is a black box component that sends entries to and receives entries from other destinations. The implementation of this I/O peer will naturally be different for local and remote destinations. While at this point the introduction of the concept is sufficient, a decision for the type and format of this communication will be made in Section 5.7.

3.7 Dynamics

In a dynamic Peer Model implementation, it shall be allowed to add and remove components while the Peer Model instance is running. *Craß et al.* [20] thus introduced a meta-model in which wirings and (sub-)peers are represented as special entry types. The basic model was extended by defining additional containers, of which the *Wiring Specification Container (WSC)* and the *Peer Specification Container (PSC)* are of interest in the context of this thesis. These two containers should serve as the store for these entry types, including particular behavior.

The Peer Model Instance can be seen as a Runtime Peer (RTP) that contains all peers on this runtime as its sub-peers. Each peer in term can contain sub-peers. These sub-peers are represented by the entries in the PSC and thus can be written into and taken from this container like any other entry. Entries representing sub-peers however naturally need to be of a defined form as well as modifications of the PSC's contents need to be adequately represented in the instance.

Wirings of peers are represented by entries in the WSC and can be accessed like any other entry, respectively. Like sub-peer entries, those entries representing wirings need to have a defined form and modifications of contents in the WSC need to be adequately represented likewise.

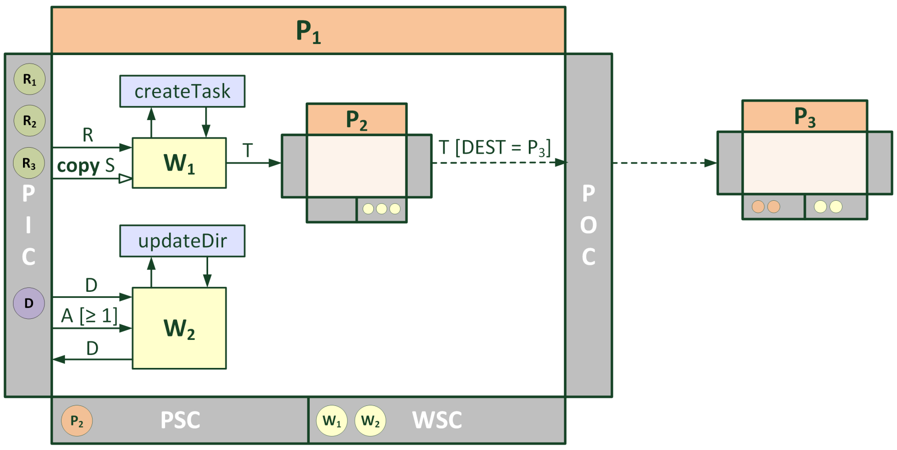


Figure 3.14: Extended meta-model example [20]

Figure 3.14 shows the mutual appearance of entries in the peer's containers and according sub-components in the peer. As such, the figure shows the entry *P2* in the *PSC* and accordingly the sub-peer *P2*; respectively the wiring *W1* corresponds to entry *W1* in the *WSC*. Like any other entry, defined operations like *TAKE* can also be used on *PSC* and *WSC*. In result, the entry is transmitted as usual together with the removal of the sub-peer or wiring in the peer itself. The handling of wirings and peers as a (special) entry indeed allow the transmission of peers and wirings to other instances. In result, this technique may allow a more straightforward implementation of a scalable solution for use cases such as load balancing.

3.8 Tracing

In a highly concurrent system like the Peer Model, debugging and monitoring is a complicated task. The identification of a bug, like a race condition, is difficult. *Csuk*'s diploma thesis [21] introduced a visual tool that allows a post-mortem analysis of a recorded Peer Model run to support developers in debugging processes. This tool enables developers to gain insight into the working of a Peer Model [21]. To that end, Peer Model implementations are urged to generate a log of events in a defined common tracing format based on JSON which is loaded into the visual monitoring tool afterward to allow a stepwise analysis of the run. The Java implementation of the Peer Model is designed to be able to generate such event logs.

A common logging format was specified in [21], consisting of two files:

- a *VIL (Visualisation Intermediate Language)* file that contains the static structure of the Peer Model. All peers, wirings, and services are listed in this file.
- a *TIL (Trace Intermediate Language)* file that contains the chain of logged events being occured during a Peer Model run (*trace*). The flow of entries is shown in the visualizer by linking the events with the VIL structure.

Naturally, a one-to-many relation between a VIL file and TIL files can be presumed, as a given Peer Model structure can be reused in many different recorded event logs. The toolchain for logging in Figure 3.15 shows the steps for generating the files that are then fed into the monitoring tool to allow a visual representation of a run.

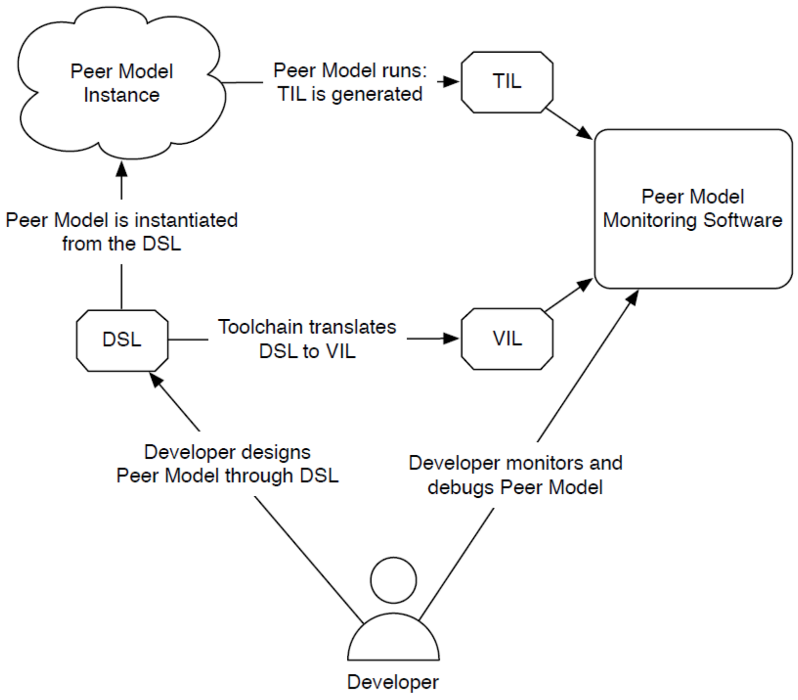

Peer Model Instance
Peer Model runs: TIL is generated
TIL
Peer Model Monitoring Software
Peer Model is instantiated from the DSL
DSL
Toolchain translates DSL to VIL
VIL
Developer designs Peer Model through DSL
Developer monitors and debugs Peer Model
Developer

Figure 3.15: Workflow for post-mortem visual monitoring of a run intended by [21]

3.8.1 Visualisation Intermediate Language (VIL)

The VIL is a snapshot of the current structure of the local Peer Model. Figure 3.15 shows that it was initially intended to generate the VIL file from the PM-DSL by the Peer Model compiler [21]. However, while this is practicable in static environments, in a dynamic

Peer Model implementation, the current structure is not given by the PM-DSL per se. Peers, wirings, and services may be added and removed while the Peer Model is running. Following these considerations, it may be necessary to generate a VIL file not only from the PM-DSL but also to generate this file from a running Peer Model implementation, containing the current structure. Later changes of the structure, for example adding a peer during logging, would remove the relation between TIL and VIL; furthermore, it is not supported by the visual monitoring tool and would require additional work on that tool. The other (current) option (i.e., a modification of the structure is forbidden during the logging) limits the functionality of the dynamic Peer Model and is thus not the optimal solution. However, the generation of the VIL file by the Java implementation remains open as possible future work. For the time being, the term "static representation" used for the VIL file in [21] has to be relaxed such that the structure must be static only while recording a trace.

3.8.2 Trace Intermediate Language (TIL)

The trace consists of a collection of consecutive events that occurred during monitoring of a run or a part of a run. As the TIL file is intended to be generated by a Peer Model implementation, the format was specified in collaboration of the authors of the monitoring tool, the visualizer tool and the Peer Model implementations in active development at that time (C [29], Java). The TIL file contains two parts:

- an *initial state*
  It is not necessary to start the logging procedure at the same time when starting the Peer Model instances. At the start of the logging, the Peer Model may already contain some entries. To support such an initially non-empty Peer Model in the visual monitoring tool, the initial state part defines the entries that are currently located in every container of a peer when logging starts.

- the *log event list*
  The list contains an ordered sequence of events. Each event contains a timestamp when it has happened and information about the entry or entries it is related to. *Csuk* intended to include both entry's co-data and app-data; however, app-data of some entries may require much space and may also be binary, thus not human-readable. The inclusion of app-data in the log is therefore avoided in this implementation. Eight different event types are briefly described in the next subsection. For a more detailed description refer to [21].

Event types

Wiring events The four event types regarding wirings and service invocation are:

- the *guard link* event, indicating the firing of one or more (wiring) guards on a container, leading to an execution of a wiring

- the *service input* event, indicating the firing of one or more (service) guards on the entry collection, leading to an execution of a service
- the *service output* event, corresponding to the return after the execution of a service including entries that are created in the service and put into the wiring's entry collection
- the *action link* event, corresponding to the activation of an action link after execution of a wiring

Transmission events The two types of events regarding transmission are:

- the *destination send* event, indicating the sending of an entry to another local or remote peer by use of the destination property functionality
- the *destination receive* event, indicating the reception of an entry from another local or remote peer

Time-lapse events The two types of events regarding timing properties are:

- the *TTS* event, indicating the expiration of the time specified by the TTS property of an entry in a container
- the *TTL* event, indicating the expiration of the time specified by the TTL property of an entry in a container

CHAPTER 4

Requirements

Requirements are separated into *functional* (Section 4.1) and *non-functional* (Section 4.2) requirements. While functional requirements describe *what* the system shall be able to, the non-functional requirements describe *how* the system shall work regarding quality aspects.

4.1 Functional requirements

The functional requirements describe the particular functions the implementation will be required to include. They mainly stem from the aspects discussed in Section 1.2.

Limited feature completeness (FR1). The implementation shall support the concepts of the Peer Model that were introduced in [38, 37, 39], including the ability to have sub-peers, the specified wiring and service logic, and co-data such as flows, the destination and timing properties (TTS and TTL). As the Peer Model specification has evolved during the writing of this thesis and the development of the Java implementation, some newer features (e.g., transactions, variables, etc.; cf. [35, 36, 40]) will not be supported immediately; their implementation remains open as future work.

Support for the meta-model (FR2). The meta-model was introduced in [20], adopting wirings and peers as particular entry types and further introducing special containers to handle these types of entries. In preparation for the Secure Peer Space that will extend this enterprise implementation with security features, the implementation will be the first one that supports this meta-model. As such, incoming entries of the special types `WIRING` and `PEER` need to be appropriately handled and reflected in the Peer Model runtime. Entries of those types also can be short-living by having a TTL set, and they can be taken just like any other entry. The removal of those entries thus needs to be adequately reflected in the Peer Model runtime, respectively.

Language-independent remoting (FR3). In support of scalable enterprise solutions, a Peer Model instance is required to be able to communicate with other instances. It is defined that those instances are not necessarily required to be written in the same programming language. Thus the implementation needs to support a language-independent mechanism for discovery of instances, as well as the following communication between those instances. Like wirings and peers may come and go during the runtime of an individual instance, remote instances will as well come online and go offline; furthermore, the unreliable network channel between the instances needs to be taken into account.

Execution of user-defined service code (FR4). The implementation of application logic is strictly decoupled from the coordination logic, which is implemented by the components of the Peer Model that were described in Chapter 3 (separation of concerns). The developer is responsible for the appropriate definition of services and their use-case specific implementation (i.e., business logic). Thus, the interfaces for the use-case developer to be able to develop services need to be defined.

4.2 Non-functional requirements

Non-functional requirements describe the quality criteria of the system. They extend the functional requirements to describe *how* the system shall operate and hence describe criteria for the evaluation (Section 7.3).

Interoperability (NFR1). Directly resulting from the functional requirement of language-independent remoting (cf. FR3) is the requirement for interoperability. The implementation thus needs to use mechanisms for discovery and remote communication that do not improperly restrict the number of supported languages and platforms.

Extensibility / Maintainability (NFR2). The concepts that were later introduced into the Peer Model specification and that were defined to be not yet included into the Java implementation require for extensibility (cf. FR1). Thus, the implementation shall easily be able to be extended such that future work can include those and additional features. Furthermore, the source code of the Java implementation needs to be easy to understand to simplify the development of the future work extensions.

Modularity / Exchangeability (NFR3). To keep components exchangeable, Java implementation classes shall be adequately separated into packages and particular modules where appropriate (cf. NFR2). Interfaces between the modules shall be defined to keep cohesion high and coupling low. Furthermore, the platform-independent remoting mechanism, including its protocols (cf. FR3), shall be exchangeable, such that it can be adjusted for remote instance developments in future work.

Scalability (NFR4). As the enterprise implementation specifies the platform-independent remoting mechanisms between Peer Model instances (cf. FR3), it shall allow the development of use cases requiring scalability, i.e., the collaboration of two or more instances (cf. NFR1). Nevertheless, the solution also needs to scale regarding resource consumption within one instance.

Simple API / Usability (NFR5). To allow a smooth development of use cases (cf. FR4), especially the API for services and entries needs to be kept simple. Furthermore, the developers of future work extensions (cf. FR1) shall be able to understand the whole API, not limited to the interfaces between the modules (cf. NFR2, NFR3).

Performance (NFR6). While the primary focus of the implementation will be on enabling remoting (cf. FR3), reasonable performance of the implementation will nevertheless be expected.

CHAPTER 5

Communication and Serialization

Distributed systems, like the Peer Model, necessarily require the ability to communicate between two instances/runtimes running either on the same host, in the same network or in another network. Furthermore, it should not be relevant whether the remote Peer Model instance is developed in the same programming language or a different one.

Section 5.1 gives an introduction to Peer Model Remoting by evaluating possible options for addressing and discovery. The used addressing mechanisms are described in Section 5.2. Next, in Section 5.3, the used discovery mechanism is described. Both decisions are based on the evaluation results. Section 5.4 proceeds with introducing possible serialization formats, while Section 5.5 handles communication formats. Based on these two enumerations, Section 5.7 decides on the serialization and communication formats to be used by evaluating them in respect of the requirements on the formats. Apache Thrift, which is the format to be used, is described in depth in Section 5.8.

5.1 Remoting Overview

The primal step to allow any communication between Peer Model instances is to achieve knowledge of other instances. Addressing can be implemented in two forms: Peer Model instances may either be addressed by using a name or may be addressed by the IP address and port on which the Peer Model is running. Earlier diploma theses have not been consistent at that point: *Csuk* [21] used a naming scheme for addressing instances, which is basically taken over in this thesis. *Rauch* [52] and *Schermann* [53] – on the other hand – used a URI to address remote Peer Model instances. Both options have their benefits and disadvantages.

In the next subsections, a sample network is shown, where the reachable hosts are colored according to the legend shown in Figure 5.1.

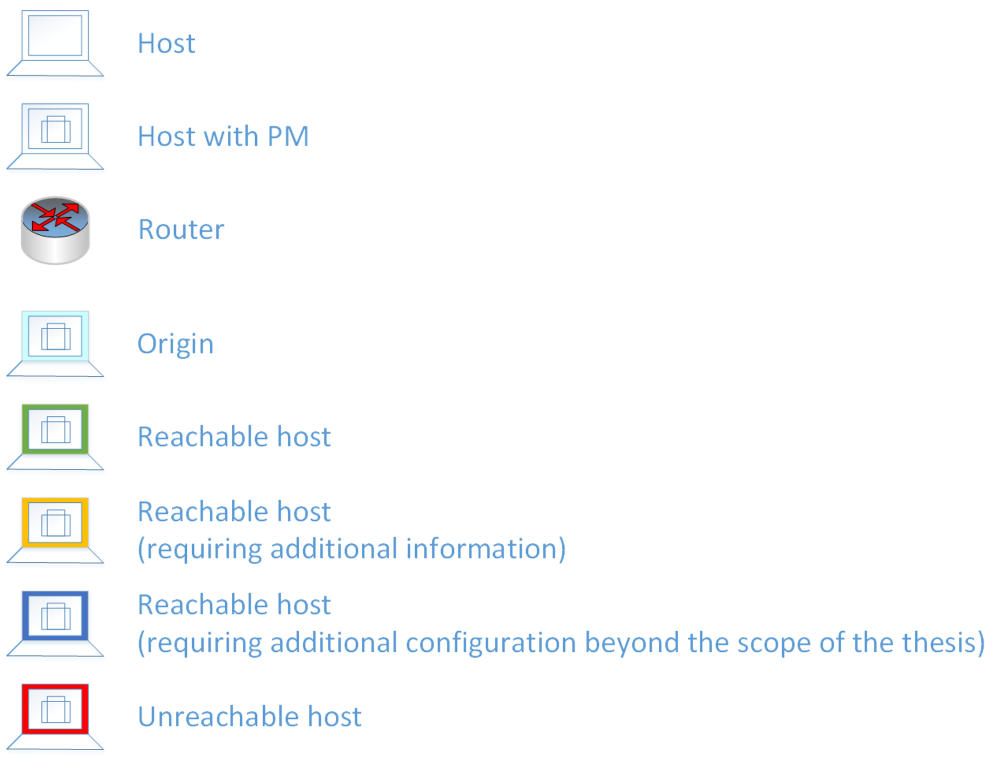

Host
Host with PM
Router
Origin
Reachable host
Reachable host
(requiring additional information)
Reachable host
(requiring additional configuration beyond the scope of the thesis)
Unreachable host

Figure 5.1: Network Legend

5.1.1 URI Addressing Approach

Figure 5.2 shows that URI addressing allows the *origin* instance in the rightmost network to reach all hosts. The main advantage is that URI addressing enables enterprise (inter-network) communication; thus Peer Model instances in the WAN could be reached, too. On the contrary, knowledge of the IP address and the port on which each Peer Model instance is running and reachable is required.

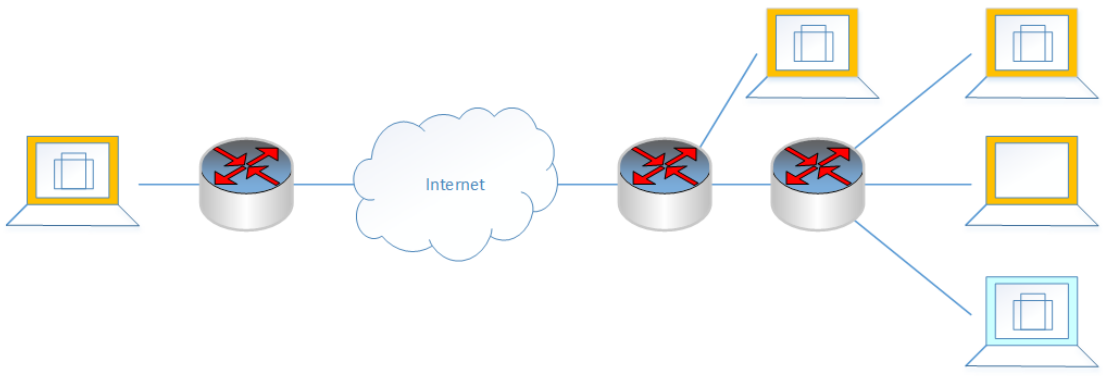


Figure 5.2: Reachable hosts by using URI addressing

5.1.2 Naming Approach

By using the naming approach, the Peer Model instance is addressed by a (in the best case unique) name. Basically, there are four options to acquire knowledge of other instances:

- Local list of known instances
- Centralized server
- Broadcast discovery
- Multicast discovery

Local list of known instances

In this option, the instance holds a list of instances to communicate with. It thus shows no benefits to URI addressing, as the instances to communicate with are required to be known in advance. As such it provides a mapping to URI only, making the solution obsolete. Scalability would be limited.

Central server

Using this option, each instance that comes online connects to a central server on start-up. A login on this server would be sufficient for proper functionality; however, the dependency of all instances on one system limits scalability. The centralized server may not be able to answer a high number of requests once the number of instances in the system becomes high; hence it could become a bottleneck in the system. Geographically scaled systems may be slowed down when requiring a long distance request. Furthermore, if a non-replicated server fails the whole system will not be working properly anymore; thus making the server a single point of failure.

Broadcast discovery

As Figure 5.3 shows, only hosts in the local network can be reached by using broadcast discovery. Broadcast messages are delivered to all hosts in the network, including the ones where no instance is running. However, by using this technique, all instances in the reachable network could be found.

Multicast discovery

Multicast discovery is an extension of the broadcast discovery approach. Using this technique, interested hosts can join a multicast group having a particular IP address in (former) class D networks. In this case, all hosts running a Peer Model instance register themselves to a defined multicast address, where discovery messages are transmitted. This approach is for example used by Hazelcast for the discovery of nodes [4] and thus also within vert.x/Gridlink [17]. As Figure 5.4 shows, basically only hosts in the local

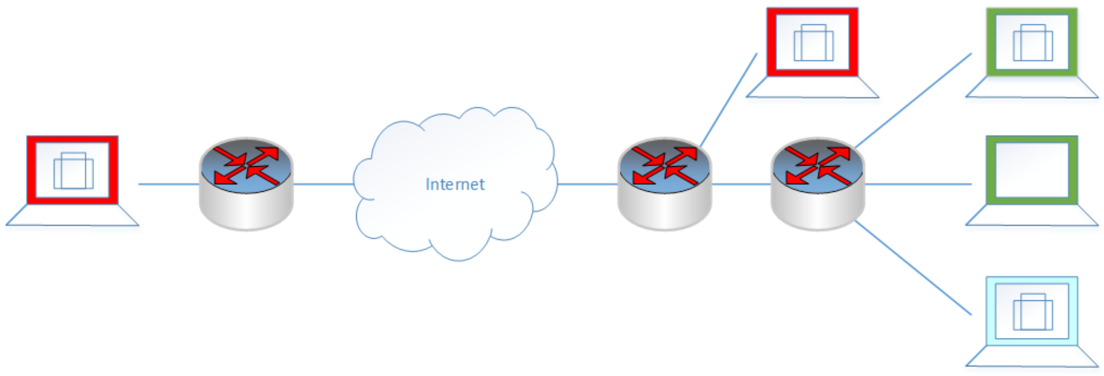


Figure 5.3: Reachable Hosts by using Naming and Broadcast Discovery

network can be reached by using multicast discovery. While broadcast messages will not be transmitted over a router into another network, local routers may be configured to transmit multicast messages into other networks. This configuration is out of scope of this thesis; thus it is assumed that all instances are part of the same network segment. As with broadcast discovery, all instances in the reachable network can be found by using this technique. As only the machines hosting an instance have joined the multicast group, hosts in the reachable network where no instance is running do not receive the message.

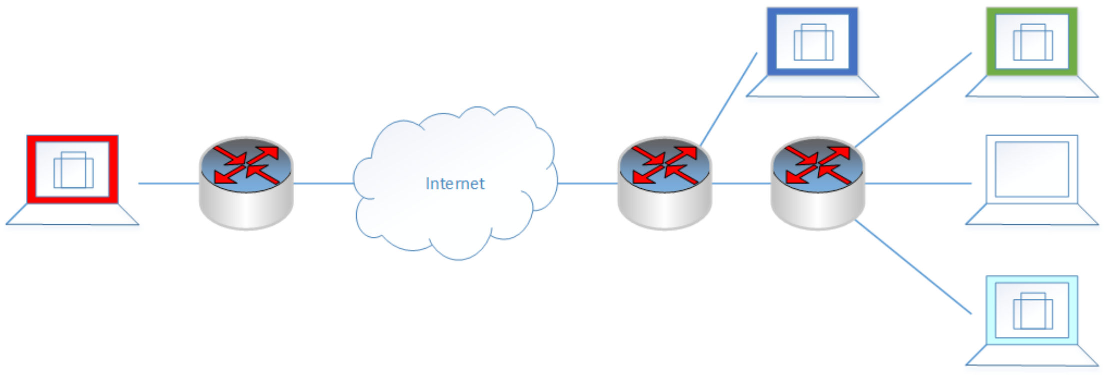


Figure 5.4: Reachable Hosts by using Naming and Multicast Discovery

The main advantage of the naming approach is that prior knowledge of the location (i.e., the URI) where an instance is running is not necessary, the instance's name is sufficient. On the contrary, a discovery mechanism for Peer Model instances is required, but achievable only in local networks.

5.1.3 Results

In result, naming is the best solution for a local network. However, for enterprise solutions the communication may exceed the local network barriers which necessarily requires

URI addressing (cf. Akka reference naming of actors). In consequence, the best suitable solution would combine both options for addressing a Peer Model instance.

This work will follow the multicast approach only. An extension to use URI addressing for remote networks remains as future work.

5.2 Peer Model Addressing

This section first introduces addressing of a peer (Section 5.2.1), and next extends the naming to peer's sub-components (Section 5.2.2).

5.2.1 Addressing of a Peer

In all previous Peer Model implementations, addresses are built by combining the instance address with the peer name. For the Mobile Peer Model [54, 58], for example, the address is build as `<protocol>://<host-name>/<peer-name>`. This addressing schema needs to be mapped manually to a valid Jabber ID (JID) of the form `name@domain/resource` by use of a registration service. This JID is used for addressing in XMPP (eXtensible Message and Presence Protocol), which is the remoting mechanism in the Mobile Peer Model implementation.

Each Peer Model instance has a name, used as destination address for the transmission of entries to the intended target instance. *Csuk* introduced a naming schema for referencing peers and their containers in his thesis [21]. The approach used in this thesis stays very similar to the well-known URI addressing scheme; thus it differs slightly regarding the delimiters between the components in contrast to the invented addressing schema in *Csuk*'s work.

At first, it is required to address the instance's name. The syntax for a (remote) *PeerReference* is as shown in Listing 5.1. Providing a significant name for an instance is among the duties of the developer who is also responsible for providing unique names among running instances.

```
<PeerReference>::=<InstanceName>/<PeerName>
```

Listing 5.1: Peer Reference

A special case of a peer is the **runtime peer**, referenced to (e.g., in logging messages) as "∼". It is automatically available when an instance is started and is the parent peer of all "first-level" peers.

5.2.2 Addressing of Sub-Components

References to peers' sub-components are achieved by using their name in conjunction with the peer's name. Sub-components in this sense are limited to those that require to be addressable, which are the sub-peers and the containers.

Sub-Peers. The conjunction of a (parent) peer reference with a sub peer name constructs another peer reference. References to sub-peers are built as shown in Listing 5.2. It is evident that the levels of sub-peers are therefore not restricted to one, i.e., a sub-peer, in turn, can contain its own sub-peers.

```
<PeerReference>::=<PeerReference>/<SubPeerName>
```

Listing 5.2: Sub-Peer Reference

Containers. A reference to a container is shown in Listing 5.3.

```
<ContainerReference>::=<PeerReference>:<ContainerName>
```

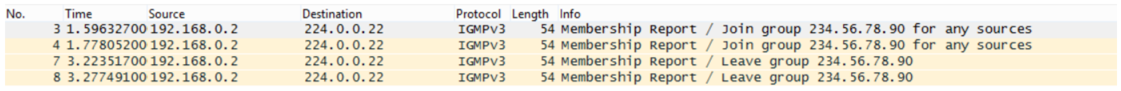
Listing 5.3: Container Reference

5.3 Peer Model Instance Discovery in Local Networks

The discovery of other Peer Model instances is achieved by using multicasting. Nevertheless, each instance holds a list of all other instances it currently knows.

5.3.1 Internet Group Management Protocol (IGMP)

When starting a new Peer Model instance, the host will join a multicast group. In Figure 5.5 a part of a Wireshark sniffing analysis is printed in which an instance is started and terminated shortly after. It shows that there are IGMP messages transmitted in the network to inform others about hosts joining and leaving the group.



```
No.     Time        Source         Destination   Protocol Length Info
      3 1.59632700 192.168.0.2    224.0.0.22    IGMPv3      54 Membership Report / Join group 234.56.78.90 for any sources
      4 1.77805200 192.168.0.2    224.0.0.22    IGMPv3      54 Membership Report / Join group 234.56.78.90 for any sources
      7 3.22351700 192.168.0.2    224.0.0.22    IGMPv3      54 Membership Report / Leave group 234.56.78.90
      8 3.27749100 192.168.0.2    224.0.0.22    IGMPv3      54 Membership Report / Leave group 234.56.78.90
```

Figure 5.5: IGMPv3 Multicast messages

For IPv4 systems (hosts and routers), the Internet Group Management Protocol (IGMP) is used to report IP multicast group membership, the most recent version IGMPv3 being defined in RFC 3376 [14]. RFC 1112, which specified the first version of IGMP, assigned IP addresses with "1110" as their high-order four bits (formerly called class D IP addresses) to host groups [23], i.e., IP addresses in the range of 224.0.0.0 to 239.255.255.255 are used. IPv6's equivalent is Multicast Listener Discovery (MLD) embedded in ICMPv6 [22]; its second version is specified in RFC 3810 [59], which implements comparable functionality to IGMPv3.

It is a straightforward question why these messages cannot directly be used for the discovery of other Peer Model instances, which would make a (more complicated) discovery protocol unnecessary. However, these messages are intended for routers only; they may not be transmitted to the local host. Even if they are received at the local host, they can neither be received in Java code nor can Java be informed of the reception of such messages. This results from the position in the IP stack as shown in Figure 5.6, where IGMP is located on the network layer, while the Peer Model Discovery Protocol (PMDP; see next subsection) is located on the application layer.

Application	PMDP
Transport	TCP/UDP
Network	IP/IGMP
Link	Ethernet

Figure 5.6: IP Stack

5.3.2 Peer Model Discovery Protocol (PMDP)

This subsection describes the basics of the discovery protocol. The Peer Model Discovery Protocol (PMDP) allows the discovery of instances in all languages that will eventually implement this protocol. It follows the multicast approach that was introduced in Section 5.1.2.

Instance Join

Peer Model instances that come online join the multicast group via IGMP Join on an IPv4 multicast address. Over this channel, the joining instance sends a message to all group members containing the name of the instance, its IP address and the TCP port on which it is reachable. Conceptually, there is no difference between this join message and a heartbeat message that will be required at a later point in the protocol. Therefore we will term those *PMHeartbeat* messages. The result of this step is that all instances that were able to receive this message know the new instance. This procedure allows the detection of all instances that come online after the current instance. To simplify the further description, an instance that is already up and running is termed A, while we call the new joining instance B. For simplification, we assume that A is the only instance that is up and B is the second instance joining. Figure 5.7 shows the detection of a new instance at other instances.

Instance List Initialization

There are three options to consider regarding the reply to B's initial *PMHeartbeat* message to allow detection of instances that have already been online before. This message, in turn, contains the instance's name, IP address, and port.

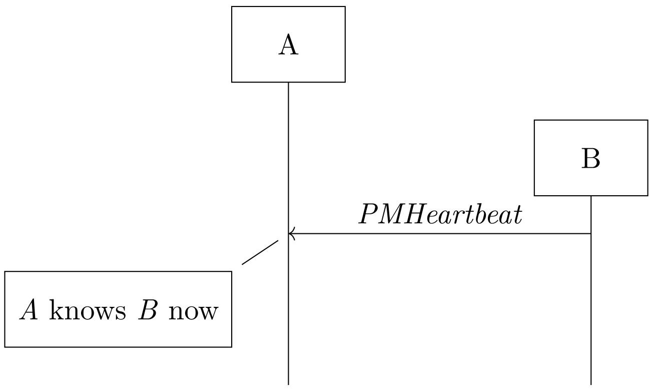


Figure 5.7: Detection of a joining instance

1. *A* replies with a *PMHeartbeat* message to the multicast group. *B* itself does not yet know *A* and will thus reply with an own *PMHeartbeat* message to the multicast channel. This message, however, is not necessary as *B* is already known to the others since they have already received and replied to *B*. This procedure is shown in Figure 5.8.

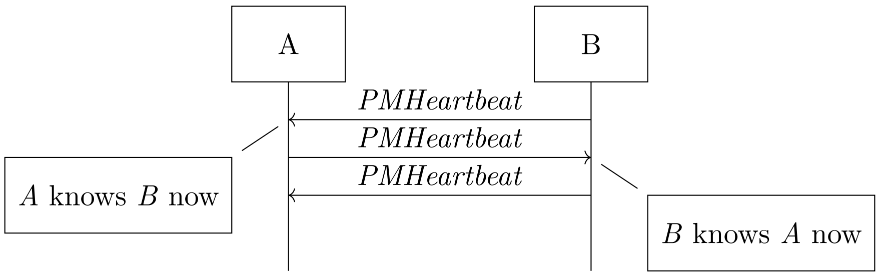


Figure 5.8: Option 1 for the detection of an existing instance

2. *A* replies with a *PMWelcome* message to the multicast group. This message is received at every multicast's member and is basically the same as the *PMHeartbeat* message. The difference to option 1 is that *B* is not required to reply with another message. Due to receiving the *PMWelcome* message, it is obvious for *B* that it is already known there. This procedure is shown in Figure 5.9.

   Problems could arise if two instances *C* and *D* join approximately at the same time. In this case, a *PMWelcome* message from *A*, actually replying to *C*, is received by *D* on the multicast address. In that case, *D* would wrongfully expect to be already known at instance *A*. However, every new joining instance always sends a *PMHeartbeat* message on its join, therefore eventually *D* will be known to the other instances.

3. *A* replies with a *PMHeartbeat* message directly per unicast to *B*. In this case, a *PMWelcome* message type is not necessary. *B* receives the message on its individual

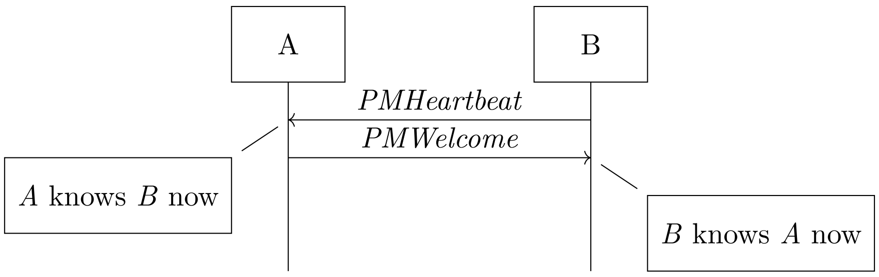


Figure 5.9: Option 2 for the detection of an existing instance

address and must hence already be known to A. Therefore, it is evident which instance was meant; thus it is also not necessary to reply with any further message. This procedure is shown in Figure 5.10.

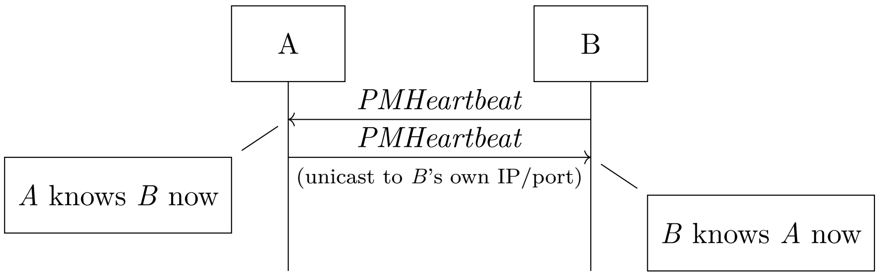


Figure 5.10: Option 3 for the detection of an existing instance

Option 3 requires a second channel (for unicast messaging) for discovery, so option 2 was chosen for the implementation.

Heartbeat

PMHeartbeat messages are sent to the multicast group periodically as long as the instance stays alive. The name of the instance is contained in the *PMHeartbeat* message together with the port where this instance is running, having the host IP address as the source address of the UDP packet.

Instance leave

Instances may leave at a later time, requiring their removal from the list of instances. It is unnecessary that they are saved once they are no longer reachable due to two considerations: First, all instances require a certain albeit small amount of storage; however, the effort required to find an instance in the cache would increase unnecessarily. Second, those instances would remain as orphans in the cache; hence the Peer Model instance would still believe that they are reachable and would try to deliver entries to the assigned destination. Therefore, instances are only temporarily saved in a cache.

Planned leave An instance intending to leave (i.e., planned shutdown) sends a *PM-Leave* message on the multicast channel, informing that it is about to retire such that the other instances can react appropriately. In result, they remove the instance from their cache. The described procedure is accomplished before the instance's host leaves the multicast group by use of an IGMP Leave command. This procedure is shown in Figure 5.11.

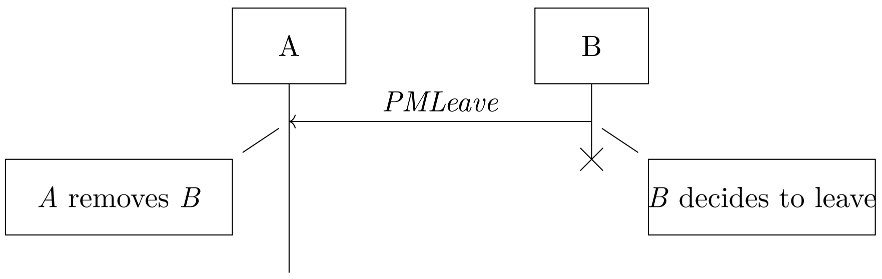


Figure 5.11: Planned leave of an instance

Unplanned leave If the instance fails to unsubscribe either because the instance left silently (e.g., is killed) or the host is no longer reachable, neither the *PMLeave* nor the *IGMP Leave* messages are sent. Such an orphan instance needs to be removed from the cache after some time. This method requires, however, to periodically inform other instances as long as the instance is still alive. Therefore, each instance periodically sends the *PMHeartbeat* message on the multicast channel, allowing all attached instances to renew the entry in their cache. After a given time interval in which *PMHeartbeat* messages are no longer received, each instance removes the information of the left instance in its cache. The procedure is shown in Figure 5.12.

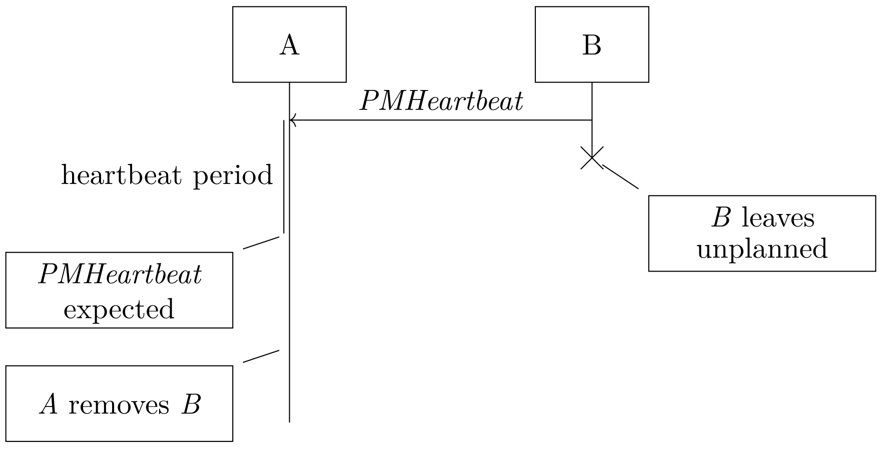


Figure 5.12: Unplanned leave of an instance

Instances could be only temporarily unreachable. In that case, the following three options are possible when the instance reappears:

1. If the remote instance reappears while the threshold time has not yet expired, it is still contained in the local instance's cache. The procedure follows as usual with a *PMHeartbeat* message.

2. If the remote instance reappears while the threshold time has not yet expired, but its IP address and/or port have changed, it is still contained in the local instance's cache. This procedure will be described in the next paragraph (Instance Information Change).

3. If the remote instance reappears after the threshold time has expired, it is no longer contained in the local instance's cache. Therefore, the remote instance is considered to be a new joining instance and added to the cache as usual.

Instance Information Change

If the IP address or the port of a named instance changes, the following *PMHeartbeat* message will contain the new information. Thus, all instances will update the information in their caches. As the new instance sends new information then, consequently no heartbeat messages containing the old information will occur later on in this case.

The requirement of a unique instance name

Problems occur if the same name is used for multiple instances. As described, an instance that is already known is temporarily saved in the other instances' caches as long as heartbeat messages are received periodically. If a new instance appears online, while another instance sharing the name is online, both instances periodically send their *PMHeartbeat* messages. For all other instances, this seems like one instance constantly changing its location. Therefore, if an entry has the ambiguous destination set, it will be transmitted to the instance currently saved in the sender instance's cache. Note that the implementation of one of the following addressed solutions remains as future work.

1. One possible solution to the addressed problem is the temporary removal of the instance on the reception of a heartbeat message containing data that do not conform to already known instance data. The new data is remembered and changed at the cache at the next heartbeat that contains the same data. If the old data is received again between the first and the second heartbeat, it is detected that the instance name is ambiguous, which would require to take further actions.

2. A similar approach is to request data from the changed source instance once the first heartbeat message was received. If only one instance replies it can be considered to be *the* instance; otherwise, it is detected that the instance name is ambiguous, which would require to take further actions.

3. Another solution would be that the one instance that receives a message containing its own name sends a denial message on the multicast channel. Other instances are

required to take further actions. In consequence, it would be required to extend the discovery protocol.

Other approaches

The task of discovering other instances is not unique to the Peer Model but required in almost every kind of distributed system. Indeed, there exist other similar or completely different approaches to identify other instances in a network.

A similar approach is for example used by Hazelcast for forming a cluster [4]. While the described procedure uses UDP multicast connections only, a primary difference is that Hazelcast switches to TCP/IP communication for heartbeat messages after the instances have discovered each other. As TCP provides point-to-point connections, no multicast messages are possible. Naturally, heartbeat messages are required to be sent by unicast to each instance individually using this approach.

5.4 Serialization formats

This section describes the decision for a proper and efficient format to serialize data for communication between Peer Model instances. It is of high relevance that communication between Peer model instances is possible regardless of the programming language the instance is developed in. Required is, therefore, a format that neither restricts the instances to be in the same programming language nor improperly restricts the supported programming languages. Peer Model implementations already exist in C [29] and C# [52]; however, it will not be possible to communicate with them using the developed technique without additional implementation effort in these implementations.

Serialization is the term for the process of encoding an object to a stream, which can then be sent through a socket or stored in a file; the reverse process is known as *deserialization*. In the use case, entries that should be transmitted to another instance need to be translated from object representation to a serialization format. We enumerate possible formats for serialization and communication, which are either text-based or a binary format. For each format in question, samples that are equivalent in respect of data that are to be persisted are provided. Due to the broad knowledge of the enumerated text-based formats, they are not required to be introduced in depth. A basic comparison between text-based and binary-based formats can be found in Table 5.1.

5.4.1 Text-based formats

The advantage of text-based formats is their human readability. They also tend to be more platform-independent than binary formats. However, the documents' size is much larger, and the serialization/deserialization procedure usually takes more time, making text-based formats less efficient than binary formats. XML and JSON are formats which are used most and supported best. YAML as a third representative will be described

	Text-based	**Binary-based**
Human-readable	yes	no
Size	bigger	smaller
Performance	usually slower	usually faster
Platform-independent	usually	depends on format
Examples considered	JSON XML YAML	Java Serialization Protocol Buffers Apache Thrift Apache Avro

Table 5.1: Comparison of formats

only briefly due to little tool support. Discussions whether XML or JSON is better span across the Internet, often biased by use case requirements and personal favor.

XML

XML (eXtensible Markup Language) is not just a format to exchange data, but as the name says a markup language. As such it is more versatile than JSON, whose only purpose is to exchange data. It additionally provides the ability to specify a domain-specific XML Schema (XSD) [26] to tell valid and invalid XML files apart. XML, currently in version 1.1, is a W3C specification [55]. The tag set is not predefined by XML itself but specified by the user or the XSD if available.

Design goals include [55]:

- *XML shall be straightforwardly usable over the Internet.*
- *XML shall support a wide variety of applications.*
- *It shall be easy to write programs which process XML documents.*
- *XML documents should be human-legible and reasonably clear.*
- *XML documents shall be easy to create.*

Several methods for processing and writing XML are directly included into the Java Standard Edition (Java SE). In earlier versions, only the tree-based DOM API (Document Object Model) and the event-based SAX API (Simple API for XML) were supported. DOM, where accesses to the XML document happen over an object tree, can write and read XML files. SAX is suitable for the sequential reading of XML documents only. Since Java 6, cursor-based StAX (Streaming API for XML) is included that handles the disadvantages of both above mentioned options. Thus it neither requires to hold the whole document in the storage like DOM nor is it only able to parse the XML document sequentially like SAX. Java Architecture for XML Binding (JAXB) 2.0 (JSR 222[1]) goes

[1]`http://www.jcp.org/en/jsr/detail?id=222`

one step further by defining the mapping of Java objects to XML and vice versa. It is also included in the JDK since Java 6. An example XML representation of a Peer Model entry is shown in Listing 5.4, the according XML Schema (XSD) is printed in Listing 5.5.

```
<entry type="E" id="1" tts="1000" ttl="5000" destination="P1:PIC" flowId="1">
    <coData>
        <coDataEntry key="coKey1" value="coVal1" />
        <coDataEntry key="coKey2" value="coVal2" />
        ...
    </coData>
    <appData>
        <appDataEntry key="appKey1" value="appVal1" />
        <appDataEntry key="appKey2" value="appVal2" />
        ...
    </appData>
</entry>
```

Listing 5.4: Example Entry in XML

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="entry" type="entryType"/>
  <xs:complexType name="entryType">
    <xs:attribute type="xs:string" name="type" use="required"/>
    <xs:attribute type="xs:string" name="id" use="required"/>
    <xs:attribute type="xs:short" name="tts" use="optional"/>
    <xs:attribute type="xs:short" name="ttl" use="optional"/>
    <xs:attribute type="xs:string" name="destination" use="required"/>
    <xs:attribute type="xs:string" name="flowId" use="optional"/>
    <xs:sequence>
      <xs:element type="coDataType" name="coData"/>
      <xs:element type="appDataType" name="appData"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="coDataType">
    <xs:sequence>
      <xs:element type="coDataEntryType" name="coDataEntry"
        maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="coDataEntryType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="key" use="required"/>
        <xs:attribute type="xs:string" name="value" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="appDataType">
    <xs:sequence>
      <xs:element type="appDataEntryType" name="appDataEntry"
        maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="appDataEntryType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="key" use="required"/>
        <xs:attribute type="xs:string" name="value" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Listing 5.5: According XML Schema

JSON

JSON, short for JavaScript Object Notation, is a human-readable text-based format for data exchange between applications. Essentially, it is language-independently writable and parsable; however, it has always been neglected by Java's official API. Even the current Java 10 release does not include a JSON API. This currently leads to support in Java via third-party libraries only (e.g., Jackson parser[2]). The importance of (first-level) JSON support is addressed by the motivation statement in JEP 198 that was earlier scheduled to include JSON support into Java 9 SE[3]:

> *JSON has become the lingua franca for web services and it is time for Java SE to adopt functionality for interacting with and utilizing JSON documents and data streams.*

Nevertheless, Mark Reinhold, chief architect of the Java Platform Group at Oracle confirmed in December 2014 to the Java mailing group that JSON support in Java 9 was dropped[4]:

> *This JEP would be a useful addition to the platform but, in the grand scheme of things, it's not as important as the other features that Oracle is funding, or considering funding, for JDK 9. [...] We may reconsider this JEP for JDK 10 or a later release [...]*

A Java Specification Request (JSR 353: Java API for JSON Processing, known as JSON-P[5]) with a reference implementation[6] is available. It defines a standard API for parsing and generating JSON data[7]. The intended tasks are to

> 1. *produce and consume JSON text in a streaming fashion (i.e., similar to StAX API for XML)*
> 2. *build a Java object model for JSON text using API classes (i.e., similar to DOM API for XML)*

[2]`https://github.com/FasterXML/jackson`
[3]`http://openjdk.java.net/jeps/198`
[4]`http://mail.openjdk.java.net/pipermail/jdk9-dev/2014-December/001670.html`
[5]`http://www.jcp.org/en/jsr/detail?id=353`
[6]`https://jsonp.java.net/`
[7]`http://www.oracle.com/technetwork/articles/java/json-1973242.html`

Besides this, there is also a Java Specification Request (JSR 367: Java API for JSON Binding (JSON-B)[8]) for a standardized way to convert JSON into Java objects and vice versa.

JSON is now standardized by the Internet Engineering Task Force (IETF) in RFC 7159 [13] and ECMA-404 [3]. Comparable to XML Schema, there also exists a draft for JSON schema by the IETF [25], providing the schema as valid JSON. In contrast to XML, JSON is more compact and often considered to be more readable due to its simpler syntax. Evaluations showed that JSON is faster than XML concerning parsing. An example JSON representation of a Peer Model entry is shown in Listing 5.6, the according JSON Schema is printed in Listing 5.7.

```
{
    "entryType" : "E",
    "entryId" : "1",
    "tts" : 1000,
    "ttl" : 5000,
    "destination" : "P1:PIC",
    "flowId" : "1",
    "coData" : [
        {"key" : "coKey1", "value" : "coVal1"},
        {"key" : "coKey2", "value" : "coVal2"},
        ...
    ],
    "appData" : [
        {"key" : "appKey1", "value" : "appVal1"},
        {"key" : "appKey2", "value" : "appVal2"},
        ...
    ]
}
```

Listing 5.6: Example Entry in JSON

```
{
  "$id": "http://example.com/example.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "entryType": {
      "$id": "/properties/entryType",
      "type": "string",
      "title": "The Entrytype Schema"
    },
    "entryId": {
      "$id": "/properties/entryId",
      "type": "string",
      "title": "The Entryid Schema"
    },
    "tts": {
      "$id": "/properties/tts",
      "type": "integer",
      "title": "The Tts Schema"
    },
    "ttl": {
      "$id": "/properties/ttl",
```

[8]http://www.jcp.org/en/jsr/detail?id=367

```
      "type": "integer",
      "title": "The Ttl Schema"
    },
    "destination": {
      "$id": "/properties/destination",
      "type": "string",
      "title": "The Destination Schema"
    },
    "flowId": {
      "$id": "/properties/flowId",
      "type": "string",
      "title": "The Flowid Schema"
    },
    "coData": {
      "$id": "/properties/coData",
      "type": "array",
      "items": {
        "$id": "/properties/coData/items",
        "type": "object",
        "properties": {
          "key": {
            "$id": "/properties/coData/items/properties/key",
            "type": "string",
            "title": "The Key Schema"
          },
          "value": {
            "$id": "/properties/coData/items/properties/value",
            "type": "string",
            "title": "The Value Schema"
          }
        },
        "required": [
          "key",
          "value"
        ]
      }
    },
    "appData": {
      "$id": "/properties/appData",
      "type": "array",
      "items": {
        "$id": "/properties/appData/items",
        "type": "object",
        "properties": {
          "key": {
            "$id": "/properties/appData/items/properties/key",
            "type": "string",
            "title": "The Key Schema"
          },
          "value": {
            "$id": "/properties/appData/items/properties/value",
            "type": "string",
            "title": "The Value Schema"
          }
        },
        "required": [
          "key",
          "value"
        ]
      }
    }
  },
  "required": [
    "entryType",
    "entryId",
    "destination"
  ]
}
```

Listing 5.7: According JSON Schema

YAML

YAML originally stands for "Yet Another Markup Language", but now serves as the recursive acronym "YAML Ain't Markup Language". As the example in Listing 5.8 shows, its simplicity excels even JSON. However, it suffers from small tool support and is also less known than XML and JSON. In Java environments, YAML can be parsed by use of an additional library to the well-known JSON parser Jackson. An example YAML representation of a Peer Model entry is shown in Listing 5.8; a schema is not available.

```
---
entryType : "E"
entryId : "1"
tts : 1000
ttl : 5000
destination : "P1:PIC"
flowId : "1"
coData :
- key : "coKey1"
  value : "coVal1"
- key : "coKey2"
  value : "coVal2"
...
appData :
- key : "appKey1"
  value : "appVal1"
- key : "appKey2"
  value : "appVal2"
...
```

Listing 5.8: Example Entry in YAML

5.4.2 Binary formats

Binary formats tend to be more efficient than text-based formats; they are however not human-readable. In this section, we evaluate the built-in Java serialization, Protocol Buffers, Apache Thrift and Apache Avro regarding the use case's requirements. It is shown that not all binary formats in question are platform-independent and thus cannot fulfill an essential requirement.

For most of the evaluated binary formats, a language-independent schema file needs to be provided of which classes are generated by the compiler. These generated classes can then be used within the application for population, serialization (marshaling), deserialization (unmarshaling), and access of messages. Serialized data can be written to a file or transmitted over the network, and received at or read in by a program written in another language; thus allowing platform independency. While text-based formats are self-describing, these binary formats stay meaningful only when having the message definition (schema) available.

Java Serialization

Java Serialization is the built-in ability of Java to persist its objects, reload and reuse them again. It is read- and writable by Java only, thus no platform independency is possible by using this format. Essentially, a Java object is serialized into a byte array that may either be persisted to disk or sent via the network. On the recipient side or when reading the persisted file on the disk, it is deserialized again to a Java object. Steps like reading all members of the object, persisting them as text in the output format, as well as creating a new object based on what is read in are not required by the developer. Thus, this procedure is by far the most convenient method as the programmer usually does not need much effort to achieve serialization/deserialization functionality.

To define Java objects as being serializable, they are required to implement the `Serializable` interface. All (non-transient) fields of the class in turn have to be serializable. In essence, the intention is to freeze the object graph, persist it or move the representation across the network and then restore the graph back into usable Java objects [46]. Primitives and most heavily used members of the Java API are serializable, as for example strings, numbers or collections. It is possible to extend standard serialization or implement a custom serialized form in places where the standard serialization algorithm would fail due to non-serializable members or where the use of the standard algorithm is inappropriate [11]. Fields that should not be serialized need to be marked with the `transient` keyword. At deserialization these fields get initialized with their default value (e.g. null for object references), therefore it may be required to provide a custom implementation to initialize these fields.

The example result for Java Serialization is binary, thus not human-readable. The according example schema for a Peer Model entry is printed in Listing 5.9.

```
class Entry implements Serializable {
   String type; // required
   String id; // required
   Integer tts; // optional
   Integer ttl; // optional
   String destination; // required
   String flowId; // optional
   CoData[] coData;
   AppData[] appData;

   // setter/getter methods
}

class CoData implements Serializable {
   String key; // required
   String value; // required

   // setter/getter methods
}

class AppData implements Serializable {
   String key; // required
   String value; // required

   // setter/getter methods
}
```

Listing 5.9: Example with Java Serializable

Protocol Buffers

Protocol Buffers[9] were initially developed by Google to be used among programs developed in different languages offering a language-neutral, platform-neutral, and extensible mechanism for serializing structured data [48]. Protocol Buffers claim [1] that in comparison to XML they

1. are simpler.
2. are 3 to 10 times smaller.
3. are 20 to 100 times faster.
4. are less ambiguous.
5. generate data access classes that are easier to use programmatically.

Protocol Buffers are heavily used by its inventor [1]:

> *Protocol buffers are now Google's lingua franca for data —at time of writing, there are 48,162 different message types defined in the Google code tree across 12,183 .proto files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems.*

Fields are uniquely identified by their tag. In the message definition they may either be specified as *required*, *optional* or *repeated*, which allows array- or list-like fields. Official support by Google is provided for Objective C, C++, C#, Go, Java, Python and Ruby. Furthermore, a large number of third-party libraries exist for other languages, currently listing about 30 additional languages[10]. The target language is specified by a flag at execution of the Protocol Buffers compiler. Protocol Buffers are released under a BSD-style license.

The sequence steps to use Protocol Buffers within a project are shown in Figure 5.13; they are similar to the steps that need to be taken in Thrift and Avro. In step 1,

[9] `https://developers.google.com/protocol-buffers/`
[10] `https://github.com/google/protobuf/wiki/Third-Party-Add-ons`

a language-independent schema file for a Peer Model entry is written using a data description language (DDL), which is printed in Listing 5.10. This file is compiled by the *protoc* compiler in step 2 to generate the language-specific code. The generated code is included to the application's code in step 3, making it possible to use it like every other Java code. The example result for a transmitted entry in Protocol Buffers is binary, thus not human-readable.

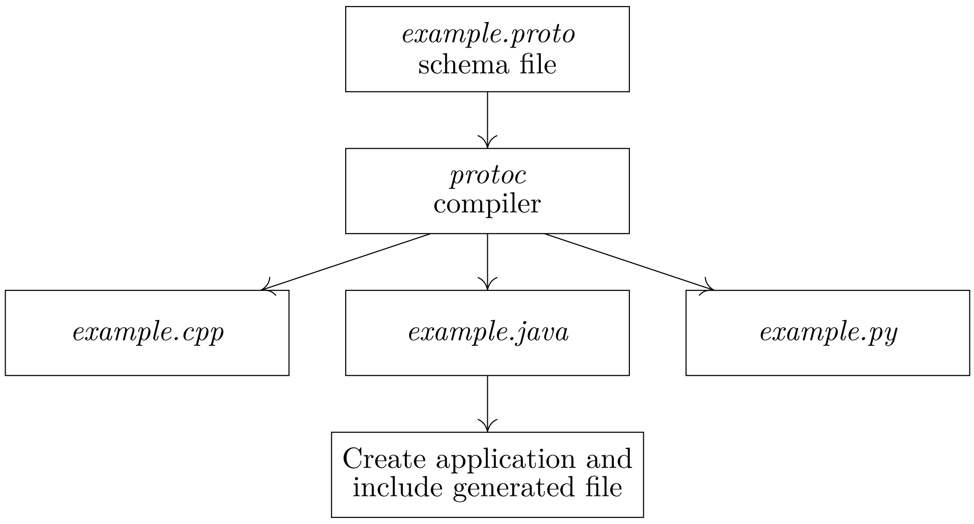


Figure 5.13: Protocol Buffers implementation steps

```
message Entry {
   required string type = 1;
   required string id = 2;
   optional int64 tts = 3;
   optional int64 ttl = 4;
   required string destination = 5;
   optional string flowId = 6;
   repeated CoData coData = 7;
   repeated AppData appData = 8;

   message CoData {
      required string key = 1;
      required string value = 2;
   }

   message AppData {
      required string key = 1;
      required string value = 2;
   }
}
```

Listing 5.10: Example in Protocol Buffers schema

Apache Thrift

Apache Thrift[11] is another option for a binary format with similarities to Protocol Buffers and also having the demand for efficient cross-language data serialization. Its goal is to enable efficient and reliable communication across programming languages [5]. Thrift was initially developed at Facebook in 2006, as they required to integrate functionalities of programs written in different languages. Facebook's culture was to choose the best tools and implementations over standardizing on one programming language [5]. While initially hosted as open-source software directly at Facebook, the project later moved to Apache to foster greater use [49]. It is released under the Apache 2 license. The list of supported target languages is comparable to Protocol Buffers, listing 26 languages[12].

The sequence steps to use Thrift within a project are shown in Figure 5.14; they are similar to the steps that need to be taken in Protocol Buffers and Avro. In step 1, a language-independent schema file for a Peer Model entry is written using a language-neutral Interface Definition Language (IDL), which is printed in Listing 5.11. This file is compiled by the *thrift* compiler in step 2 to generate the language-specific code. The generated code is included to the application's code in step 3, making it possible to use it like every other Java code. Thrift offers not only one protocol as Protocol Buffers and Avro. Among them are also human-readable text-based protocols if required, e.g. for debugging purposes. A list of the supported protocols can be found in Section 5.8.

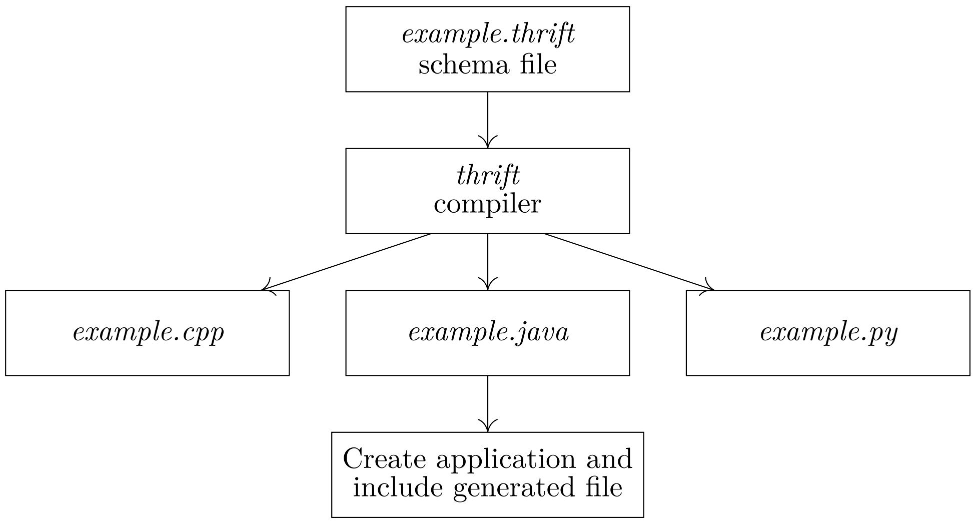


Figure 5.14: Thrift implementation steps

```
struct Entry {
   1: required string type,
   2: required string id,
   3: optional i64 tts,
   4: optional i64 ttl,
```

[11]https://thrift.apache.org

[12]https://thrift.apache.org/docs/Languages

```
    5: required string destination,
    6: optional string flowId,
    7: optional list<CoData> coData,
    8: optional list<AppData> appData
}

struct CoData {
    1: required string key,
    2: required string value
}

struct AppData {
    1: required string key,
    2: required string value
}
```

Listing 5.11: Example in Thrift schema

Apache Avro

Apache Avro[13] is another option that provides a binary format for efficient, cross-language serialization and code generation developed within Apache's Hadoop program. The Avro schema may either be specified in a JSON format or using an IDL. The schema is integrated in the beginning of the stream, which makes sense for a big number of elements following that schema. If only a small amount of data is transmitted with an attached schema, the size to be transmitted is unnecessarily increased. As another option, client and server can thus negotiate on a schema at handshake that is used for the rest of a session. Optional fields are not available in Avro, but they can be substituted by use of a union type with null. Supported languages are C, C++, C#, Java, Perl, Python, Ruby and PHP[14]; thus the list is significantly shorter than in Protocol Buffers and Thrift. Avro is mainly used in Apache Hadoop and is released under the Apache 2 license.

The sequence steps to use Avro within a project are shown in Figure 5.15; they are similar to the steps that need to be taken in Protocol Buffers and Thrift. In step 1, a language-independent schema file for a Peer Model entry is written using the IDL, which is printed in Listing 5.12. This file is compiled by the *avro* compiler in step 2 to generate the language-specific code. The generated code is included to the application's code in step 3, making it possible to use it like every other Java code. The example result for a transmitted entry in Avro is binary, thus not human-readable.

```
{
  "namespace": "avro",
  "type": "record",
  "name": "Entry",
  "fields": [
    {"name": "type", "type": "string"},
    {"name": "id", "type": "string"},
    {"name": "tts", "type": ["null", "long"]},
    {"name": "ttl", "type": ["null", "long"]},
```

[13] https://avro.apache.org

[14] https://cwiki.apache.org/confluence/display/AVRO/Supported+Languages

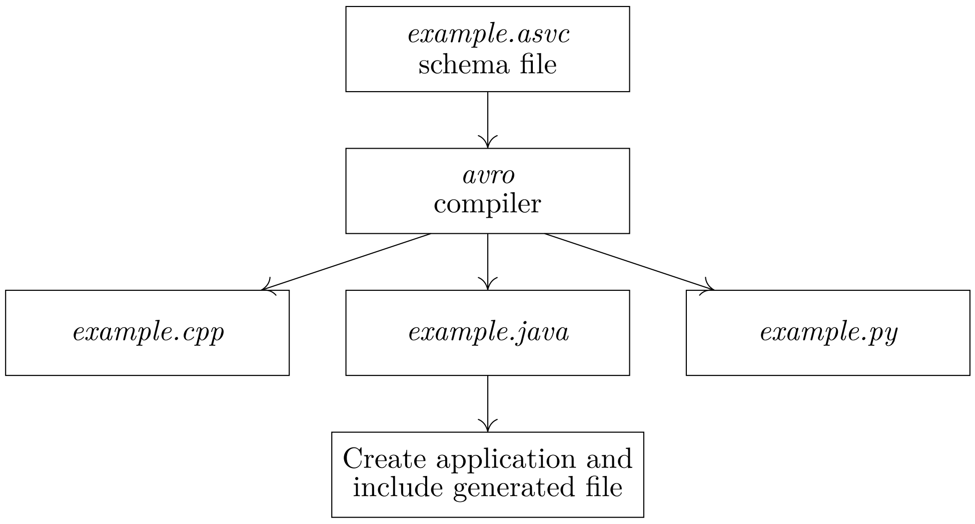


Figure 5.15: Avro implementation steps

```
      {"name": "destination", "type": "string"},
      {"name": "flowId", "type": ["null", "string"]},
      {"name": "coData",
        "type": {
          "type" : "array",
          "items": {
            "name": "CoDataEntry",
            "type": "record",
            "fields" : [
              {"name": "key", "type": "string"},
              {"name": "value", "type": "string"}
            ]
          }
        }
      },
      {"name": "appData",
        "type": {
          "type" : "array",
          "items": {
            "name": "AppDataEntry",
            "type": "record",
            "fields" : [
              {"name": "key", "type": "string"},
              {"name": "value", "type": "string"}
            ]
          }
        }
      }
    ]
}
```

Listing 5.12: Example in Avro schema

Other formats

The evaluated formats are only a small fraction of available formats, however the most common and most suitable formats were listed. Some other formats combine JSON with the advantages of binary formats (e.g. BSON, MessagePack, Smile etc.) and provide support in a majority of current languages. It would have been possible to define an own format, which would be a simple and flexible approach and, as it would be requirement-driven, best suited for the use case. However, it would require to write own encoding and parsing algorithms, although the described more general approaches exist.

5.4.3 Schema Evolution

The effect of an addition or removal of parameters often results in incompatibility of remote interfaces. Schema evolution or versioning is the term for changes in the schema when already in use, i.e. either the adding or the removal of a field in the schema, and how the generated code deals with it. This is important, as it is problematic to change a field in the schema to an optional field later. Readers with knowledge of the old version will consider messages without this field as incomplete and throw an exception in that case. We cannot rule out that in our use case the schema will be changed at a later time.

A blog entry by *Kleppmann* [34] compares the binary formats of Protocol Buffer, Apache Thrift and Apache Avro regarding the support for schema evolution by showing how data is encoded into bytes. In Protocol Buffers, fields are represented by their ordinal number in the schema and not by their name. Thus, field names can be changed without any consequences; but the occurence of the field in the schema (i.e. its tag number) cannot. There is no difference in the encoding whether fields are marked as optional, required, or repeated; for the latter one the number of times the tag number appears in the encoding is simply higher. From the encoding point of view, this aspect can thus be changed without consequences. However, runtime errors will occur if an unavailable field is expected at the receiver. An optional field that has not been set is omitted from the encoded data, thus those fields could safely be removed. Fields can also be added to the schema, but this field should get a tag number that has not been used before. Apache Thrift's schema evolution is likewise, with the exception of repeated fields that are represented by lists there. Encodings of records in Apache Thrift will be shown later on in Figures 5.17 and 5.18. In Apache Avro, the encoding is different as it contains no tag numbers in its schema. By providing Avro the schema with which the data was written (the writer's schema) and the schema the consumer is expecting (the reader's schema), resolution rules are used to translate data from the writer schema into the reader schema. Details can be found in the mentioned blog entry, but will not be given in depth in this thesis.

Thus, five cases can be distinguished [5]:

- **Case 1: Added field, old client, new server**
  In this case the old client does not send a field that was added with a new schema

version. This case is unproblematic if the new field is marked as `optional`. A default value can be chosen that will be used if a value is not provided at transmission. If it is possible that an old client connects to the new server, new fields should always be marked as optional.

- **Case 2: Removed field, old client, new server**
  In this case the old client sends a field that was removed in the new schema version. This case is unproblematic, an unknown field received at the server is ignored.
- **Case 3: Added field, new client, old server**
  In this case the new client sends a field added in the new schema version that cannot be recognized at the server using the old schema version. This case is unproblematic, as the server ignores fields that it does not know.
- **Case 4: Removed field, new client, old server**
  In this case the new client does not send a field that was removed in the new schema version to the server using the old schema version. This case is problematic if the field was `required` in the old version. This is the reason why it is sometimes discouraged to use required fields at all [28].
- **Case 5: Data type changed for field**
  The most problematic case is the one where the data type of an existing field is changed. Deserializationg will always fail as another encoding of the field is expected. Therefore, the ordinal number of a field in the struct shall never be reused. Those numbers can also be omitted in the schema; thus the solution is to use a new ordinal number for the changed field.

5.5 Communication formats

One of the most important goals of distributed systems is *openness* [57], i.e., services of a system are offered according to a standardized interface that describes syntax and semantics. *Tanenbaum/Van Steen* state that a properly specified interface allows two independent parties to build completely different implementations leading to two separate implementations that operate in exactly the same way [57].

Remote Procedure Calls (RPC) are a well known and heavily used technique to achieve inter-process communication first introduced by *Birrell and Nelson* in 1984 [10]. An RPC consists of a client program (caller) and a server program (callee). The client sends a call to the server, which replies. Data that was serialized by using one of the methods described in the previous section are transmitted over the network on this call. They are received by a program executed either as another process on the same machine or on another computer. At the recipient, the data is deserialized, the proper procedure is executed, and data are replied to the original sender, if applicable. In the Peer Model context, the terms client and server only apply to a particular transaction; therefore, it is possible that in the next transmission the roles are inverted. The exchange of entries is the only service required to be offered by the callee. For the required platform independence, it is equally crucial as with serialization formats that the communication formats are supported by as many different programming languages as possible.

The binary formats that were enumerated in the previous section have the handy advantage that they are either already intertwined with an RPC implementation or that an RPC system exists that is suitable for those formats. Consequently, it was concluded to use such an RPC implementation and not consider other possible communication formats such as REST.

Java RMI

Java provides functionality for RPC by Remote Method Invocation (RMI), but communication with programs developed in other languages is not easy to achieve. As RMI can transmit serializable objects only, this option is not usable in any other languages.

gRPC

Protocol Buffers were initially developed to define messages for communication between servers [48]. However, while Google made their Protocol Buffers publicly available already in 2008, they did not provide their internally used RPC system. RPC stubs could automatically be generated if the schema contains a service definition, but for years real RPC functionality could be achieved with third-party libraries only.

During the implementation of the Java Peer Model version, Google launched gRPC (gRPC Remote Procedure Calls)[15]. It is available under Apache 2.0 license. Both synchronous and asynchronous communication is allowed.

Using the *protoc* compiler with a gRPC plugin generates client and server code in the target language as well as the regular Protocol Buffer code. At the moment the supported languages are C++, Java, Objective-C, Python, Ruby, Go, C# and Node.js. An example service definition is printed in Listing 5.13.

```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

Listing 5.13: gRPC service example[16]

[15]http://grpc.io

Apache Thrift

In contrast to the other evaluated binary formats, Apache Thrift includes its own RPC framework. Data types and RPC service interfaces are defined in the same language-neutral Interface Definition Language (IDL) schema file. Thrift provides RPC implementations for both client and server across multiple languages, including asynchronous variants in many languages. The RPC services are generated for the target language by the *thrift* compiler. An example service definition is printed in Listing 5.16 on page 80.

Apache Avro

Comparable to Protocol Buffers, Apache Avro does also not include its own RPC framework but uses the netty server instead.

5.6 Serialization and remoting mechanisms in related frameworks

As serialization and remoting is required in every distributed system to transmit messages between its actors, first the mechanisms used by the frameworks and earlier Peer Model implementations of the related work (Chapter 2) shall briefly be mentioned. In Akka, for serialization either Java's built-in serialization or Protocol Buffers can be used [44]. vert.x does not specify any serialization format to use, in Gridlink by default JSON is used [16]. WS-BPEL transmits all messages in the WSDL format in XML. In Gigaspaces XAP, by default the Java serialization is used; other serialization mechanisms such as the Kryo serializer[17] can be plugged in. The Embedded PeerSpace uses an own binary stream implementation [29], for the PeerSpace.NET the internal format of its remoting mechanism Xcoordination Application Space is used [52]. The Mobile Peer Model allows to either use a human-readable JSON, or a non-readable binary Protocol Buffers format [54]. It is – however – explicitly mentioned that it would be easy to exchange the format to use Apache Thrift.

For remoting, both Akka and vert.x use the netty server, while Gigaspaces XAP uses EJB (Enterprise Java Beans) or Spring remoting. For the already available Peer Model implementations, the embedded version uses a low-level mechanism for the abstraction of several wireless radio transmission interface types [29]. The PeerSpace.NET uses the Xcoordination Application Space, which is in principle comparable to XVSM but does not fully follow its concepts [52]. The Mobile Peer Model uses the eXtensible Message and Presence Protocol (XMPP) [54, 58], where XML messages called XML stanzas are transmitted between the instances. XMPP necessarily requires a server component, which in their case is Openfire[18] hosted in the AWS cloud[19].

[17]https://github.com/EsotericSoftware/kryo
[18]https://www.igniterealtime.org/projects/openfire/
[19]https://aws.amazon.com/

5.7 Format evaluation

The choice of an adequate data exchange format has significant consequences on data transmission rates and performance, hence an efficient format is required. Plenty of evaluations between different formats are available on the Internet, many of them being use case driven. Objective evaluations do not put one option in favor of another but conclude that the best format depends on the requirements. The main difference between Protocol Buffers, Thrift, and Avro according to Kleppmann [34] is that

> *Thrift favours the "one-stop shop" style that gives you an entire integrated RPC framework and many choices (with varying cross-language support), whereas Protocol Buffers and Avro appear to follow much more of a "do one thing and do it well" style.*

In [18], for evaluation, a file of 2592000 measurement values (i.e., one month of records having one entry each second) containing a timestamp and a float value was written to the disk and read in again. The time required for persisting the data and the time required for reading the data back in, as well as the generated file size, were evaluated; results are shown in Table 5.2. Note, that the final version of [18] did not include the results for the text-based protocols.

	XML	JSON	YAML	Java Serialization	Protocol Buffers	Apache Thrift	Apache Avro
write time (ms)	3257	1607	8144	8082	238	552	1079
read time (ms)	4915	1818	9237	20694	1314	1665	668
size (MB)	192	121	120	54	38	36	29

Table 5.2: Evaluation Results [18]

A significant overhead in all competitions were observed for the text-based formats. However, the worst results were not achieved by any text-based format, but by the Java Serialization, which on the one hand requires more time (by a factor of 10) than the other options and on the other hand can also not convince regarding file size. In contrast to the other evaluated formats, it even requires more time than JSON and XML; however, as expected the file size is less than half of the text-based formats' sizes. Protocol Buffers and Apache Thrift show comparable results in the benchmarks, with Protocol Buffers leading in write and read speed. The claim [1], however, that Protocol Buffers are 20 to 100 times faster than XML does not hold in this example, where only a factor of $\sim$5 has been observed. Apache Avro is much slower in writing, however much faster in reading.

Regarding file size, Avro excels the other options, with Thrift (or to be more precise its compact protocol) being better than Protocol Buffers. This result was not expected, as sources (e.g., [50]) claim that Protocol Buffers' file size is smaller than Thrift's. The claim [1] that Protocol Buffers is 3 to 10 times smaller than XML holds. Compression in Apache Avro is best, although only in Avro the schema is persisted together with the file.

In result, we concluded in [18] that a winner of the three binary options is hard to determine and subject to a weighting of the results. In the Peer Model use case, the entries will be significantly smaller than the files that were persisted in the use case of [18]. Data transmitted between the Peer Model instances are usually very small in size depending mainly on the entry's user co-data and app-data. All fields that are specified by the Peer Model itself can be expected to require a (nearly) constant size. It can be expected that the size and the time for serialization/deserialization will not differ much between the formats. Thus, the evaluated criteria shall not have the primary influence on the format decision. Table 5.3 lists possible functional and non-functional requirements on the serialization formats and evaluates them for the formats in question.

	XML	JSON	YAML	Java Serialization	Protocol Buffers	Apache Thrift	Apache Avro
platform-independent	+	+	+	−	+	+	+
RPC functionality	−	−	−	◇	◇	+	◇
human-readable	+	+	+	−	−	●	−
additional libraries	−	+	+	−	+	+	+
tools required	−	−	−	−	+	+	+

Table 5.3: Evaluation Results

◇: RPC functionality is not directly included in Java Serialization, Protocol Buffers, and Apache Avro, but there are RPC implementations tightly connected to them.

●: Human-readability in Apache Thrift depends on the protocol in use.

The main criteria for the format decision are the first two entries listed in Table 5.3. It is shown that all of the formats except for the Java serialization allow a platform-independent exchange of data with varying degrees of languages supported between the binary formats. Thrift is the only format that directly includes an RPC framework. Java includes RMI for sending serialized Java objects to other Java programs; Protocol Buffers and Avro use an external RPC framework. As mentioned, in Avro the used schema accompanies the data. In the Peer Model use case, it would be fatal regarding data size to append the schema to each entry that is submitted. For such cases it is possible to reach a consensus on the schema at handshake; that would however require to keep the connection between the instances open.

Human readability may ease debugging but is not required for the productive system. Furthermore, it can already be concluded from Table 5.2 that serialization and deserialization of these formats require more resources than by using binary formats. As Java includes XML parsing functionalities and serialization is an integral part of the language, these are the only formats that do not require additional libraries for execution. The binary formats furthermore require additional tools for generating Java classes from their platform-independent schema file.

When deciding on a platform-independent communication format between Peer Model instances, it was planned that an implementation in Go would follow. Thus, it was crucial to use a communication format that supports Go, which is not the case for Avro. While Protocol Buffers have (at least third party) support for more languages than Thrift, gRPC would significantly reduce the number of possible languages. In result, Thrift excels the other options considered by including RPC functionality and supporting the highest number of platforms.

In the Android implementation that was written concurrently by *Schoba* [54] and *Tillian* [58], Protocol Buffers were chosen as their serialization format mainly due to not requiring RPC functionality in their solution and Protocol Buffers performing slightly better at their benchmarks. However, they designed their system such that the communication system could easily be exchanged to use Apache Thrift later on.

It is significant, that all of the described options have their justification and lead to different decisions when having different requirements. The text-based formats, for example, would be the best option if human-readability is a requirement or in case of communication with web services.

5.8 Apache Thrift in detail

The evaluation of formats in the previous section led to the result that Apache Thrift is the best option for the implementation of communication between Peer Model instances. The Peer Model is in good company with some popular services that also use Thrift, such as the inventor Facebook, hadoop, and last.fm. It allows platform independence and efficiency in data size as well as in serialization and deserialization speed. Furthermore, it already includes an RPC stack, simplifying the necessary network communication. The basics of Apache Thrift were already described in the previous section.

Deeper views inside Apache Thrift that were out of scope in the basic evaluation are given in this section. We show the types that are offered by Apache Thrift and its IDL, and introduce the concept of services used for RPC functionality. The Thrift Network Stack is shown and its layers are described. As Apache Thrift — in contrast to Protocol Buffers and Apache Avro — allows not only one serialization format, a list of possible formats with a comparison is given.

5.8.1 Type System

For the schema definition by use of the IDL, Thrift allows several types. A universal type system requires that respective types in all target languages exist. That is the reason why for instance unsigned integer types are not supported, as they have no representation in all target languages [28]. The limitation of allowing natively defined types only leads to never requiring the developer to write any code for serialization or transport of special types [5]. Types get translated by the compiler to their respective member in the target language.

There are

- eight **base types** (`bool, binary, byte, i16, i32, i64, double` and `string`). In Java, they are mapped to the corresponding primitive types, `strings` to objects of type `java.lang.String`, and `binarys` to `byte[]`.
- **container types** (`list<t>, set<t>` and `map<t1,t2>`) . The type `t` may be any Thrift type except for services. In Java they are mapped to a generic `ArrayList<T>`, `HashSet<T>` or `HashMap<T1,T2>` types.
- **Structs**, defining common objects to be used across languages [5]. They are composed of various fields of any Thrift type, conceptually similar to a C struct and mapped in Java to a class.

  The following rules need to be considered:

  1. Fields need to be marked as either *required* or *optional*
  2. Every field is required to have a unique, positive integer identifier. This tag is used to identify the field in the wire format.
  3. Structs are normal Thrift types. Therefore, it is possible to use a struct type in another struct.
  4. It is possible to specify a default value for optional fields.
  5. Values for required fields always need to be provided, otherwise an error is thrown.

  An example for an optional field with a default value is printed in Listing 5.14.

  ```
  1: optional i32 number = 0
  ```

  Listing 5.14: Optional field with default value

  There is also a **union** type similar to those in C and C++ providing a set of possible fields of which only one can be used.

- **Services**, semantically equivalent to defining an interface in an object-oriented programming language [5]. They are used for RPC, for which the Thrift compiler generates stub classes for the server and the client that implement the interface. Further details can be found in Section 5.8.2.

- **Exception types**, syntactically and functionally equivalent to struct types [5], which are used to declare exceptions in RPC services. The compiler generates classes accordingly to the target language's exception handling, e.g. exception classes extending `org.apache.thrift.TException` in Java.

An example showing base types, container types and structs was printed in Listing 5.11. This example did not include RPC calls and therefore did not show any services and exceptions types. An example showing these latter mentioned types is printed in Listing 5.16. The full Thrift file in use for the Peer Model implementation, including all of the type classes, is printed in Listing 6.29.

Thrift also allows C/C++ style *typedefs* and *enums*. In Java, `typedefs` are not included; therefore, the Thrift compiler generates Java code that uses the base type. Enums are mapped to Java `enum` types. If a namespace is specified in the schema file as printed in Listing 5.15, the generated classes will be filed in the specified Java package. A Thrift file may also include other Thrift files. The Thrift compiler generates one Java class file per struct, enum and service [28].

```
namespace java peermodel.communication.thriftgenerated
```

Listing 5.15: Namespace definition

5.8.2 Services

Services were introduced as a type in the section before. They group one or more related functions, comparable to object-oriented languages. The definition of a service including two functions is somewhat intuitive as shown by example in Listing 5.16. Function calls may have parameters of any Thrift type, they may also declare a return type and exception types that may be thrown.

Communication in a distributed system is unreliable. Thrift allows three ways to call functions:

- Functions that have any return type declared naturally wait for a reply message to be received (i.e. synchronous communication).
- Functions that are marked only with the keyword *void* at least wait for a response from the callee that guarantees that the message was received and the operation has completed on the server's side [5] (i.e., synchronous communication).
- Functions that are marked with the keyword *oneway* indicate that the client only makes a request and does not wait for any response (fire-and-forget) [28]. Obviously, they must not declare a return type (i.e., asynchronous communication).

Whether it is acceptable that an entry may not be received at the server without any notice to the caller shall not be decided here. The general decision is to allow both methods, such that the I/O peer should decide which form of transmission is to be used. Listing 5.16 shows a Thrift service definition that includes both methods. Note that in this example it is also shown that an exception may be thrown in the RPC call.

```
service ExampleService {
   oneway void tryAddEntry(1: Entry entry);
   void addEntry(1: Entry entry) throws (1: NotAccepted exc);
}

exception NotAccepted {
   1: string message
}
```

Listing 5.16: Thrift service definition

The actual Thrift file that is in use in the implementation, including all types and the service, is shown in Listing 6.29.

5.8.3 Thrift Network Stack

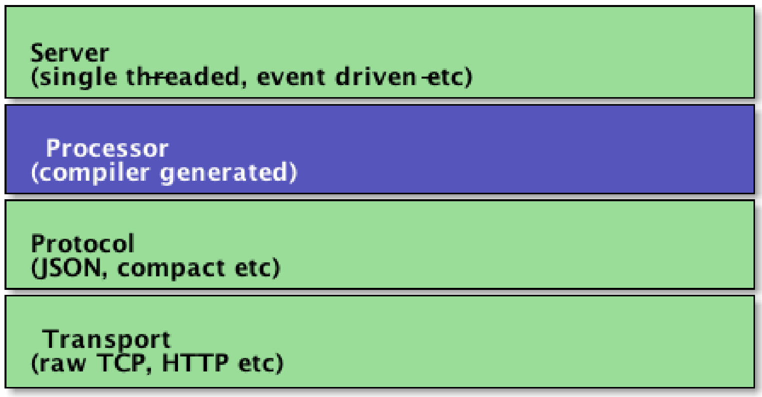


Figure 5.16: Thrift Network Stack [28]

The Thrift Network Stack as shown in Figure 5.16 consists of four layers. Each of those layers provides several exchangeable implementations of a specific interface. Thus, functionalities of different layers are decoupled of each other and can be combined according to particular requirements. Application developers should not be concerned about the transport and protocol layers [5]. Decoupling of these layers allows that generated Thrift code only needs to know how to read and write data; the origin and destination of the data are not relevant [5]. As mentioned, Thrift supports a majority of the currently most used programming languages, however in varying degrees. It may be required to switch to another protocol, as not every implementation implements all protocols. Allowing a configuration for the proper protocols to be used is therefore essential not just for evaluation purposes, but may also be required to allow changes in the settings when communicating with implementations in other languages that do not (yet) support the intended protocols.

This section will introduce a selection of implementations at the four layers that are available in Java and may be suitable for the Peer Model use case. In Section 6.4.3 the layer implementations that are used in the Peer Model implementation are shown.

Transport layer. The Transport layer as the lowest layer encapsulates I/O functionality. In short, this layer defines *how* messages are transmitted between client and server. It provides a simple abstraction for writing to the network and reading from it [28]. Methods of the `TTransport` interface on the client's side (sender) include `open`, `close`, `read`, `write`. On the server's side (receiver), methods of `TServerTransport` include `open`, `listen`, `accept`, `close`.

The choice of the transport layer implementation to be used is dictated by the architecture of the solution [50]. Transmission of data is possible over various channels such as HTTP, sockets or files. The available implementations of `TTransport` differ between the various languages; among available implementations in Java are:

- `TFileTransport`, used to read from and write to files.
- `TIOStreamTransport` with its subclass implementation `TSocket`, providing a wrapper around a standard blocking socket (e.g. `java.io.Socket`), thus only one connection can be active at a time.
- `TNonblockingTransport` with its subclass implementation `TNonblocking-Socket` providing an asynchronous non-blocking socket transport.
- `TSaslTransport`, wrapping another transport implementation for using the Simple Authentication and Security Layer (SASL).

Among the available implementations of `TServerTransport` in Java are:

- `TServerSocket`, providing a wrapper around a `ServerSocket` for a blocking server transport.
- `TNonblockingServerTransport` with its subclass implementation `TNonblockingServerSocket`, providing a wrapper around a `ServerSocket-Channel` for a non-blocking server transport.

Protocol layer. The Protocol layer defines the mechanisms for encoding in-memory data structures to a wire format and the decoding at the receiver [28]. In short, this layer defines *what* is transmitted [49], including how messages are serialized. As such, methods of the `TProtocol` interface include `writeMessageBegin`, `writeString`, readMessageBegin, readString. Examples include binary and text protocols, the latter ones useful for debugging purposes as they are human-readable. The choice of a proper protocol format is crucial to enable efficient data sizes in the transmission. However, at this layer, various implementations are not able to understand all protocols, which may require to switch to another, less efficient format to enable communication with them.

Among the available implementations of TProtocol in Java are:

- TJSONProtocol, a human-readable protocol, e.g., for debugging purposes.
- TBinaryProtocol, a non-human-readable, simple and universal binary protocol (Figure 5.17).
- TCompactProtocol, an optimized binary protocol such that the payload is as small as possible (Figure 5.18).

Kleppmann [34] compared the two binary protocols by serializing the struct in Listing 5.17. Results are shown in Figures 5.17 and 5.18. For this simple example, the TBinaryProtocol requires 59 bytes by using a straightforward encoding, while the TCompactProtocol requires only 34 bytes by using variable-length integers and bit packing.

```
struct Person {
  1: string        userName,
  2: optional i64  favouriteNumber,
  3: list<string>  interests
}
```

Listing 5.17: Struct for binary format comparison [34]

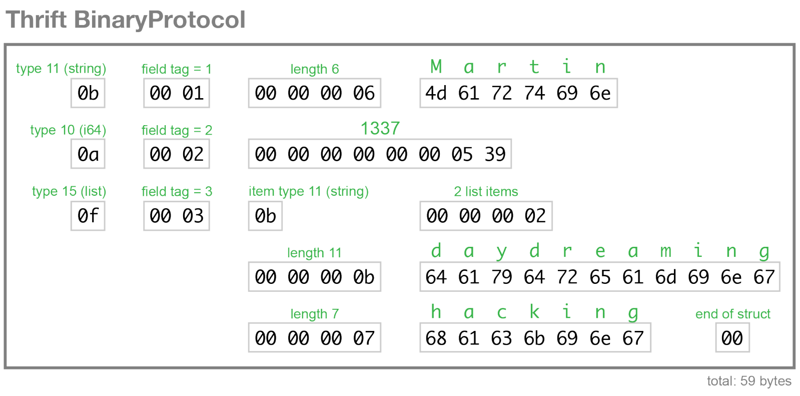


Figure 5.17: Binary Protocol Encoding [34]

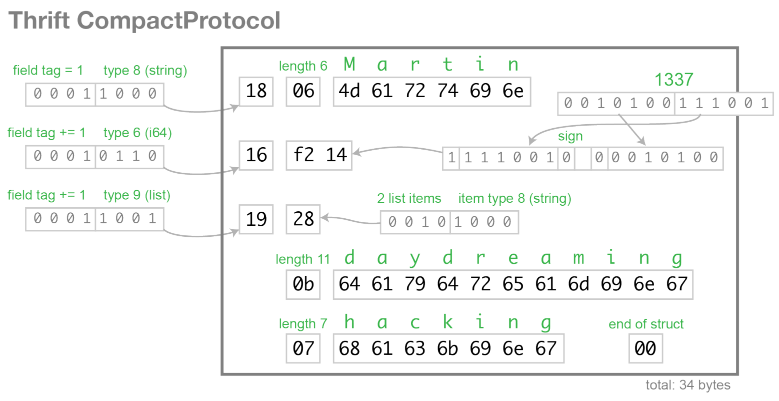


Figure 5.18: Compact Protocol Encoding [34]

Processor layer. The Processor layer reads data from an input stream from the network using the input protocol, delegates processing to a handler implemented by the user and writes the response into an output stream to the network using the output protocol [28]. Service-specific processor implementations are generated by the Thrift compiler.

Server layer. The Server layer finally puts the features described before together [28]. It

- creates a transport.
- creates input/output protocols for the transport.
- creates a processor based on the input/output protocols.
- waits for incoming connections and hands them to the processor.

Various blocking, non-blocking, single and multithreaded server implementations of `TServer` are available, among them:

- `TSimpleServer`, a simple single-threaded server with blocking I/O for testing purposes only.
- `TNonblockingServer`, a server having one I/O thread and one worker thread for fairness amongst all connected clients regarding invocations, but still being able to process only one request at a time.
- `TThreadPoolServer`, a server which uses Java's built-in ThreadPool management for a worker pool. It creates dedicated threads for all incoming connections and thus provides better throughput than the other options while consuming a lot of resources.

Dhanushka [24] concludes that the `TThreadPoolServer` shall be used for a peer-to-peer (P2P) architecture, while the `TNonblockingServer` is best suited for client-server architectures.

Layer implementations Table 5.4 shows the varying support of different layer implementations at the various programming languages supported in Apache Thrift.

	Low-Level Transports						Transport Wrappers			Protocols				Servers				
	Domain	File	Memory	Pipe	Socket	TLS	Framed	http	zlib	Binary	Compact	JSON	Multiplex	Forking	Nonblocking	Simple	Threaded	ThreadPool
ActionScript					X			X		X								
C		X	X		X	X	X			X	X		X			X		
C++	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X
C#			X	X	X	X	X	X		X	X	X	X			X	X	X
Cocoa		X	X		X	X	X	X		X	X		X					X
Common Lisp					X		X			X			X			X		
Dlang		X	X	X	X	X	X	X	X	X	X	X			X	X	X	X
Dart					X		X	X		X	X	X	X					
Delphi					X		X	X		X	X	X	X			X		
.NET Core			X	X	X	X	X	X		X	X	X	X		X			
Erlang		X	X		X	X	X	X		X	X	X	X			X		
Go			X		X	X	X	X	X	X	X	X	X			X		
Haskell		X	X		X	X	X	X		X	X	X				X	X	
Haxe		X			X		X	X		X	X	X	X			X		
Java (SE)		X	X		X	X	X	X	X	X	X	X	X		X	X	X	X
Java (ME)			X		X	X		X		X		X						
Javascript					X	X		X				X	X					
Lua			X		X		X	X		X	X	X				X		
node.js	X				X	X	X	X		X	X	X	X			X		
node.ts					X	X				X						X		
OCaml					X		X			X						X	X	
Perl	X		X		X	X	X	X		X		X	X	X		X		
PHP			X		X	X	X	X		X	X	X	X	X		X		
Python					X	X		X	X	X	X	X	X	X	X	X		
Ruby	X		X		X	X	X	X		X	X	X	X		X	X	X	X
Rust			X		X		X			X	X		X				X	
Smalltalk					X					X								
Swift		X	X		X	X	X	X		X	X		X					X

Table 5.4: Protocol support in different languages (based on [56])

5.8.4 Code generation and inclusion

The Thrift compiler generates code for the services when calling the Thrift compiler using the command in Listing 5.18.

```
thrift --gen java example.thrift
```

Listing 5.18: Thrift compiler call

For the example in Listing 5.16 an `ExampleService` class is generated (limited to the `tryAddEntry` function), including an inner class `ExampleService.Client` for the client side (cf. Listing 5.19) and an inner class `ExampleService.Processor` for the server side (cf. Listing 5.20). The `ExampleService.Iface` shown in Listing 5.21 contains the generated interface. This interface needs to be implemented by the developer; the class needs to be provided to the `ExampleService.Processor` as shown in Listing 5.20 and its methods get executed on a received call. The example uses a `TJSONProtocol` at the protocol layer, `TSocket/TServerSocket` for transport and a blocking `TSimpleServer`. This combination is feasible for testing purposes only due to the single-threaded server and a non-efficient, text-based, human-readable protocol format.

```java
private ExampleService.Client client;

public Client() {
    TTransport transport = new TSocket(/*java.io.Socket*/);
    TProtocol protocol = new TJSONProtocol(transport);
    client = new ExampleService.Client(protocol);
}

public void tryAddEntry(Entry entry) throws TException {
    client.tryAddEntry(entry);
}
```

Listing 5.19: Client

```java
public Server(ExampleService.Iface handler) {
    ExampleService.Processor<? extends ExampleService.Iface> processor =
        new ExampleService.Processor<>(handler);
    TServerTransport serverTransport = new TServerSocket(/*java.io.ServerSocket*/);
    TServer server = new TSimpleServer(
        new Args(serverTransport).
            processor(processor).
            protocolFactory(new TJSONProtocol.Factory()));
    server.serve();
}
```

Listing 5.20: Server

```java
public interface Iface {
    public void tryAddEntry(Entry entry) throws TException;
}
```

Listing 5.21: Interface

CHAPTER 6

Implementation

To ease extensibility/maintainability (NFR2) and modularity/exchangeability (NFR3), the Peer Model implementation at hand was divided into several sub-modules. The used implementation of containers as well as the remote communication can thus easily be replaced.

- **peermodel** module
  including the basic functionality, basic interfaces, and classes (Sections 6.1 and 6.2).
- **containerimpl** module
  including an implementation for containers based on the standard Java Communication Framework JCF (Section 6.3). Earlier it was planned that for *Craß*' PhD thesis another implementation based on XVSM spaces [41] would be used. This decision was later changed; however, it remains easy to substitute the container implementation by replacing this package. The before mentioned *peermodel* sub-project, therefore, contains the interfaces to a container only.
- **communication** module
  including classes for communication between Peer Model instances based on Apache Thrift (Section 6.4). The before mentioned *peermodel* sub-project contains the interfaces for remote communication only.
- **tracing** module
  including classes for logging a Peer Model run to be used in a visual monitoring tool [21] for post-mortem analysis (Section 6.5).

Figure 6.1 shows the relations between the four modules, and the most important classes within them. The depicted classes, as well as the interfaces and enums (printed in italic font) are distributed into several packages within the modules; arrows show inheritance

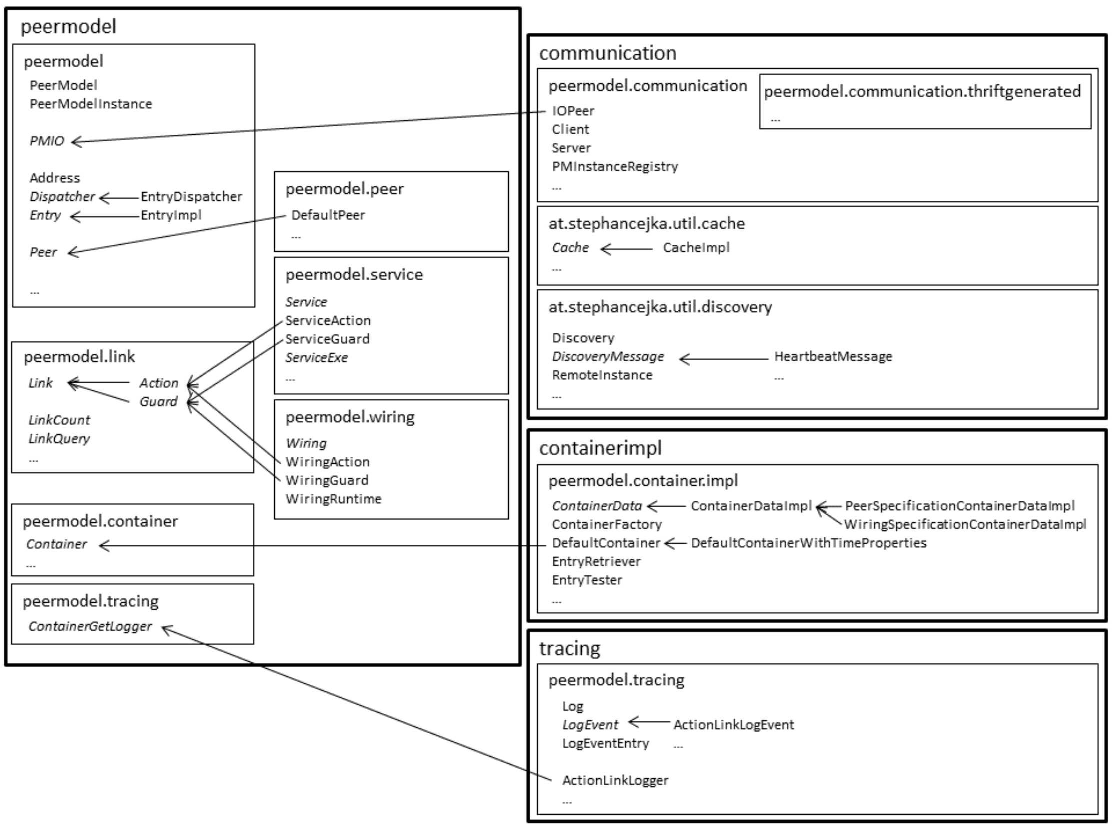


Figure 6.1: Modules and their relations within the Java implementation

relations of classes and implementations of interfaces. They are described in their respective section.

6.1 Peer Model

This section deals with the `peermodel` sub-project within the implementation. It contains the basic classes and functionality, such as peers, wirings, services, and entries. For containers and remote communication, it contains the basic interfaces only. The use-case developer will mainly interact with this sub-project; therefore a brief description of this API will be given in Section 6.2.

A default peer consists of four containers:

- `PIC`, the peer in container
- `POC`, the peer out container
- `WSC`, the wiring specification container
- `PSC`, the peer specification container

While the implementation does not specify any individual behavior on entries put to the first two of the containers mentioned, entries of type `WIRING_TYPE` in the WSC and entries of type `PEER_TYPE` in the PSC need to be specially treated as they reflect the peer's wirings and sub-peers. However, the Java implementation does not bound the number of containers in one peer; therefore, it is possible to create peers with more than the described four containers.

The crucial component of the Peer Model implementation is the following procedure that evaluates for each peer whether a wiring can be executed at that time (i.e., all of its guards can be fulfilled), and in that case executes the wiring. The run of the procedure is initiated

- on start-up of the instance,
- on adding an entry to a container of the peer,
- on adding an entry to a container of a sub-peer,
- on adding a wiring or sub-peer to the peer,
- on remove of a wiring or sub-peer from the peer,
- on remove of an entry from a container of the peer, and
- on remove of an entry from a container of a sub-peer.

Each run of this thread consists of the following steps:

1. **Execution Phase**
   All wirings, which are currently not locked, are tested whether they can be executed. Note that there is no defined ordering of the wirings' execution.
   a) The wiring is locked.
   b) **Wiring Guards**
      i. The concerned sub-peers are locked.
      ii. Guards concerning sub-peers are tested whether they can be fulfilled.
      iii. Guards concerning the own peer's containers are tested whether they can be fulfilled and the entries are retrieved if so.
      iv. Entries that are determined by the guards are retrieved from the sub-peers.
      v. The sub-peers are unlocked.
   c) **Wiring Execution**
      All specified services are executed in order.
      i. **Service Guards**
         A. The service guards specify which entries are to be retrieved from the wiring's entry collection.
         B. If the service execution is specified as `REQUIRED` and a service guard fails, an exception is thrown.
      ii. **Service Execution**
         The service is executed.
      iii. **Service Actions**
         Entries returned by the service and specified in the service's action section are put to the wiring's entry collection.
   d) **Wiring Actions**
      The wiring's action section defines which entries are taken from the entry collection and dispatched to the action's destination container or to the entry's destination as defined by the `DEST` property. If both options are set, the `DEST` property overrules the action destination.
   e) Entries that remain in the entry collection are dismissed.
   f) The wiring is unlocked.
2. **Entry Dispatching Phase**
   The transmission of entries is transparent whether the recipient is located on the same instance or on another one.

a) If the entry's destination is not located on the current instance (i.e., the target instance name is not equal to the current one), the entry dispatcher calls the I/O peer to transmit the entry to the designated receiver instance.
b) Otherwise, the entry dispatcher adds the entry to the local runtime peer, which in turn adds it (recursively) to the correct local sub-peer.

The procedure is executed single-threaded for every peer. If one of the events that trigger the run occurs while active, the procedure gets executed immediately again after it has finished. As soon as the guards are successfully evaluated, and the entries are composed for the wiring execution (1.c), another thread of a thread pool takes over responsibility for the wiring's (and its services') execution. Thus, the testing thread can not be blocked by a long-running service. The thread pool can be limited according to the available resources on the machine. Entry dispatching (2) is equally handled by particular threads tied to the local dispatcher.

6.2 Peer Model API

This section shall briefly describe the Peer Model API, with which the use-case developer will mainly interact. Figure 6.2 shows the package diagram with the most important classes and interfaces of this module.

6.2.1 Initialization of the Peer Model

Listing 6.1 shows the first steps to be undertaken (e.g., within the application's main method) to create a Peer Model instance. Initialization requires an instance name under which the instance is reachable for remote instances and a configuration file.

```
PeerModelInstance.initPeerModel(
    "Test-P-1", "config.properties");
PeerModel model = PeerModelInstance.getInstance();
```

Listing 6.1: Peer Model initialization

Configurable properties are read in by the `ConfigProvider` class. If a value is not specified in the `config.properties` file, the default value is used:

- `maxThreads=16`
  specifies the maximum number of threads concurrently executing a wiring (i.e., wiring executor thread pool).
- `multicastGroupId=234.56.78.90`
  specifies the address that is used for multicast discovery.

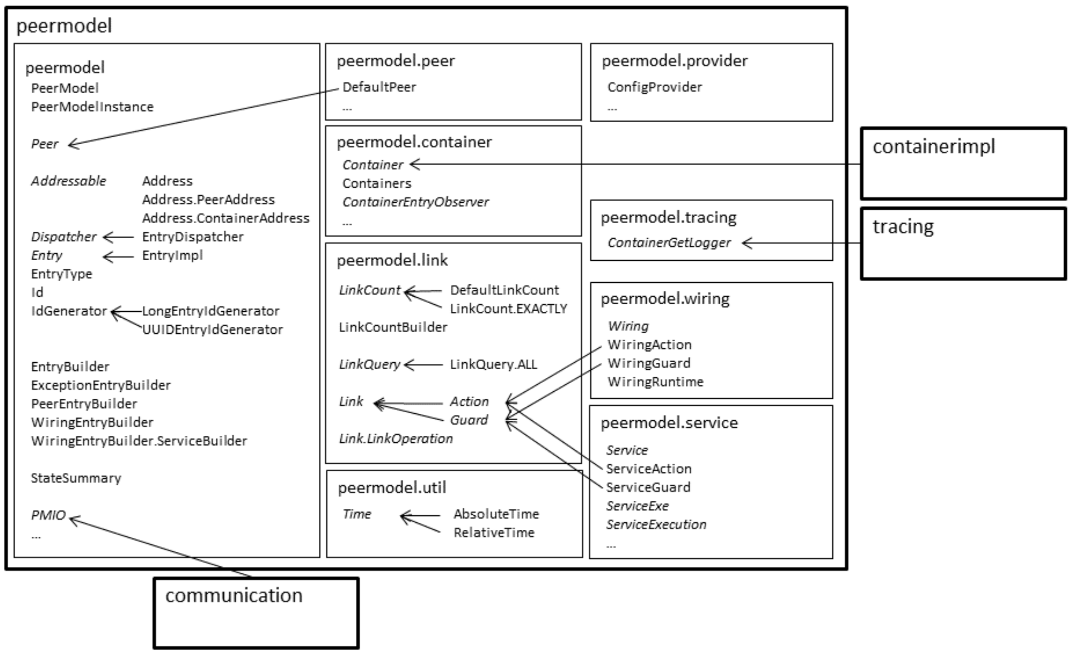


Figure 6.2: Package diagram of the `peermodel` module

- `multicastGroupPort=5123`
  specifies the port that is used for multicast discovery.

Each Peer Model instance consists of a **runtime peer** that is automatically available and is referenced to as "∼". Listing 6.2 shows the most important methods of the `PeerModel` class.

```
public void start();
public void pause();
public void step();
public void resume();
public void close();

public void addEntry(Entry entry);

public void startLogging();
public void endLogging();
```

Listing 6.2: `PeerModel` class

6.2.2 State Summary

For testing purposes, the `StateSummary` class can be used. Listing 6.3 shows its functions, basically comparable to assertions of unit test frameworks.

```
 1  StateSummary summary = new StateSummary(model);
 2  summary.expectEmptyModel();
 3  ...
 4  summary.update();
 5  summary.expect("~:POC", 0);
 6  summary.expect("~:WSC", 0);
 7  summary.expect("~:PIC", 0);
 8  summary.expect("~:PSC", 1);
 9  summary.expect("~/P2:POC", 0);
10  summary.expect("~/P2:WSC", 0);
11  summary.expect("~/P2:PIC", 1);
12  summary.expect("~/P2:PSC", 0);
```

Listing 6.3: State summary

Lines 5-8 expect the POC, WSC and PIC containers of the runtime peer (~) to be empty, i.e., they must not contain any entry. The PSC container contains one entry; the following lines in the listing indeed show that the runtime peer contains one sub-peer `P2`. The method `expectEmptyModel` in line 2 of the listing is a simplification for expecting every container in the model to be empty.

6.2.3 Adding entries

Listing 6.4 shows how to add an entry to the Peer Model. It creates an entry of type `E` with no further co-data or app-data. This entry will eventually be put into the instance's runtime peer (i.e., per default in its `PIC`).

```
model.addEntry(
  new EntryBuilder("E").
      dest(Address.RUNTIME_PEER_ADDRESS).
      build()
);
```

Listing 6.4: Adding an entry

The destination address allows various options. Listing 6.5 shows constants defined in the `peermodel.Address` class:

```
public static final Address.PeerAddress RUNTIME_PEER_ADDRESS
public static final Address.ContainerAddress PICAddress
public static final Address.ContainerAddress POCAddress
public static final Address.ContainerAddress WSCAddress
public static final Address.ContainerAddress PSCAddress
```

Listing 6.5: Address constants

The constructors of an `Address` allows to specify the instance address (for remote destinations), the peer address and the container address. There are default values provided if a part of the address is not set. The available options are printed in Listing 6.6. Note that in this implementation only absolute paths can be used.

```
public PeerAddress(String name)
public ContainerAddress(String name)

public Address(ContainerAddress containerAddress)
// evaluates to the provided container address in the current peer

public Address(PeerAddress peerAddress)
// evaluates to the default container in the given peer

public Address(PeerAddress peerAddress, ContainerAddress containerAddress) {
public Address(String instanceAddress, PeerAddress peerAddress)
public Address(String instanceAddress, PeerAddress peerAddress,
    ContainerAddress containerAddress)
```

Listing 6.6: `peermodel.Address` class

EntryBuilder class

The `EntryBuilder` class creates entries using the entry's type as constructor parameter and the following methods for further specification:

- `newFlow()`
  creates a new flow and sets the entry's flow to the created one. In this implementation, the flow id is a UUID [42] by default.
- `flowId(Id<?> flowId)`
  sets the entry's flow to the given one. This id can be generated by subclasses of `IdGenerator`, which are either `LongEntryIdGenerator` or `UUIDEntryId-Generator`.
- `copyFlowId(Entry entry)`
  set the flow to the one that the given entry is of.
- `tts(Time tts)`
  defining a time-to-start (TTS property), which can either be an instance of class `AbsoluteTime` specifying a Unix timestamp after which the entry is valid or an instance of class `RelativeTime` specifying the time in milliseconds to pass until the entry becomes valid.
- `ttl(Time ttl)`
  defining a time-to-live (TTL property), either as `AbsoluteTime` or as `Relative-Time` specifying when the entry becomes invalid.
- `dest(Address dest)`
  specifying a remote instance, a peer, and/or a container where the entry should eventually be put in (DEST property).
- `coData(String key, Serializable value)`
  specifying use-case-specific co-data (i.e. user co-data)

- `appData(String key, Serializable value)`
  specifying use-case-specific app-data

The TTS, TTL and DEST properties are per default set to null. The event dispatcher throws an exception if an entry with no destination is added to the model; however, if such an entry is created in a service, the entry is dispatched to the destination container of the wiring action link. Further methods are available to copy an entry and to retrieve the mentioned values.

EntryType class

The `EntryType` contains the type of the entry. In normal entries, the type is built using an arbitrary string. Special entry type constants exist for

- `EntryType.WIRING_TYPE`
- `EntryType.PEER_TYPE`
- `EntryType.EXCEPTION_TYPE`

These three special entry types are specifically handled at some occasions in the Peer Model. For example, an entry of type `WIRING_TYPE` that is added at some point to a peer's *WSC* needs to be reflected properly as a wiring in the meta-model.

Entry interface and EntryImpl class

The `build()` method of the `EntryBuilder` class is called to finally create the entry itself. It returns an object of type `EntryImpl` which implements the `Entry` interface. The `Entry` interface only contains the public API for an entry, its methods are shown in Listing 6.7.

```
Serializable getUserCoData(String key);
Serializable getAppData(String key);
EntryType getEntryType();
```

Listing 6.7: `Entry` interface

The `EntryImpl` contains additional methods for retrieving system co-data, i.e. TTS, TTL, DEST, FLOW, and the entry's id. As the parameters of most methods, especially the ones in a user's service implementation, are of type `Entry`, methods for system co-data can remain hidden.

6.2.4 Adding sub-peers

Listing 6.8 shows how to add a sub-peer to a peer in the Peer Model. In this example, a peer `P2` is added as the runtime peer's sub-peer; thus the hierarchically created address of the new peer will be "`~/P2`".

```
model.addEntry(
   new PeerEntryBuilder("P2").
      dest(new Address(
         Address.RUNTIME_PEER_ADDRESS,
         Address.PSCAddress))
      .build()
);
```

Listing 6.8: Adding a sub-peer

The `PeerEntryBuilder` class creates sub-peer entries using the sub-peer's name as constructor parameter and the methods shown in Listing 6.9 for further specification.

```
public PeerEntryBuilder tts(Time tts);
public PeerEntryBuilder ttl(Time ttl);
public PeerEntryBuilder dest(Address dest);

public Entry build();
```

Listing 6.9: `PeerEntryBuilder` class

The entries created by the `build` method use `EntryType.PEER_TYPE` as entry type. They need to be added to a peer's PSC such that they are properly reflected as the peer's sub-peer in the meta-model. Further methods are available to copy an entry and to get the mentioned values.

6.2.5 Adding wirings

Listing 6.10 shows how to add a wiring to a peer in the Peer Model. In this example, a wiring `W1` is added to the runtime peer. The part concerning services is postponed to Listing 6.13 for providing better readability.

```
ServiceBuilder service = ...;

model.addEntry(
   new WiringEntryBuilder("W1").
      guard(
         EntryType.getEntryType("E"),
         Address.PICAddress,
         LinkOperation.TAKE,
         LinkQuery.ALL,
         LinkCount.EXACTLY_ONE,
         false
      ).
      service(
         service
      ).
      action(
         EntryType.getEntryType("E"),
         Address.POCAddress,
         LinkOperation.TAKE,
         LinkQuery.ALL,
         LinkCount.EXACTLY_ONE
      ).
      dest(new Address(
         Address.RUNTIME_PEER_ADDRESS,
            Address.WSCAddress)
        ).
    build()
);
```

Listing 6.10: Adding a wiring

The `WiringEntryBuilder` class creates wiring entries using the wiring's name as constructor parameter. All entries created by the `WiringEntryBuilder` use `EntryType.WIRING_TYPE` as entry type. They need to be added to a peer's WSC such that they are properly represented as the peer's wiring in the meta-model. Further methods (e.g., `tts`) are available like in the `PeerEntryBuilder`. This example shows some complexity, therefore the most relevant methods should be mentioned here:

Listing 6.11 shows the signature of the `guard` method, defining the type of the entry to be taken, the source container, the link operation (`TAKE`, `READ`, `NONE`, `TEST`, `DELETE`, `CREATE`), the link query (`ALL`, `NONE` or a filter as defined later in this section), the link count (as defined later in this section), and whether the guard should act flow-dependently. The affected entries are put into the wiring's entry collection and can then be used by the wiring's services.

```
guard(
    EntryType entryType,
    Address.ContainerAddress source,
    LinkOperation linkOperation,
    LinkQuery linkQuery,
    LinkCount linkCount,
    boolean flowDependent
)
```

Listing 6.11: Guards

Wiring `W1` in Listing 6.10 thus has exactly one flow-independent guard that shall take exactly one entry of type `E` from the PIC without a link query set that would limit the appropriate entries.

The `service` method defines the service to be executed once the guards fire. It uses the `ServiceBuilder` class that will be described later in this section.

Listing 6.12 shows the signature of the `action` method, defining the action to be executed after the service has been executed. Method parameters are comparable to the ones in the `guard` method. The affected entries are taken from the wiring's entry collection and put into the destination container. Entries left in the entry collection after the wiring execution has been concluded (i.e., after executing the actions) are discarded.

```
action(
   EntryType entryType,
   Address.ContainerAddress destination,
   LinkOperation linkOperation,
   LinkQuery linkQuery,
   LinkCount linkCount
)
```

Listing 6.12: Actions

Wiring `W1` in Listing 6.10 thus takes exactly one entry of type `E` from the entry collection and puts it into the POC without a link query set that would limit the appropriate entries.

ServiceBuilder class

Listing 6.13 shows the omitted parts of Listing 6.10 concerning the service definition.

```
ServiceBuilder service = new ServiceBuilder(
    "S1",
    ServiceExecution.REQUIRED,
    Test_P_2_ServiceClass.class,
    ... // parameters for service class constructor
).
guard(
    EntryType.getEntryType("E"),
    LinkOperation.TAKE,
    LinkQuery.ALL,
    LinkCount.EXACTLY_ONE
).
action(
    EntryType.getEntryType("E")
);
```

Listing 6.13: Adding a service

Methods of the `ServiceBuilder` class are comparable to the ones described for the `WiringEntryBuilder` and will therefore not be described in detail. Constructor parameters are the name of the service, whether the service necessarily needs to be executed (`ServiceExecution.REQUIRED`) or not (`ServiceExecution.OPTIONAL`), a class defining the service method and optional user-specified constructor parameters for the service class.

It may be the case that a wiring is executed due to its guards being fulfilled, while the service's guards are not. Services may thus be defined as either required or optional. If the service execution is only optional and the requirements of the service are not fulfilled at that point, the service is skipped, and the execution of the wiring is continued. If the service execution is required and the requirements of the service are not fulfilled at that point, an exception is thrown, logged in the console, and the execution of the wiring is aborted.

Service definition

As shown in Listing 6.13, a class implementation of the service needs to be provided by the use case developer. Such an example service implementation is printed in Listing 6.14.

```
public class Test_P_2_ServiceClass implements ServiceExe {
    public Test_P_2_ServiceClass(List<Serializable> parameters) {
        ...
    }

    @Override
    public Collection<Entry> execute(
       Collection<Entry> entries
    ) {
        System.out.println("service");
        return Collections.singletonList(
           EntryBuilder.simple("E")
        );
    }
}
```

Listing 6.14: Service implementation example

Once `execute` is called (i.e. the guards have fired), the entries for the service are given as the method's parameters. This example service just prints `service` on the console output and returns an entry of type `E` to be delivered by the `action` section.

Link Operation

The `LinkOperation` class defines the operation type in which entries are retrieved by the guard. The following constants are defined (see Section 3.5.1):

- `TAKE`
- `READ`
- `NONE`
- `TEST`
- `DELETE`
- `CREATE`

Link Query

The `LinkQuery` class defines which entries should be retrieved by the guard (see Section 3.5.3), the signature of the `filter` method that needs to be implemented is shown in Listing 6.15.

```
public abstract class LinkQuery {
    abstract boolean filter(Map<String, Serializable> coData);
}
```

Listing 6.15: LinkQuery class

An example of a query is shown in Listing 6.16. In Figure 6.3 (in graphical representation and using the Peer Model syntax) a guard using the example link query is shown.

```
new LinkQuery() {
    @Override
    protected boolean filter(Map<String, Serializable> coData) {
        return coData.containsKey("answer") && coData.get("answer").equals("42");
    }
}
```

Listing 6.16: LinkQuery example

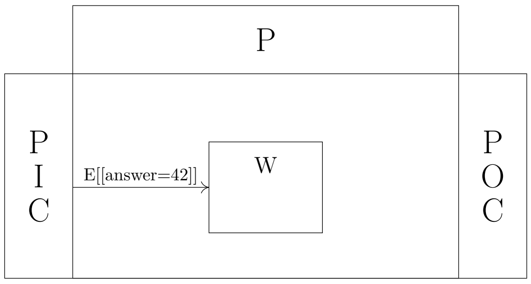


Figure 6.3: The according link query in the graphical representation

For simplification, `LinkQuery.ALL` is predefined; this filter returns `true` on all entries and thus does not filter any entry.

Link Count

The `LinkCount` class defines how many entries should be retrieved by the guard (see Section 3.5.2), its methods that need to be implemented are shown in Listing 6.17.

```
public abstract class LinkCount {
    abstract int getMinimumEntries();
    abstract int getMaximumEntries();
}
```

Listing 6.17: LinkCount class

For simplification, the following most common link counts are predefined:

- `EXACTLY_ONE`
  returns exactly one entry.
- `ALL`
  returns all available entries, including the possibility that zero entries are returned.

- `AT_LEAST_ONE`
  returns all available entries, but at least one.
- `EXACTLY(`n`)`
  returns exactly n entries.
- `NONE`
  returns exactly zero entries. As each guard requires the specification of a link count, this predefined link count shall be used for all guards of `LinkOperation.NONE`.

Other link counts can be defined by use of the `LinkCountBuilder` and the `Default-LinkCount` classes.

Wiring and WiringRuntime classes

Whenever an entry of type `EntryType.WIRING_TYPE` enters the WSC, an object of class `Wiring` is created that serves as the representation of the wiring (cf. Section 6.3.3). Once the procedure described in Section 6.1 determines that the wiring is executable, an object of type `WiringRuntime` is created. It encapsulates the execution of the wiring, including the service calls. It gets executed by a thread of the wiring executor thread pool configured in Section 6.2.1.

6.2.6 Exceptions

Exceptions are another type of entries using `EntryType.EXCEPTION_TYPE` as entry type. The exceptions are build using the `ExceptionEntryBuilder` class; a part of this class is printed in Listing 6.18.

```
private static final long DEFAULT_TTL = 2500L;
private static final Address DEFAULT_DEST =
    new Address(Address.RUNTIME_PEER_ADDRESS, Address.PICAddress);

...

Entry exceptionEntry = new EntryBuilder(EntryType.EXCEPTION_TYPE).
    coData("entry", entry).
    coData("reason", reason).
    ttl(exceptionTTL).
    dest(exceptionDest).
    build();
```

Listing 6.18: Building exceptions

On three occasions, the Peer Model implementation generates exceptions:

1. on the expiration of an entry's TTL [36, 40],
2. on an error at transmission to remote instances, or
3. if the destination of an entry is invalid.

Exceptions have a TTL set, hence they are expected to be short-living and get removed after some time from the model if not handled (i.e., taken by a proper wiring). They contain the original entry, as well as a reason why this exception was generated. The exception's TTL and destination can be set at the original entry as shown in Listing 6.19. They define how long the exception will be valid and where the exception shall be delivered to.

Expired TTL handling Exceptions are generated if the TTL of an entry expires while it has not been taken. Note that exceptions themselves have a TTL set, but no further exception on an expired exception entry is generated.

Example. The example in Listing 6.19 adds three entries to the Peer Model (`A`, `B`, `C`), each having a TTL of 1000 milliseconds set. Note that for simplification, additional co-data like the DEST property are omitted.

```
model.addEntry(new EntryBuilder("A").
    ttl(new RelativeTime(1000L)).
    build());

model.addEntry(new EntryBuilder("B").
    ttl(new RelativeTime(1000L)).
    exceptionDest(dest).
    build());

model.addEntry(new EntryBuilder("C").
    ttl(new RelativeTime(1000L)).
    exceptionDest(dest).
    exceptionTTL(new RelativeTime(5000L)).
    build());
```

Listing 6.19: Exception Handling Example

`A` has no exception-related co-data set.

`B` has an exception destination set.

`C` has exception destination and exception TTL set.

After 2000 ms, the entries `A`, `B`, and `C` have already been removed due to their expired TTL. Instead of them, entries `excB` and `excC` are now in the container; there is no `excA` entry as there were no exception-related properties set.

After further 2000 ms, entry `excB` has been removed due to its expired default TTL. For expired exceptions, no new exception is generated; thus only `excC` is left.

6.3 Container Implementation

This section deals with the `containerimpl` sub-project of the implementation, which is outsourced to provide exchangeability. The interfaces are defined directly in the `peermodel` module, which can in turn be implemented by different concrete implementations, for example by using a database, a space, or an in-memory data structure. The container implementation is chosen by an entry in a property file stating a factory class name. This class is instantiated using Java's reflection API. Thus, it is easy to replace the current implementation with another, perhaps more efficient one in future work. Figure 6.4 shows the package diagram with the most important classes and interfaces of this module.

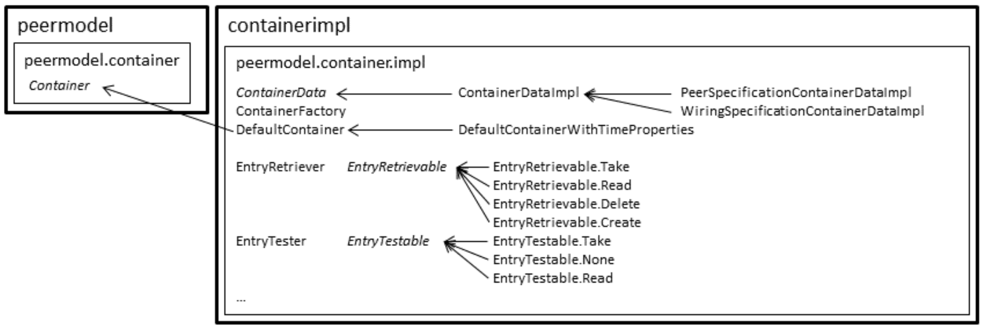


Figure 6.4: Package diagram of the `containerimpl` module

The current implementation uses the in-memory standard collections of the Java Collection Framework (JCF). This implementation mainly consists of a map having an entry type as key and a set containing all entries of this type as the value (such a map is also known as a multi-map). According to the set definition, it is not guaranteed that entries are retrieved in any specific or fair order. *Selectors* (e.g., FIFO, priority, etc.) that would determine a particular order of the entries to be taken (cf. [36, 40]) are not available in this implementation.

Containers (in the broader sense) are used for

1. the peer's containers (i.e., containers in the narrower sense), and
2. the wiring's entry collection.

While the first type requires proper handling of TTS and TTL properties, the entry collection must not take these into account.

For a peer's container, an entry is not yet available for guards if their TTS has not been reached. They are therefore for the time being put into a temporary store. An entry will no longer be available for any guard if its TTL is expired. The proper handling of peer containers is implemented by the `DefaultContainerWithTimeProperties` class.

In the entry collection, entries with a TTS set, nevertheless, need to be handled immediately. The implementation thus ignores these time properties; in effect, entries are delivered to the destination container immediately. The receiver has to prohibit access to the received entries as long as the entry's TTS is not reached. The proper handling of the entry collection container is implemented by the `DefaultContainer` class.

6.3.1 Link implementation and entry retrieval

The testing whether all requirements are fulfilled and the subsequent retrieval of these entries need to take place in an atomic manner. Therefore, no other operations are possible during that time. As the requirements may address not only one container of a peer, during the described procedure, all other accesses to any of the peer's containers as well as affected sub-peers are suspended.

The basic methods, specified by the `Container` interface in the `peermodel` sub-project are:

- `boolean isAvailable(List<? extends Link> requirements)`
  This method tests whether all requirements can be fulfilled by (some of) the current elements in this container. It returns false if any requirement could not be fulfilled.
- `Collection<Entry> get(List<? extends Link> requirements)`
  This method returns elements from this container according to the requirements. If the conditions cannot be fulfilled, an exception is thrown. Flow-dependent guards restrict the entries retrieved to those specified by the first entry's flow ID and to those entries not having any flow ID set.
- `Collection<Entry> testAndGet(List<? extends Link> reqs)`
  This method returns elements from this container according to the requirements. The method first calls the `isAvailable` method and returns null if any requirement cannot be fulfilled.

Depending on the link operation, entries are taken (removed) or copied from the container. For a successful execution, all requirements need to be fulfilled. Efficiency requires further requirements to be no longer tested if any requirement is already known to be unsatisfiable. Thus, it is suitable to use Java's functional programming features, i.e., the Stream API and lambda expressions [45], for the evaluation of the eligible entries. As the operation needs to be atomic, it is not possible that any additional entries are added, or any entries are removed while the testing takes place. If at least one of the requirements cannot be fulfilled, no entries are read or taken from the container nor are any entries returned to the caller.

Testing for available entries

The `isAvailable` method calls the `EntryTester` class, which evaluates whether the conditions can be fulfilled.

- For a **TAKE**, **READ**, **TEST** or **DELETE** link, it returns true if the number of available entries (of the defined type) that fulfill the specified condition (i.e. link query) is at least the minimum required number of entries. A simplification of the procedure for the TAKE operation is shown in Listing 6.20. Note that the testing procedure is equivalent for the READ, TEST, and DELETE operations.

```
 1  public boolean isAvailable(
 2          Link link, Id<?> flowId, Map<EntryType, Set<Entry>> entries) {
 3      Set<Entry> entriesForThisType = entries.get(link.getEntryType());
 4      Stream<Entry> entryStream = entriesForThisType.stream();
 5
 6      LinkQuery query = link.getLinkQuery();
 7      entryStream = entryStream.filter(query);
 8
 9      if (link.isFlowDependant()) {
10          entryStream = entryStream.filter(entry ->
11              entry.getFlowId() == null ||
12              entry.getFlowId().equals(flowId));
13      }
14
15      LinkCount count = link.getLinkCount();
16      int minimumEntries = count.getMinimumEntries();
17      return entryStream.count() >= minimumEntries;
18  }
```

Listing 6.20: Simplified entry testing example for the TAKE operation (`EntryTestable.Take`)

A high level description of the entry testing example follows: In lines 3-4, the stream of entries of the requested entry type is retrieved from the container. In lines 6-7, these entries are filtered according to the link query. If the link is flow-dependent, the entries that do not share the evaluated flow ID are filtered out (except for entries that have no flow ID set) in lines 9-13. Finally, in lines 15-17 it is evaluated whether the remaining entries in the stream obey the desired number of entries. Thus, this method returns true, if the number of entries is at least as high as the number of minimum required entries.

- For a **NONE** link, it returns true if the number of available entries that fulfill the specified condition is zero.
- For a **CREATE** link, it always returns true.

Retrieving entries

The `get` method calls the `EntryRetriever` class, which returns entries from the container that fulfill the conditions.

- For a **TAKE** link, it returns the maximum number of entries that are required, available in the source container, and that fulfill the condition. Furthermore, they get removed from the source container. A simplification of this procedure is shown in Listing 6.21.

```
 1  public Collection<Entry> get(
 2          Link link, Id<?> flowId, Map<EntryType, Set<Entry>> entries) {
 3      Set<Entry> entriesForThisType = entries.get(link.getEntryType());
 4      Stream<Entry> entryStream = entriesForThisType.stream();
 5
 6      LinkQuery query = link.getLinkQuery();
 7      entryStream = entryStream.filter(query);
 8
 9      if (link.isFlowDependant()) {
10          entryStream = entryStream.filter(entry ->
11              entry.getFlowId() == null ||
12              entry.getFlowId().equals(flowId));
13      }
14
15      LinkCount count = link.getLinkCount();
16      int minimumEntries = count.getMinimumEntries();
17      int maximumEntries = count.getMaximumEntries();
18      List<Entry> gotten = entryStream.limit(maximumEntries).
19          collect(Collectors.toList());
20      if (gotten.size() < minimumEntries) {
21          throw new UnexpectedContainerException();
22      }
23
24      gotten.forEach(entriesForThisType::remove);
25
26      return gotten;
27  }
```

Listing 6.21: Simplified entry retrieval example for the TAKE operation (`EntryRetrievable.Take`)

A high level description of the entry testing example follows: Lines 3-13 are equivalent to Listing 6.20 and the description given there. In lines 15-22 it is evaluated whether the remaining entries in the stream obey the desired number of entries. If the number of entries underrun the minimum number of required entries, an exception is thrown. Otherwise, the stream is cut at the maximum number of entries, such that the number of entries will be between the minimum and the maximum requested count. Finally, just before the return of the method, in line 24 all entries that are taken from the container are indeed removed there.

- For a **READ** link, it returns copies of the maximum number of entries that are required, available in the source container, and that fulfill the condition. The original entries are kept in the source container.

- For a **TEST** or **NONE** link, it returns no elements and keeps the source container unchanged.

- For a **DELETE** link, it returns no elements but removes the maximum number of available entries that fulfill the condition from the source container.

- For a **CREATE** link, it creates and returns new entries keeping the contents of the source container unchanged. The number of generated entries is the maximum link count, and if the link is defined as flow dependent the current flow under observation is used for the generated entries. Other co-data are not set.

Several events triggered by the entry retrieval were described in the procedure in Section 6.1. Thus on added or removed entries of a container, the container's observer is called to initiate another run of the procedure of the affected peer and of its parent peer. Depending on the requirement guards, it may be the case that after a get operation, the observer gets notified that zero entries were removed. This is a normal case if only read, test or none operations were executed. Respectively, it may be the case that after a get operation, the observer gets notified that some entries were removed, but the get operation does not return any elements. This is a normal case if only delete operations were executed. A guarantee that elements were removed from the container and were also returned to the caller can only be given if at least one take operation is part of the requirements.

6.3.2 Handling of TTS and TTL

TTS and TTL handling are achieved by using delayed executions provided by the `java.util.concurrent.Delayed` interface. The `java.util.concurrent.DelayQueue` is an unbounded `BlockingQueue` of `Delayed` elements, in which an element can only be taken when its delay has expired. Elements in this queue are ordered according to the expiration of the Delayed element. Expiration occurs when an element's `getDelay()` method returns a value less than or equal to zero. Accesses to elements that are not yet expired via the queue's interface are not possible.

If an entry is transmitted to a remote instance, an absolute time (i.e., Unix timestamp) is used for the TTS and TTL. Depending on the remote system clock it could be the case that an entry with a TTS set becomes immediately valid; an entry with a TTL set, however, could have already been expired, respectively.

6.3.3 Special handling of entries in PSC and WSC

Entries of type `PEER_TYPE` or `WIRING_TYPE` added to the PSC or WSC need to be appropriately reflected in the Peer Model, as they represent sub-peers and wirings.

Wiring handling If a wiring entry is added to the WSC, this entry is transformed into an object of class `Wiring` as represented by the entry's co-data. As there is a thread pool providing the threads for wirings' execution (cf. Section 6.2.1), there is no need to create wiring-specific threads at the time of the wiring's creation. The wiring is then added to the intended peer.

If a wiring entry is taken (removed) from the WSC, this needs to be also represented by the meta-model. In that case, the wiring is removed from the peer; however, it if is currently executed the execution will be completed.

This described behavior is implemented by the `WiringSpecificationContainer-DataImpl` class.

Sub-Peer handling Likewise to wiring handling, peer entries added or removed from the PSC need to be reflected in the meta-model. On the creation of a new sub-peer, the according threads for the peer are created (e.g., for the procedure described in Section 6.1) and the peer is added to the sub-peer list of the parent.

On shutdown, the peer is removed from the sub-peer list of the parent and the threads are stopped. However, the execution of wirings that are currently running is completed. Furthermore, this shutdown procedure is done for all of the peer's own sub-peers.

This behavior is implemented by the `PeerSpecificationContainerDataImpl` class.

Example. The example code in Listing 6.22 adds a wiring W2 to the WSC of the runtime peer that will take one entry of the `PEER_TYPE` (i.e., sub-peers) from the PSC. Thus, adding this entry to the WSC needs to be reflected as a new wiring of the peer. Furthermore, once the guard fires and the wiring gets executed, the removal of the sub-peer needs to be reflected in the model, too.

```
model.addEntry(
    new WiringEntryBuilder("W2").
        guard(EntryType.PEER_TYPE, Address.PSCAddress, Link.LinkOperation.TAKE,
            LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
        dest(new Address(Address.RUNTIME_PEER_ADDRESS, Address.WSCAddress)).
        build()
    );
```

Listing 6.22: Example for PSC and WSC handling

6.4 Communication

This section deals with the `communication` sub-project of the implementation. To allow remote communication to be replaceable, it is outsourced to this package. This implementation uses Apache Thrift (cf. Section 5.8) for remoting. Figure 6.5 shows the package diagram with the most important classes and interfaces of this module.

6.4.1 Instance Discovery and List

Concept

The discovery of other Peer Model instances is achieved by using multicast messaging over UDP as described in Section 5.3.2. To simplify the further description, an instance that is already up and running is termed *A*, while we call the newly joining instance *B*. For simplification we assume that *A* is the only instance that is up and *B* is the second instance joining.

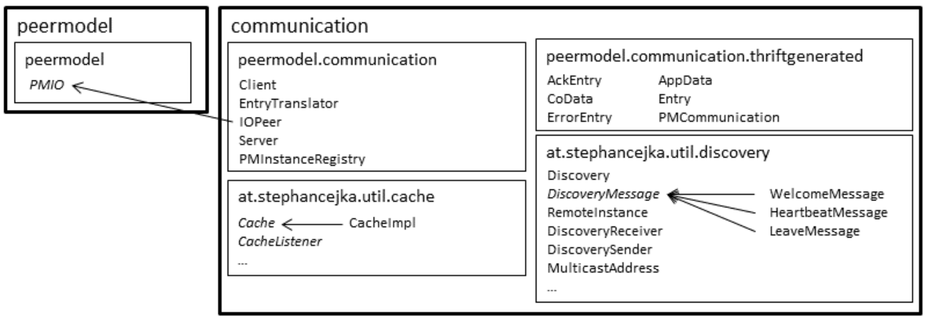


Figure 6.5: Package diagram of the `communication` module

Once *B* comes online, it joins a multicast group (configurable via the config file, cf. Section 6.2.1); thus it can receive messages that are sent to this group by other instances. *B* sends a *PMHeartbeat* message to the multicast group containing the instance's name, address and port where the instance is running and able to receive messages. *A* receives that message and temporarily saves the new instance in its cache table. As the cache by design drops entries after a specified hold time, it is required to send a *PMHeartbeat* message in a specified interval. These messages are sent to the multicast address, such that all instances that are still part of the group receive this message. In that way, the entry in the cache is renewed on reception of the heartbeat. If the source address or the port contained in the message changes, the old values in the cache are replaced.

On the planned leave of an instance, a *PMLeave* message is sent to the multicast address informing all members of the group of its leave. On reception of this message, the instance is immediately removed from the cache. If an instance retires silently or the network connection to the instance gets lost, no such message is sent. No *PMHeartbeat* messages will be received any longer, eventually resulting in a drop of this instance in the cache. Entries that are sent during that time span to an instance that is no longer reachable may get lost if the *IOPeer* employs a fire-and-forget manner. In that case, the sending instance does not get informed whether the transmission of an entry to another instance failed or of the successful reception of the entry at the other instance.

Implementation

As the discovery protocol is required to be used among implementations in other programming languages, it is required to describe the format of such messages more in-depth. All of these messages are sent per UDP to the configured multicast address.

PMHeartbeat message Heartbeat messages are sent on the join of a new instance as well as in a specified interval to inform other instances that this instance is still alive. The period in which these messages are sent is currently defined to be 15 seconds. The Java implementation drops all instances of which no heartbeat was received within two periods. A simplified `PMHeartbeat` message originating from an instance named *PM123* running on TCP port *54321* is shown in Listing 6.23. The host of the instance can easily be identified as the UDP packet's sender IP address.

```
H PM123 54321
```

Listing 6.23: `PMHeartbeat` message

PMWelcome message Welcome messages are sent as the reply to a join of an up to now unknown instance. A simplified `PMWelcome` message originating from an instance named *PM456* running on TCP port *56789* is shown in Listing 6.24. The host of the instance can easily be identified as the UDP packet's sender IP address.

```
W PM456 56789
```

Listing 6.24: `PMWelcome` message

PMLeave message Leave messages are sent on the planned discharge of an instance. A `PMLeave` message originating from an instance named *PM123* running on TCP port *54321* is shown in Listing 6.25. The host of the instance can easily be identified as the UDP packet's sender IP address. Only the instance's name would be required for the leave message, but in that way the same message structure is kept.

```
L PM123 54321
```

Listing 6.25: `PMLeave` message

Cache

We require a cache for temporarily holding the remote instances. A cache is a collection that saves entries for a defined time and removes those entries afterward. Such a cache collection implementation is missing in the JDK. To avoid including unnecessary big packages such as Google Guava or Apache Commons, an own implementation was created. Instances of a cache are constructed as shown in Listing 6.26.

```
Cache.<MapKey,MapValue>createCache(
    ExpirationPolicy policy, long time, TimeUnit unit)
```

Listing 6.26: Creating the cache

`MapKey` is the type of the key of the cache's entries, `MapValue` the type of the value of the cache's entries. The time and the time unit specify how long an entry should be held in the cache before it is removed again. Which elements are removed is based on the expiration policy, which is one of the following two:

1. **CREATED policy:** This expiration policy specifies that entries expire – hence get removed – the defined time after their creation.

2. **ACCESSED policy:** This expiration policy specifies that entries expire – hence get removed – the defined time after the entry has been last accessed.

In the Peer Model implementation, we use the CREATED policy only. Once a heartbeat message is received, the instance's information is put into the cache. Whenever a new heartbeat message is received from that instance, the instance entry is renewed. If no heartbeat message is received within the defined hold time, the entry is automatically dismissed from the cache. Any read access – i.e., when trying to find the instance to send a Peer Model entry to – is not relevant for the expiration of the instance entry in the cache.

The cache naturally contains methods for putting an entry to the cache, retrieving an entry with and without the subsequent removal of this entry, the number of current elements and the boolean query whether an element is contained in the cache. It follows as much as possible the method names and behavior that is specified by Java's `java.util.Map` interface. All possible API calls are shown in Listing 6.27; methods behavior should be self-explanatory except for the `renewCreate` method. This method is used for renewing the entry when using the CREATED expiration policy. The expiration time will no longer be based on the entry's creation time but on the time of the call of this method.

```
public interface Cache<MapKey, MapValue> {
   ExpirationPolicy getExpirationPolicy();
   long getHoldTime(TimeUnit unit);
   void put(MapKey key, MapValue value);
   void addCacheListener(
      CacheListener<MapKey, MapValue> listener);
   void removeCacheListener(
      CacheListener<MapKey, MapValue> listener);
   MapValue get(MapKey key);
   MapValue remove(MapKey key);
   int size();
   boolean contains(MapKey key);
   void renewCreate(MapKey key);
}
```

Listing 6.27: `Cache` interface

6.4.2 Remote instance registry

In the Peer Model implementation, whenever an entry in the cache is removed due to the expiration of its time, a listener interface (`CacheListener`) updates the instance registry that holds all currently known remote instances. Listing 6.28 shows the methods of the `PMInstanceRegistry` class that get executed whenever an instance joins or retires. The remote instances are obtained by the `IOPeer` whenever a Peer Model entry is going to be transmitted.

```
void instanceJoined(RemoteInstance instance);
void instanceRetired(RemoteInstance instance);
RemoteInstance getInstance(String name);
```

Listing 6.28: `PMInstanceRegistry` class

6.4.3 Remote entry transmission

As described in Section 5.8, Thrift's network stack consists of four layers. On these four layers, different possible protocols and implementations can be used. For this chapter, it remains to decide which implementation to use.

- **Transport layer**: In the Peer Model implementation, `TSocket`, which provides a wrapper to standard sockets, is used. On the recipient side, the `TServerSocket` provides a wrapper to standard server sockets.
- **Protocol layer**: In the Peer Model implementation, the binary `TCompact-Protocol` is used. As described, not all protocols may be available in all languages (cf. Figure 5.4). Therefore, another protocol needs to be chosen if communication with future Peer Model implementations that do not support this protocol is necessary.
- **Processor layer**: As described in Section 5.8, this layer is always generated by the compiler.
- **Server layer**: In the Peer Model implementation, the `TThreadPoolServer` is used (cf. [24]).

The basic interface of the communication API is `PMIO` in the `peermodel` module. The following methods are available and need to be implemented by classes:

- `void entryIn(Entry)`
  for the local add of an entry that is received from a remote location.
- `void entryOut(Entry)`
  for the transmission of an entry to a remote instance.

The `IOPeer` class implements this interface and encapsulates instance discovery, the instance registry, the server and the client for RPC calls. Classes for the communication are generated by the Thrift compiler from the Thrift file, shown in Listing 6.29. It extends the basic type definitions printed in Listing 5.11 as well as the basic service definitions printed in Listing 5.16 and combines them. For an acknowledged delivery of an entry (as defined next in this section) a struct `AckEntry` is introduced, as well as an exception type `ErrorEntry` generated on delivery failures.

```
namespace java peermodel.communication.thriftgenerated

struct Entry {
    1: required string entryType;
    2: required string entryId;
    3: optional i64 tts;
    4: optional i64 ttl;
    5: optional string flowId;
    6: required string destination;

    7: optional map<string,CoData> coData;
    8: optional map<string,AppData> appData;

    9: optional binary subjectTree; // used for Secure PM extension
    10: optional binary credentials; // used for Secure PM extension
}

union CoData {
    1: string coDataString,
    2: binary coDataBinary,
    3: map<string,CoData> coDataMap,
    4: list<CoData> coDataList
}

union AppData {
    1: string appDataString,
    2: binary appDataBinary,
    3: map<string,AppData> appDataMap,
    4: list<AppData> appDataList
}

struct AckEntry {
    1: required Entry entry;
}

exception ErrorEntry {
    1: required Entry entry;
    2: optional string message;
}

service PMCommunication {
    oneway void tryAddEntry(1: Entry entry);
    void addEntry(1: Entry entry);
    AckEntry ackAddEntry(1: Entry entry)
        throws (1: ErrorEntry error);
}
```

Listing 6.29: Thrift file

The Thrift compiler (cf. Section 5.8.4) generates classes for `AckEntry`, `AppData`, `CoData`, `Entry`, and `ErrorEntry` for the struct types as well as the class `PMCommunication` for the service. The classes that are generated by the Thrift compiler are used for communication only and are different to the ones that are used in the Peer Model implementation. The `EntryTranslator` class thus provides translation of entries in both directions. Generated classes are very long, e.g. the generated `PMCommunication` consists of over 2300 lines of code. In the Java implementation, the `ErrorEntry` struct is mapped to a normal Java exception class.

Remote entry delivery failure detection and handling

Messages are delivered in an at-most-once kind such that no guaranteed delivery can be assumed. This means that a message can get lost, it is either delivered once or not at all. It shows the highest performance with the least implementation overhead because a fire-and-forget fashion does not need to keep track of messages. In comparison, an at-least-once delivery or an exactly-once delivery are much more complicated to implement, as they require keeping a state and necessarily need to provide an acknowledgment mechanism. Problems can arise during various steps of the transmission procedure [44]:

1. Is the message sent out on the network?
2. Is the message received by the other host?
3. Is the message put into the target's mailbox?
4. Is the message processed (in any kind) by the target?
5. Is the message processed successfully by the target?

Three types of errors can be distinguished for the transmission of entries:

- **Recipient unknown (type *a* failure)**
  The transmission of the entry fails due to the recipient being unknown at the sender side (e.g., the receiver was not detected by the discovery mechanism or has left since).
- **Transmission failed (type *b* failure)**
  The transmission of the entry fails due to a transmission error (e.g., the recipient – although known – was not available).
- **Remote Not Delivered to Recipient (type *c* failure)** The transmission of the entry fails due to an unknown destination on the recipient side (e.g., the peer to which the message should be delivered to is not available).

Generally, entries could be sent to remote instances using one of the four enumerated message delivery types (Table 6.1). Based on the delivery option, exception entries – introduced in Section 6.2.6 – are generated on a detected delivery failure either at the sender or the recipient instance.

1. **fire and forget**
   When using this delivery type, it remains unknown whether the message was even sent. It corresponds to the `tryAddEntry` option of the Thrift file (Listing 6.29), which is marked with the keyword `oneway`. Thus there will not be any response from a receiver, and no exception entry will be generated ever.

		Detect type *a* failures	Detect type *b* failures	Detect type *c* failures
1	fire and forget	no	no	no
2	acknowledge send	yes	no	no
3	acknowledge transmission	yes	yes	no
4	acknowledge delivery	yes	yes	yes

Table 6.1: Transmission error detection

2. **acknowledge send**
   When using this delivery type, it is guaranteed on success that the message was sent, but not if it has successfully been transmitted or received at the recipient. This option, however, cannot be represented by the Thrift RPC functionality.

3. **acknowledge transmission**
   When using this delivery type, it can be guaranteed that the message has successfully been transmitted to and received on the recipient Peer Model instance. It cannot be guaranteed that the message was delivered to the correct peer or container at this instance (i.e., if the recipient peer is unknown, the delivery would be successful for the sender instance). It corresponds to the `addEntry` option of the Thrift file (Listing 6.29), which is marked with the keyword `void`. Thus the only response from the receiver is the successful return of the call; otherwise an exception entry would be generated on the sender side (e.g., on timeout).

4. **acknowledge delivery**
   Using this delivery type on each transmission either an `AckEntry` or an `ErrorEntry` is returned; guaranteeing on reception of the `AckEntry` that the entry was transmitted to the correct peer receiver in the remote instance. It corresponds to the `ackAddEntry` option of the Thrift file (Listing 6.29), which has a defined return type (`AckEntry`) and a defined exception type that is thrown on an error (`ErrorEntry`). In that case, an exception entry is generated on the recipient side.

Server

The server provides the handler for messages that reach the recipient. It is called by Thrift once a message was received. Note, that the implementations for `tryAddEntry` and `addEntry` are identical on this level – Thrift has already confirmed the successful delivery in the second case, while there is no confirmation in the first case.

```
@Override
public void tryAddEntry(
  peermodel.communication.thriftgenerated.Entry entry)
  throws TException {
    IOPeer.this.addEntry(EntryTranslator.toPMEntry(entry));
}

@Override
public void addEntry(
  peermodel.communication.thriftgenerated.Entry entry)
  throws TException {
    IOPeer.this.addEntry(EntryTranslator.toPMEntry(entry));
}

@Override
public AckEntry ackAddEntry(
  peermodel.communication.thriftgenerated.Entry entry)
  throws TException {
    IOPeer.this.addEntry(EntryTranslator.toPMEntry(entry));
    return new AckEntry(entry);
}
```

Listing 6.30: Thrift Server

Client

Whenever an entry needs to be transmitted to a remote location, the IOPeer's `transmit` method is called. Based on the destination's instance address, the registry either returns the remote instance information saved in the cache or cancels the operation with an exception that the instance is unknown. Lookup errors (i.e. type a errors) can thus be detected by using the Peer Model's internal lookup mechanism (`PMInstanceRegistry` class), even when using fire and forget in Thrift. Sub-peer entries and wiring entries can be transmitted like other entries; note – however – that the service execution implementation class cannot be serialized and transmitted. The service class thus needs to be within the remote instance's classpath to work.

6.5 Tracing

This section deals with the `tracing` module of the implementation. Figure 6.6 shows the package diagram with the most important classes and interfaces of this module.

The tracing format was defined by the diploma and PhD students within the Space Based Computing group in 2014 (see Section 3.8). Results of this workshop were included in [21]; however, *Csuk*'s visualizer implementation does not run on logs following this format. In future work, it will be necessary to update the visualizer to follow the specification and thus be able to visualize the generated traces by the Java Peer Model implementation following the specification.

The methods to start and stop the logging procedure were already shown in Listing 6.2. As specified, the tracing is split into a VIL and a TIL file, which are both represented in

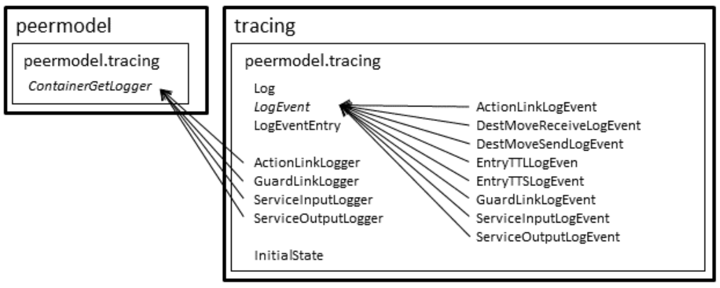


Figure 6.6: Package diagram of the `tracing` module

JSON format. As described in Section 5.4.1, first-level support of JSON is still missing in Java. In this case, we only require to generate a JSON file, not to parse one. Complex big implementations like Jackson are therefore not required here; to provide all necessary functionality we use the `minimal-json` package[1], requiring only 33 kB in the current version 0.9.5.

Visualisation Intermediate Language (VIL) file

The VIL file is generated at the end of the logging procedure, as at that time all components that were used are known. It should be avoided to create and remove wirings and sub-peers while tracing is active as the format does not allow any changes in the structure during the log run. The VIL file defines the used `entryTypes`, `services`, `wirings` and `peers` that are referenced in the TIL file. Although the visualizer does not allow to review distributed runs over more than one instance, the VIL file also contains a `processors` section defining the Peer Model instance (cf. Section 5.2).

Trace Intermediate Language (TIL) file

The TIL file contains the initial state and the events that occurred during the run. The possible types of logged events are according to the eight enumerated event types in Section 3.8.2.

[1]`https://github.com/ralfstx/minimal-json`

CHAPTER 7

Evaluation

This chapter first compares the Java implementation with the related work for evaluation (Section 7.1). Section 7.2 benchmarks the implementation to show its performance and scalability. Finally, the chapter is concluded with a description of the fulfilled requirements introduced in Chapter 4 (Section 7.3).

7.1 Comparison with related work

A first comparison between related work was already made in Section 2.6. It evaluated which features are missing in the evaluated frameworks, as well as in the earlier Peer Model implementations, and thus raised related functional and non-functional requirements (cf. FR2, FR3, NFR1). In this section, the Java implementation is compared with the related work frameworks using the criteria that were defined there.

7.1.1 Acting on messages

In Section 2.6.1, three use cases were introduced that were complicated or even impossible to be implemented in the evaluated frameworks. Only the Peer Model is able to act on more than one message of various types at once, and to act when no message of a certain type is available.

Use Case 1: *A function shall be executed every time five messages of type* A *have been received.* Using the Peer Model, this use case only requires to define a certain link count for the guard (Figure 7.1 and Listing 7.1; note that the call to the service is simplified).

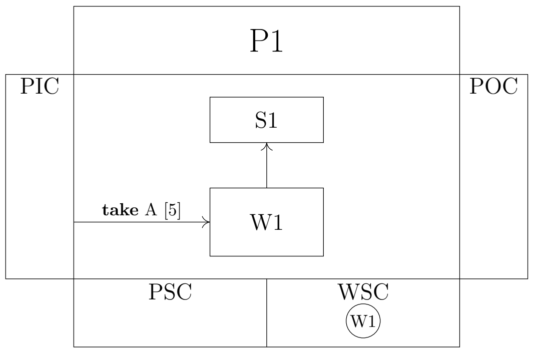


Figure 7.1: Use case 1 in PM graphical representation

```
model.addEntry(
   new WiringEntryBuilder("W1").
      guard(
         EntryType.getEntryType("A"), Address.PICAddress, Link.LinkOperation.TAKE,
         LinkQuery.ALL, new LinkCount.EXACTLY(5), false).
      service(
         new WiringEntryBuilder.ServiceBuilder("S1", ...).
         ...).
      dest(new Address(new Address.PeerAddress("P1"), Address.WSCAddress)).
      build()
);
```

Listing 7.1: Use Case 1 implementation

Use Case 2: *A function shall be executed if one message of type* A *and one message of type* B *have been received. The message of type* A*, however, shall not be removed after it has been processed, i.e., as long as the message of type* A *is still available, the function shall be executed each time a message of type* B *arrives.* Using the Peer Model, this use case only requires to utilize a READ guard for type *A* and a TAKE guard for type *B* (Figure 7.2 and Listing 7.2; note that the call to the service is simplified).

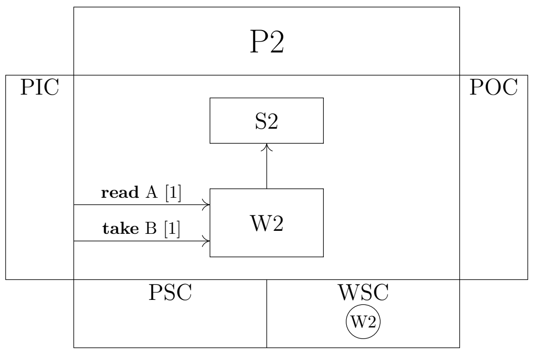


Figure 7.2: Use case 2 in PM graphical representation

```
model.addEntry(
   new WiringEntryBuilder("W2").
      guard(
         EntryType.getEntryType("A"), Address.PICAddress, Link.LinkOperation.READ,
         LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
      guard(
         EntryType.getEntryType("B"), Address.PICAddress, Link.LinkOperation.TAKE,
         LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
      service(
         new WiringEntryBuilder.ServiceBuilder("S2", ...).
         ...).
      dest(new Address(new Address.PeerAddress("P2"), Address.WSCAddress)).
      build()
);
```

Listing 7.2: Use Case 2 implementation

Use Case 3: *A function shall be executed whenever no entry of type* C *is available.* Using the Peer Model, this use case only requires to utilize a NONE guard for type *C* (Figure 7.3 and Listing 7.3; note that the call to the service is simplified). Note that using the NONE guard alone, the framework could run into an endless loop; thus, it is advised that at least one consuming guard is defined for every wiring.

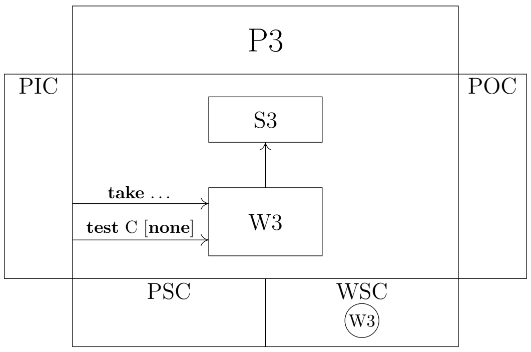


Figure 7.3: Use case 3 in PM graphical representation

```
model.addEntry(
   new WiringEntryBuilder("W3").
      guard(..., Link.LinkOperation.TAKE, ...).
      guard(
         EntryType.getEntryType("C"), Address.PICAddress, Link.LinkOperation.NONE,
         LinkQuery.ALL, LinkCount.NONE, false).
      service(
         new WiringEntryBuilder.ServiceBuilder("S3", ...).
         ...).
      dest(new Address(new Address.PeerAddress("P3"), Address.WSCAddress)).
      build()
);
```

Listing 7.3: Use Case 3 implementation

7.1.2 Interoperability

In Section 2.6.2 it was described that a central requirement of any distributed systems is a communication mechanism. However, most of the evaluated frameworks, including the previous Peer Model implementations, do not allow communication with instances using other implementations. The thesis thus defined a platform-independent communication format to allow communication between instances implemented in different languages. By using a framework that is supported in the majority of current programming languages, succeeding Peer Model implementations in other programming languages can thus communicate with each other; the Java implementation being the first one implementing the format.

Communication with implementations in another language
Akka ☐ vert.x/Gridlink ☑ WS-BPEL ☐ GigaSpaces XAP ⊟
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐
Java implementation ☑

7.1.3 Discovery

In Section 2.6.3 it was described that additionally to the communication format, there needs to be a platform-independent way of discovery of other instances. However, not all of the evaluated frameworks, including previous Peer Model implementations, have included a discovery mechanism. The thesis thus defined a platform-independent protocol for discovery of instances implemented in different languages. Succeeding Peer Model implementations in other programming languages can thus discover each other; the Java implementation being the first one implementing the protocol.

Akka ☐ vert.x/Gridlink ☑ WS-BPEL ☐ GigaSpaces XAP ☑
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐
Java implementation ☑

7.1.4 Dynamic exchange of logics

In Section 2.6.4 it was described that the Peer Model meta-model was not yet implemented in any of the previous Peer Model implementations. Thus, the Java implementation is the first implementation that includes the meta-model and thus the handling of wirings and peers as special types of entries.

Akka ⊟ vert.x/Gridlink ⊟ WS-BPEL ☐ GigaSpaces XAP ☐
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐
Java implementation ☑

7.1.5 Remote component creation and termination

In Section 2.6.5 it was described that components (i.e., wirings and peers) cannot be created and terminated on a remote instance in previous Peer Model implementations due to their tight connection with the meta-model. Thus, the Java implementation is the first implementation that allows to transmit peer and wiring entries to remote instances like any other entry; note, however, that the service implementation needs to be available in the classpath of the receiving instance as the service implementation is not serializable.

Akka ☑ vert.x/Gridlink [~] WS-BPEL ☐ GigaSpaces XAP ☐
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☐
Java implementation ☑

7.1.6 Exceptions

In Section 2.6.6 it was described that exception entries shall be introduced as additonal types of entries, especially to handle expired entries and communication failures properly. This entry type was already supported by the Mobile Peer Model implementation. Depending on the transmission mode (cf. Section 6.4.3), exceptions are generated at the Java Peer Model implementation on transmission failures. The use case developer thus needs to react adequately for exceptions to issue retries. Furthermore, exception entries are generated on an expired TTL.

Akka ☑ vert.x/Gridlink ☑ WS-BPEL ☑ GigaSpaces XAP ☐
Embedded PeerSpace ☐ PeerSpace.NET ☐ Mobile Peer Model ☑
Java implementation ☑

7.2 Benchmarks

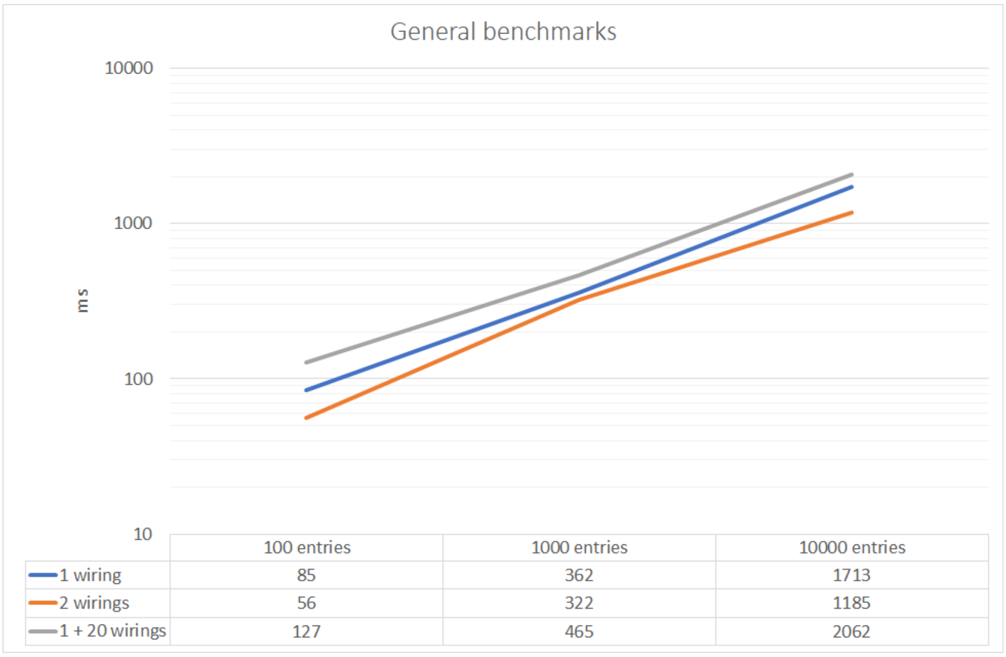
This section presents benchmarks to show the performance and scalability of the implementation. First, a general benchmark is shown in Section 7.2.1, afterwards a benchmark involving communication with a remote instance is shown in Section 7.2.2.

7.2.1 General benchmarks

This benchmark evaluates how fast wirings are triggered and thus shall prove scalability. The detailed setting is described in Appendix A.1.

Tests were made for 100, 1000, and 10000 entries to be processed, i.e., the time required from starting to add entries until the last entry had been processed by the service. In the first benchmark, only one processing wiring was added to the Peer Model instance. As a wiring cannot be executed again concurrently while it is already running, it was furthermore tested how it influences the results if a second (identical) wiring is added. As the current implementation tests all wirings on each incoming entry (cf. Section 6.1), it was also tested how the results are influenced when additional 20 wirings are added to the peer.

The results shown in Figure 7.4 are as expected. When using a second wiring, the required time is reduced by a bit, while the additional wirings introduce some – although no significant – penalty. By using a logarithmic scale it is clearly shown that the solution scales.



	100 entries	1000 entries	10000 entries
1 wiring	85	362	1713
2 wirings	56	322	1185
1 + 20 wirings	127	465	2062

Figure 7.4: General benchmark results

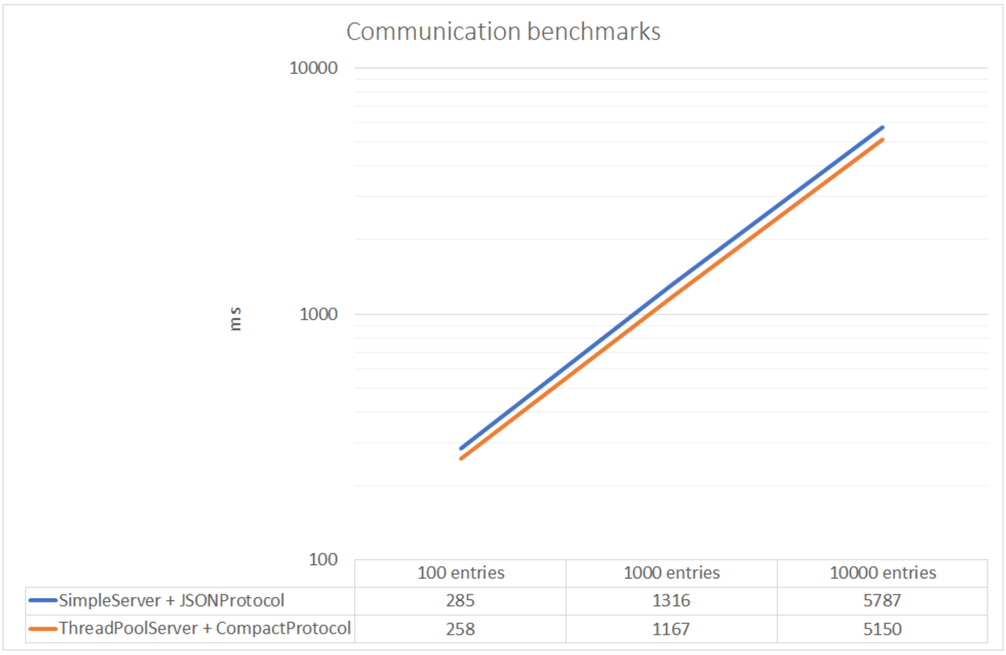
Furthermore, it was tested how long it takes until a service is first executed for a wiring requiring only one entry. Thus the time between adding this entry to the instance until the service execution was measured – it took around 15 ms.

7.2.2 Communication benchmarks

In extension to the first benchmark, the communication benchmarks test how fast wirings are triggered using collaboration with a remote instance. The detailed setting is described in Appendix A.2.

Again, tests were made for 100, 1000, and 10000 entries to be processed. They were added on the local instance having a remote destination set, transmitted to that instance, and processed there. The remote service generates a response entry on each processed original entry. Its destination is set to the original instance, where it is again transmitted to and processed there in a local responder service.

The time that was required from starting to add entries until the last entry had been processed by the local responder service is shown in Figure 7.5. The benchmarks were made with two settings: on one hand using the `TSimpleServer` with a human-readable `TJSONProtocol`, and on the other hand using the `TThreadPoolServer` with the `TCompactProtocol`. It is shown that the latter option saves approximately 10 % of the time. Once again, by using a logarithmic scale it is shown that the solution scales.



	100 entries	1000 entries	10000 entries
SimpleServer + JSONProtocol	285	1316	5787
ThreadPoolServer + CompactProtocol	258	1167	5150

Figure 7.5: Communication benchmark results

7.3 Fulfillment of requirements

This section shall evaluate whether the functional and non-functional requirements as defined in Chapter 4 are reasonably fulfilled by the Java implementation.

7.3.1 Functional requirements

Limited feature completeness (FR1): As initially defined, the Java implementation supports the initially introduced concepts of the Peer Model as specified (cf. [38, 37, 39]) and keeps subsequently introduced concepts (e.g., variables, transactions; cf. [35, 36, 40]) open for future work.

Support for the meta-model (FR2): As the meta-model [20] already introduced new containers (especially WSC and PSC), the specification of containers was extended to theoretically allowing an arbitrary number of containers. Wirings and peers are treated as special entry types on which the two mentioned designated containers react adequately, such that adding and removing entries of those types are adequately reflected in the Peer Model runtime.

Language-independent remoting (FR3): As the primary task of this thesis and the implementation was to enable collaboration with remote instances, an appropriate format for discovery and remoting that would even support to communicate with instances in a vast number of other programming languages was introduced and implemented.

Execution of user-defined service code (FR4): The seperation of concern leads to a strict decoupling of coordination logic and application logic. While the responsibility for coordination logic lies in the Peer Model components, the use-case developer is free and solely responsible for the application logic (i.e., business logic) by implementing use-case-specific services.

7.3.2 Non-functional requirements

Interoperability (NFR1): Language-independent remoting requires for a interoperable solution. The thesis evaluated several mechanisms for discovery of other instances and the subsequent communication with them. The solution described in this thesis and implemented in the Java implementation allows interoperability for discovery with all platforms that support multicast addressing. For communication, Apache Thrift as the chosen solution currently supports more than 25 different languages at various maturity levels (cf. Figure 5.4), among them the currently most used languages. Thus, the requirement for interoperability is fulfilled, although additional implementation effort is required to allow communication with the earlier Peer Model implementations in C [29, 31], C# [52], and Android [54, 58].

Extensibility / Maintainability (NFR2): The later introduced concepts (variables, transactions, etc.; cf. [35, 36, 40]) were intentionally not included into the Java implementation for now. As described, an inclusion of these and further components to a following version requires an extensible solution. Its extensibility and maintainability, however, was proven as security concepts have already been included into the implementation in the context of a PhD thesis at work. This inclusion had been further extended using a Public Key Infrastructure (PKI) in a Public Resource Computing use case in *Lettmayer*'s diploma thesis [43]. As the mentioned PhD thesis and the diploma thesis were in work concurrently to this diploma thesis, it was possible to influence the development of the Java implementation based on their comments and experiences. Consequently, the requirement for extensibility and maintainability is sufficiently fulfilled.

Modularity / Exchangeability (NFR3): As shown in Figure 6.1, components intended to be replacable or that seemed suitable (i.e., communication module, container implementation, tracing) were outsourced into their own module with interfaces defined between the modules. This aspect simplifies their replacement, for example, for using a database or a space as the container implementation instead. In result, the requirement for modularity and exchangeability is reasonably fulfilled.

Scalability (NFR4): The scalability of the solution was shown in the benchmarks. By implementing the meta-model and enabling remote communication in the implementation, entities (i.e., wirings, peers, and whole instances) can come and go during the instance's runtime. Thus, the collaboration of two or more instances is feasible using this implementation. Consequently, the solution allows for developing scalable use cases.

Simple API / Usability (NFR5): A full example for a simple use case involving remote communication is printed in Appendix A.2. While it might seem complicated at first glance, it has a straightforward API for coordination logic as well as for application logic. Already during development of the Java implementation it had been used for the implementation of a use case concerning routing mechanisms by *Zischka* [60] in the context of his diploma thesis. Just like with the extending use cases (cf. NFR2), it was thus possible to let the user's perspective influence the development of the Java implementation based on his comments and experiences with it.

Performance (NFR6): Reasonable performance of the Java implementation was shown in the benchmarks. However, the primary focus was not laid on performance; evidently, there is room for improvements at this point as has been pointed out especially at the container implementation for entry testing and their subsequent retrieval.

CHAPTER 8

Conclusion

In this last chapter of the thesis, it remains open to point out possible future work (Section 8.1) and to conclude the thesis with a summary of the work done (Section 8.2).

8.1 Future Work

The future work can be divided into three categories:

- first, the general future work in the Peer Model toolchain (Section 8.1.1),
- second, issues remaining open for this thesis to smoothly integrate with the tools of the toolchain that are already available (Section 8.1.2), and
- finally, improvements in the implementation itself (Section 8.1.3).

8.1.1 Peer Model toolchain

Some years have passed since the original publication of the Peer Model in [38], and the Peer Model kept evolving during this time span. Referring to the honeycomb image introduced in [37], it can be seen which tools are planned but not yet implemented for the Peer Model toolchain. General future work in the Peer Model area includes the missing tools shown in white in the honeycomb picture in Figure 2.1. These are the (formal) verification of a Peer Model, a graphical modeling tool and a simulation tool. Other tools are currently in development or have been finished during the writing of this thesis.

8.1.2 Integration issues

For smooth integration of the implementation into the toolchain, some issues arise. The visual monitoring software developed by *Csuk* [21] is currently able to provide post-mortem analysis of one instance's trace only. With the introduction of dynamics and the Peer Model's meta-model, the VIL file structure needs refinement, and the visualizer needs to correctly treat wirings and peers that come and go during runtime. As now communication between different instances is defined and possible, the tool needs to be extended to allow the inclusion of traces of more than one instance. Furthermore, it would ease debugging if it would be possible to include not only post-mortem analysis but also enable live monitoring.

Further open issues are the integration of this implementation with other tools, e.g., the code generator and the PM-DSL. Those two tools have been developed quite some time before this thesis; thus they need to be enabled to generate sources for the Java implementation.

8.1.3 Implementation-specific improvements

Since the thesis and the implementation focused on enabling communication between Peer Model instances, the implementation is not as performant as it could be. Improvements on these aspects, for example, the implementation of a more performant container API or using a database, or a space instead, remain as future work. For this reason, extendability and exchangeability were one of the requirements; especially with containers, communication and tracing having been outsourced to their own modules. Currently, each operation that affects any entry in any container of a peer leads to testing of all of its wirings whether they can be executed now or not (depending on whether the guards are fulfilled). A suggested improvement would be an intelligent algorithm that decides which wirings are going to be tested by taking into account that the ability to fulfill most of the wirings will probably not be affected by the insertion of an additional entry in a container. A solution to this issue could be the use of a tree. As a result of the general benchmarks (cf. Figure 7.4), the wiring execution could be extended to allow concurrent executions of the same wiring. Furthermore, an improved version could a priori exclude flows to be evaluated depending on the known current availability of entries in the containers.

As the time in which this thesis and the implementation were written was longer than usual diploma theses require, the Peer Model has further evolved. New components, such as transactions, variables, and assertions [35, 36, 40], were newly introduced but have not been considered in this work. Further work will be required to extend the Java implementation to address these concepts.

Exception handling has been introduced in this thesis for TTL handling and communication failures; still missing is the fail-safe handling of those exception entries including the development of patterns to handle them.

Addressing has been introduced in two forms – URI addressing and naming. While the naming approach that was followed is sufficient for local networks, WAN support will necessarily require URI addressing. Therefore, remoting and the I/O peer will require additional work to allow enterprise use cases in the WAN.

The Secure Peer Space, implementing the features that were introduced in [20] only by means of original Peer Model concepts, is already in development. Furthermore, the Java implementation has already been extended by *Lettmayer* [43] to use a secured protocol between the instances. To be able to change the Thrift protocols on demand, they shall be configurable – also when keeping in mind that some protocols may not be available in various Thrift implementations.

Further improvements may stem from future use cases that will utilize this implementation.

8.2 Summary

This thesis introduced the enterprise implementation of the Peer Model in Java. It enables scalable collaboration by introducing platform-independent communication for the Peer Model. Thus, this thesis evaluated different approaches for the naming and addressing of instances. It introduced a particular discovery mechanism for Peer Model instances in the local network. Further networking with instances outside the local network was intentionally left out and remains as future work. Several serialization and communication formats were evaluated to find suitable ones that support a vast number of different programming languages and still show proper performance.

With the introduced discovery mechanism and the communication format decided, this implementation now allows for building scalable solutions where the involved Peer Model instances do not necessarily need to use this Java implementation. While other implementations of the Peer Model already exist in C, C#, and Android, this implementation is also the first one that supports the meta-model. Thus, the implementation allows to dynamically add and remove entities during runtime (i.e., wirings, peers) without consequences on the functionality. Furthermore, by utilizing the Peer Model's discovery mechanism, instances can join and leave during the runtime of a system of connected Peer Model instances. As the specification of the Peer Model has continually evolved during the last few years, some of the later introduced concepts, such as variables, assertions, and transactions [35, 36, 40], have not yet been considered for the Java implementation. Their inclusion into this implementation remains open as future work.

Previous theses in the Peer Model area compared the model with similar approaches (e.g., the Actor Model) as suitable for their use cases. Thus, this thesis puts the primary focus of evaluation on communication. As just mentioned, various tools and implementations in the Peer Model environment have already been finished, are currently in work, or are planned. They will all integrate eventually into a toolchain. Continuous evaluation of the implementation was possible due to various diploma theses and one PhD thesis already using this implementation for their use cases. Therefore, several helpful comments and feature requests could successfully influence this work during the implementation phase.

APPENDIX A

Benchmark settings

In this appendix, details about the benchmarks in Section 7.2 are shown. The classification made there is kept, thus the general benchmarks' settings can be found in Section A.1, while the communication benchmarks are covered in Section A.2.

For each benchmark, 10 test rounds were executed. Since the variances of the results were not significant, the mean value was calculated and used for the comparison charts. The following test environment was used:

CPU: Intel Core i7-7500U Dual Core 2.7-3.5 GHz
RAM: 8 GB
OS: Windows 10
Java 8

First of all, the `ServiceHelper` class that is used throughout the benchmarks needs to be introduced (Listing A.1). Its `startXXX` methods are called on the start of the according benchmark to set the start time stamp. The other methods are called from the service implementations on processing an entry. Once the last entry to be processed is handled, the deviation of the current time to the start time stamp determines the required time for that benchmark.

```java
public class ServiceHelper implements Serializable {

  private long firstTriggeringStartedAt;

  private long bulkTriggeringStartedAt;
  private int expectedEntries;
  private final AtomicInteger seenEntries;
  private final AtomicInteger triggered;

  ServiceHelper() {
    this.seenEntries = new AtomicInteger(0);
    this.triggered = new AtomicInteger(0);
  }

  public void startFirstTriggering() {
    this.firstTriggeringStartedAt = System.nanoTime();
  }

  public void endFirstTriggering() {
    long required = System.nanoTime() - firstTriggeringStartedAt;
    System.out.println("required␣time:" + required / 1000000);
  }

  public void startBulkTriggering(int expectedEntries) {
    this.expectedEntries = expectedEntries;
    this.bulkTriggeringStartedAt = System.nanoTime();
  }

  public void updateBulkTriggering(int size) {
    triggered.incrementAndGet();
    int currentEntries = seenEntries.addAndGet(size);
    if (currentEntries == expectedEntries) {
      long required = System.nanoTime() - bulkTriggeringStartedAt;
      System.out.println("required␣time:␣" + required / 1000000);
      System.out.println("service␣triggered:␣" + triggered.get());
    }
  }
}
```

Listing A.1: `ServiceHelper` class

A.1 General benchmarks

For the general benchmarks, the setting is shown in Figure A.1. *W1* and *S1* are used for the first benchmark that evaluates how long it takes until the service is executed (Section A.1.1). *W2* and *S2* are used for the second benchmark that evaluates how long it takes until n entries of type `request` were processed (Section A.1.2).

Both benchmarks use the class `Benchmark1` that is printed in Listing A.2. It prepares the structure that was shown in Figure A.1. It thus adds *P1* to the instance's *PSC* and afterwards the wirings *W1* and *W2* to *P1*'s *WSC*. Additional (benchmark-specific) code that is inserted in this class as well as the services' implementations are printed at the according benchmark.

```java
public class Benchmark1 {

  private static final int N = ...;

  public static void main(String[] args) throws InterruptedException {
    PeerModelInstance.initPeerModel("Benchmark1", "config.properties");
    PeerModel model = PeerModelInstance.getInstance();
    model.start();

    model.addEntry(
      new PeerEntryBuilder("P1").
        dest(new Address(Address.RUNTIME_PEER_ADDRESS, Address.PSCAddress)).
```

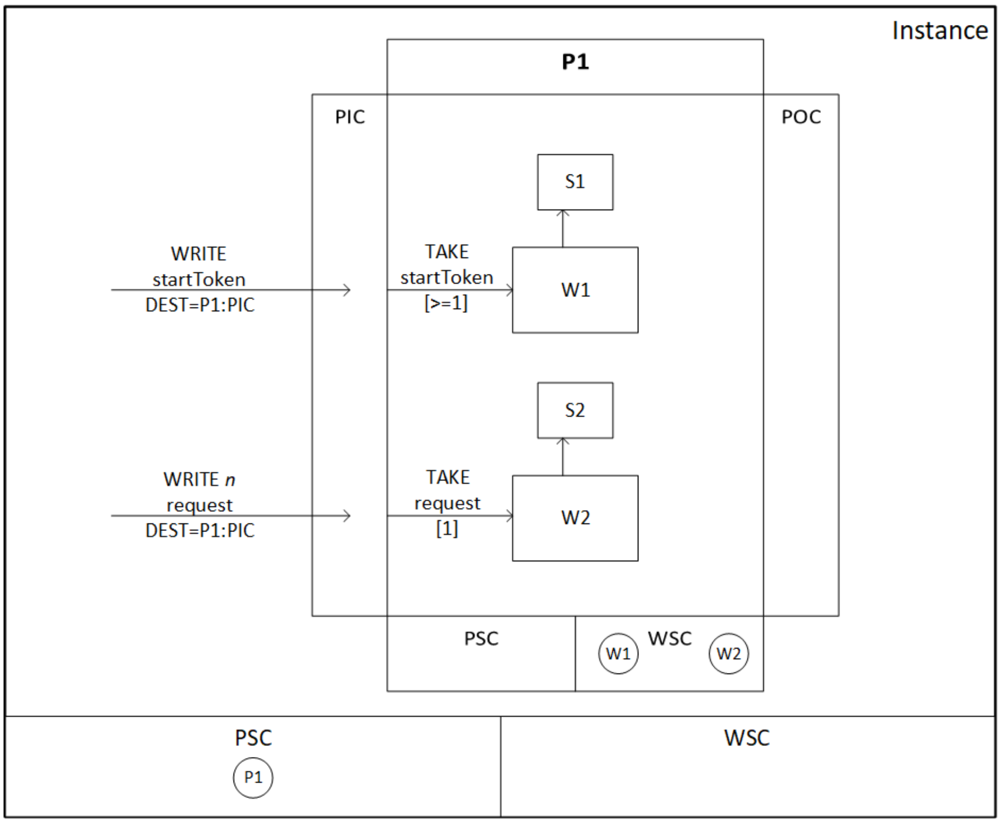


Figure A.1: General benchmarks setting

```
        build()
    );

    ServiceHelper serviceHelper = new ServiceHelper();
    List<Serializable> serviceArgs = Collections.singletonList(serviceHelper);

    model.addEntry(
      new WiringEntryBuilder("W1").
        guard(EntryType.getEntryType("startToken"), Address.PICAddress,
            Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.AT_LEAST_ONE, false).
        service(
          new WiringEntryBuilder.ServiceBuilder("S1", Service1.class, serviceArgs).
            guard(EntryType.getEntryType("startToken"), Link.LinkOperation.TAKE,
                LinkQuery.ALL, LinkCount.AT_LEAST_ONE)
        ).
        dest(new Address(new Address.PeerAddress("P1"), Address.WSCAddress)).
        build()
    );

    model.addEntry(
      new WiringEntryBuilder("W2").
        guard(EntryType.getEntryType("request"), Address.PICAddress,
            Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
        service(
          new WiringEntryBuilder.ServiceBuilder("S2", Service2.class, serviceArgs).
```

```
38              guard(EntryType.getEntryType("request"), Link.LinkOperation.TAKE,
39                  LinkQuery.ALL, LinkCount.EXACTLY_ONE)
40            ).
41            dest(new Address(new Address.PeerAddress("P1"), Address.WSCAddress)).
42            build()
43        );
44
45        // see next sections
46
47    }
48  }
```

Listing A.2: `Benchmark1` class

A.1.1 Benchmark 1: First triggering of a wiring

The first benchmark (implemented by use of *W1* and *S1*) evaluates how long it takes until the service has processed an entry.

Coordination logic

The code section of Listing A.3 is inserted into line 45 of Listing A.2. Before the entry is added to the instance, the service helper is called to save the start time stamp.

```
serviceHelper.startFirstTriggering();
model.addEntry(
  new EntryBuilder("startToken").
    dest(new Address.PeerAddress("P1")).
    build()
);
```

Listing A.3: Additional source code for benchmark 1 in `Benchmark1` class

Application logic

Listing A.4 shows the implementation of *S1*. As soon as the service is called, the service helper is called to determine the end time stamp, and thus to calculate the time that was required until the service had been called.

```
public class Service1 implements ServiceExe {

  private final ServiceHelper serviceHelper;

  public Service1(List<Serializable> serviceArgs) {
    this.serviceHelper = (ServiceHelper) serviceArgs.get(0);
  }

  @Override
  public Collection<Entry> execute(Collection<Entry> entries) {
    serviceHelper.endFirstTriggering();
    return null;
  }
}
```

Listing A.4: `Service1` class

A.1.2 Benchmark 2: Processing *n* entries

The second benchmark (implemented by use of *W2* and *S2*) evaluates how long it takes until *n* entries have been processed by the service, i.e., how long it takes until the wiring is executed *n* times for the entries.

Coordination logic

Benchmark 2a. This benchmark evaluates the processing of *n* entries using one wiring. The code section of Listing A.5 is inserted into line 45 of Listing A.2. Before the entries are added to the instance, the service helper is called to save the start time stamp.

```java
serviceHelper.startBulkTriggering(N);
for (int i = 0; i < N; i++) {
  model.addEntry(
    new EntryBuilder("request").
      dest(new Address.PeerAddress("P1")).
      build()
  );
}
```

Listing A.5: Additional source code for benchmark 2a–2c in `Benchmark1` class

Benchmark 2b. This benchmark evaluates the processing of *n* entries using two identical wirings. The code section of Listing A.6 adds another adequate wiring; it is inserted into line 45 of Listing A.2. Furthermore, the original Listing A.5 is added afterwards.

```java
model.addEntry(
  new WiringEntryBuilder("W3").
    guard(EntryType.getEntryType("request"), Address.PICAddress,
        Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
    service(
      new WiringEntryBuilder.ServiceBuilder("S3", Service2.class, serviceArgs).
      guard(EntryType.getEntryType("request"), Link.LinkOperation.TAKE,
          LinkQuery.ALL, LinkCount.EXACTLY_ONE)
    ).
    dest(new Address(new Address.PeerAddress("P1"), Address.WSCAddress)).
    build()
);
```

Listing A.6: Additional source code for benchmark 2b in `Benchmark1` class

Benchmark 2c. This benchmark evaluates the processing of *n* entries using one wiring while 20 other wirings are active on the peer. The code section of Listing A.7 adds the additional wirings; it is inserted into line 45 of Listing A.2. Furthermore, the original Listing A.5 is added afterwards.

```java
for (int i = 0; i < 20; i++) {
  model.addEntry(
    new WiringEntryBuilder("V" + i).
      guard(EntryType.getEntryType("A"), Address.PICAddress,
          Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
```

```
        dest(new Address(new Address.PeerAddress("P1"), Address.WSCAddress)).
        build()
    );
}
```

Listing A.7: Additional source code for benchmark 2c in `Benchmark1` class

Application logic

Listing A.8 shows the implementation of *S2* (and *S3* for benchmark 2b). For each processed entry, the service helper is called to increment the number of processed entries. Once the number of processed entries reaches n, the service helper determines the end time stamp and calculates the time since the start (cf. lines 32ff of Listing A.1).

```
public class Service2 implements ServiceExe {

  private final ServiceHelper serviceHelper;

  public Service2(List<Serializable> serviceArgs) {
    this.serviceHelper = (ServiceHelper) serviceArgs.get(0);
  }

  @Override
  public Collection<Entry> execute(Collection<Entry> entries) {
    serviceHelper.updateBulkTriggering(entries.size());
    return null;
  }
}
```

Listing A.8: `Service2` class

A.2 Communication benchmarks

For the communication benchmarks, the setting is shown in Figure A.2. Both instances are executed on the same machine, thus typical issues in networking such as latencies do not need to be taken into account. It utilizes the communication mechanisms by sending entries of type `request` to a remote instance, processing them there and replying with entries of type `response` to the original instance. Section A.2.1 shows the implementation of the local instance; Section A.2.2 the remote instance, respectively. The benchmark evaluates how long it takes until n entries of type `response` were received and processed at the sender instance.

A.2.1 Local instance

This section shows coordination and application logic of the local instance.

Coordination logic

Listing A.9 shows the coordination logic implementation of the local instance. It thus adds *P3* to the instance's *PSC* and afterwards the wiring *W3* to *P3*'s *WSC*. Before the

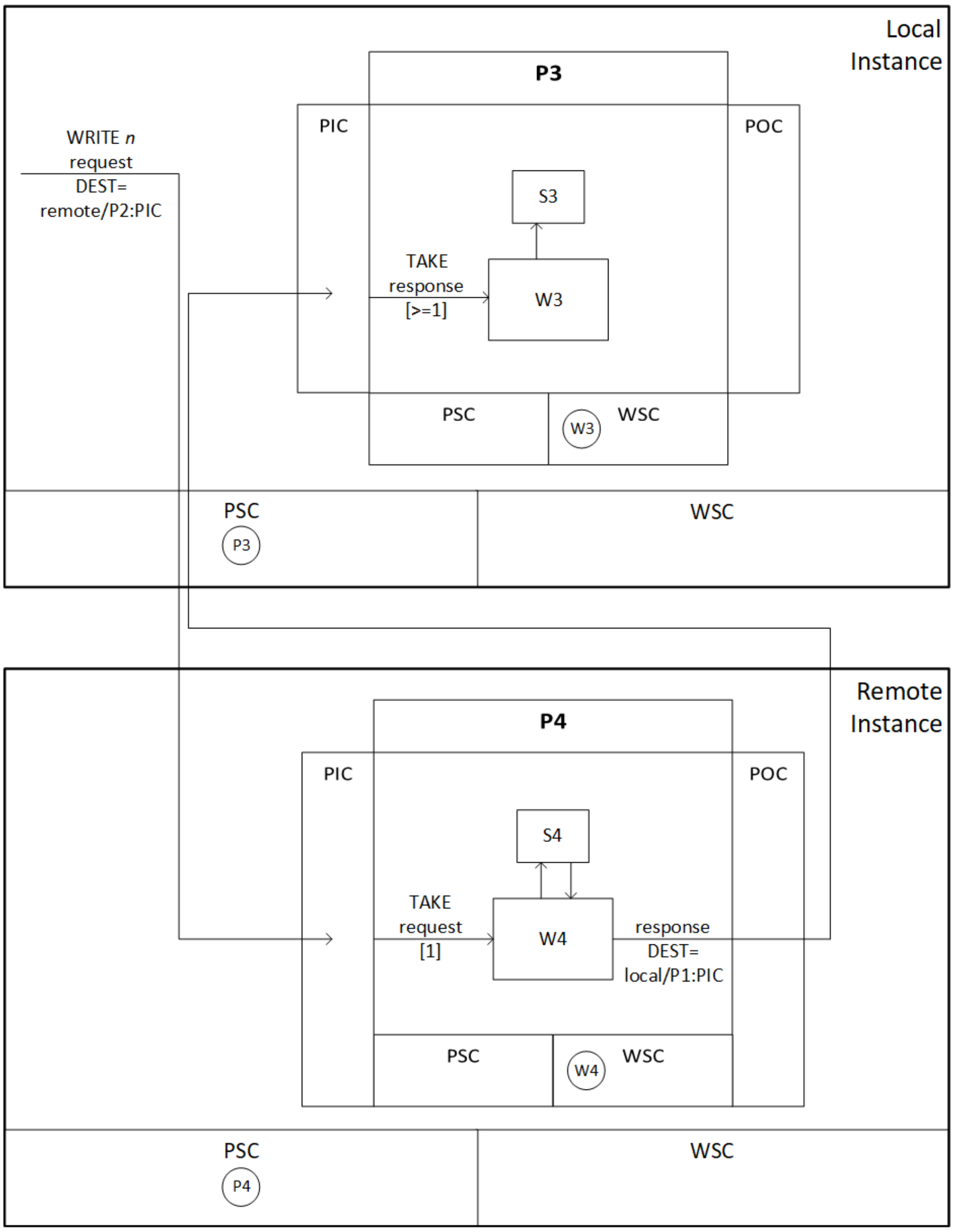


Figure A.2: Communication benchmarks setting

entries are added to the instance, the service helper is called to save the start time stamp (line 33). Subsequently, n entries are added to be transmitted to the remote instance (line 35ff).

```java
 1 public class Benchmark3A {
 2   private static final int N = ...;
 3
 4   public static void main(String[] args) throws InterruptedException {
 5     PeerModelInstance.initPeerModel("local", "config.properties");
 6     PeerModel model = PeerModelInstance.getInstance();
 7     model.start();
 8
 9     model.addEntry(
10       new PeerEntryBuilder("P3").
11         dest(new Address(Address.RUNTIME_PEER_ADDRESS, Address.PSCAddress)).
12         build()
13     );
14
15     ServiceHelper serviceHelper = new ServiceHelper();
16     List<Serializable> serviceArgs = Collections.singletonList(serviceHelper);
17
18     model.addEntry(
19       new WiringEntryBuilder("W3").
20         guard(EntryType.getEntryType("response"), Address.PICAddress,
21             Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.AT_LEAST_ONE, false).
22         service(
23           new WiringEntryBuilder.ServiceBuilder("S3", Service2.class, serviceArgs).
24             guard(EntryType.getEntryType("response"), Link.LinkOperation.TAKE,
25                 LinkQuery.ALL, LinkCount.AT_LEAST_ONE)
26         ).
27         dest(new Address(new Address.PeerAddress("P3"), Address.WSCAddress)).
28         build()
29     );
30
31     Thread.sleep(2000);
32
33     serviceHelper.startBulkTriggering(N);
34     for (int i = 0; i < N; i++) {
35       model.addEntry(
36         new EntryBuilder("request").
37           dest(new Address("remote", new Address.PeerAddress("P4"),
38               Address.PICAddress)).
39           build());
40     }
41   }
42 }
```

Listing A.9: `Benchmark3A` class – representing the local instance

In contrast to the guard of `W2` (`TAKE 1 request`, cf. Figure A.1), the guard of `W3` was changed (`TAKE` $\geq$ `1 response`, cf. Figure A.2) to test how often the wiring is triggered. It was shown that for 100, 1000, and 10000 entries in evaluation, the service is executed for ~ 80 % times of the entry count, i.e., that sometimes indeed `W3` is executed with more than one entry. These results are, however, not significant and were thus not included in the main text.

Application logic

The service implementation `S3` is reused from the general benchmark's `Service2` class (Listing A.8). There is no difference in its implementation except for working on entries of type `response` instead of `request`. However, the actual entry is not used in the service itself as it just counts the processed entries.

A.2.2 Remote instance

This section shows coordination and application logic of the remote instance.

Coordination logic

Listing A.10 shows the coordination logic implementation of the remote instance. It thus adds *P4* to the instance's *PSC* and afterwards the wiring *W4* to *P4*'s *WSC*.

```java
public class Benchmark3B {

  public static void main(String[] args) throws InterruptedException {
    PeerModelInstance.initPeerModel("remote", "config.properties");
    PeerModel model = PeerModelInstance.getInstance();
    model.start();

    model.addEntry(
      new PeerEntryBuilder("P4").
        dest(new Address(Address.RUNTIME_PEER_ADDRESS, Address.PSCAddress)).
        build()
    );

    model.addEntry(
      new WiringEntryBuilder("W4").
        guard(EntryType.getEntryType("request"), Address.PICAddress,
            Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE, false).
        service(
          new WiringEntryBuilder.ServiceBuilder("S4", Service4.class).
            guard(EntryType.getEntryType("request"),
                Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.EXACTLY_ONE).
            action(EntryType.getEntryType("response"))
        ).
        action(EntryType.getEntryType("response"), Address.POCAddress,
            Link.LinkOperation.TAKE, LinkQuery.ALL, LinkCount.ALL).
        dest(new Address(new Address.PeerAddress("P4"), Address.WSCAddress)).
        build()
    );
  }
}
```

Listing A.10: `Benchmark3B` class – representing the remote instance

Application logic

Listing A.11 shows the implementation of *S4*. For each call, an entry of type `response` is created and its destination set to the local instance.

```
public class Service4 implements ServiceExe {
  @Override
  public Collection<Entry> execute(Collection<Entry> entries) {
    return Collections.singleton(
      new EntryBuilder("response").
        dest(new Address("local", new Address.PeerAddress("P3"),
            Address.PICAddress)).
        build());
  }
}
```

Listing A.11: `Service4` class

Listings

List of Figures

List of Tables

Glossary

API Application Programming Interface. 3, 9, 15, 16, 45, 59, 61, 62, 65, 89, 91, 95, 103, 112, 127, 150

App-Data Application Data. 25, 27, 33, 76, 93, 95

Co-Data Coordination Data. 25, 27, 33, 37, 76, 93–95, 102, 107, 149, 150

CPN Colored Petri Nets. 7, 8

CPU Central Processing Unit. 11

DEST Destination Property (co-data). 27, 37, 90, 94, 95, 102

DOM Document Object Model. 59, 61

FIFO First In First Out. 15

FLOW Flow Property (co-data). 27, 33, 37, 95

ICMP Internet Control Message Protocol. 52

IDL Interface Description Language. 68, 69, 74, 77, 78

IETF Internet Engineering Task Force. 62

IGMP Internet Group Management Protocol. 52, 53, 56

IP Internet Protocol. 47–49, 52, 53, 55, 57, 58, 110

JAXB Java Architecture for XML Binding. 59

JCF Java Collection Framework. 87, 103

JDK Java Development Kit. 60, 110, 149

JEP JDK Enhancement Proposal. 61

JSON JavaScript Object Notation. 17, 38, 58, 59, 61, 62, 64, 69, 74, 75, 117

JSR Java Specification Request. 59, 61, 62

JVM Java Virtual Machine. 13, 14, 16

MLD Multicast Listener Discovery. 52

PIC Peer In Container. 28, 89, 93, 97

PKI Public Key Infrastructure. 10

PM-DSL Peer Model - Domain Specific Language. 2, 9, 39, 40, 130

PMDP Peer Model Discovery Protocol. 53

POC Peer Out Container. 28, 89, 93, 98

PSC Peer Specification Container. 37, 38, 89, 93, 96, 107, 108, 125

REST Representational State Transfer. 73

RFC Request for Comments. 52, 62

RMI Remote Method Invocation. 73, 76

RPC Remote Procedure Call. 66, 72–74, 76, 77, 112, 115

RTP Runtime Peer. 3, 37

SAX Simple API for XML. 59

StAX Streaming API for XML. 59, 61

TCP Transmission Control Protocol. 12, 53, 58, 110

TIL Trace Intermediate Language. 39, 40, 116, 117

TTL Time To Live Property (co-data). 4, 20, 23, 24, 27, 41, 43, 94, 95, 101–103, 107, 123, 130

TTS Time To Start Property (co-data). 4, 27, 41, 43, 94, 95, 103, 104, 107

UDP User Datagram Protocol. 55, 108–110

URI Uniform Resource Identifier. 13, 47, 49–51, 130

UUID Universally Unique Identifier. 26, 27, 94

VIL Visualisation Intermediate Language. 39, 40, 116, 117, 130

W3C World Wide Web Consortium. 59

WAN Wide Area Network. 48, 130

WSC Wiring Specification Container. 37, 38, 89, 93, 95, 97, 101, 107, 108, 125

WSDL Web Services Description Language. 18, 19, 74

XML Extensible Markup Language. 18, 58–62, 64, 74–76, 149–151

XSD XML Schema. 59, 60

XVSM eXtensible Virtual Shared Memory. 31, 74, 87

YAML YAML Ain't Markup Language (originally "Yet Another Markup Language"). 58, 64

Bibliography

[1] Protocol Buffers Documentation. `https://developers.google.com/protocol-buffers/`. [online, last accessed on 2019-01-29].

[2] vert.x Core Manual. `http://vertx.io/docs/vertx-core/java/`. [online, last accessed on 2019-01-29].

[3] The JSON Data Interchange Format. Technical report, ECMA International, 2013. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`. [online, last accessed on 2019-01-29].

[4] Hazelcast IMDG Reference Manual. Technical report, Hazelcast, Inc., 2018. `http://docs.hazelcast.org/docs/3.10/manual/pdf/index.pdf`. [online, last accessed on 2019-01-29].

[5] Aditya Agarwal, Mark Slee, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, 2007. `http://thrift.apache.org/static/files/thrift-20070401.pdf`. [online, last accessed on 2019-01-29].

[6] Gul Agha. An overview of actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, 1986.

[7] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[8] Gul Agha. *Abstractions, Semantic Models and Analysis Tools for Concurrent Systems: Progress and Open Problems*, pages 3–8. 2016.

[9] Gul Agha and Christian J. Callsen. Actorspace: An open distributed programming paradigm. *ACM SIGPLAN Notices*, 28(7):23–32, 1993.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[11] Joshua Bloch. *Effective Java*. Prentice Hall, 2 edition, 2008.

[12] F. D. J Bowden. A brief survey and synthesis of the roles of time in petri nets. *Mathematical and Computer Modelling*, 31(10-12):55–68, 2000.

[13] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, 2014.

[14] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol (IGMP), Version 3. RFC 3376, 2002.

[15] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[16] Stephan Cejka, Albin Frischenschlager, Mario Faschang, Mark Stefan, and Konrad Diwold. Operation of modular smart grid applications interacting through a distributed middleware. *Open Journal of Big Data*, 4(1):14–29, 2018.

[17] Stephan Cejka, Alexander Hanzlik, and Andreas Plank. A framework for communication and provisioning in an intelligent secondary substation. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2016.

[18] Stephan Cejka, Ralf Mosshammer, and Alfred Einfalt. Java embedded storage for time series and meta data in Smart Grids. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 434–439, 2015.

[19] Stephan Cejka and Matthias Schwayer. Visualization and Simulation of Wireless Sensor Networks in a Railway Crossing Use Case. Bachelor thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2013.

[20] 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*, pages 519–537, 2015.

[21] Maximilian Alexander Csuk. Developing an Interactive, Visual Monitoring Software for the Peer Model Approach. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2014.

[22] S. Deering, W. Fenner, and B. Haberman. Multicast Listener Discovery (MLD) for IPv6. RFC 2710, 1999.

[23] S.E. Deering. Host extensions for IP multicasting. RFC 1112, 1989.

[24] Malith Dhanushka. Comparison of Apache Thrift Java Servers, 2012. `http://mmalithh.blogspot.com/2012/10/comparison-of-apache-thrift-java-servers.html` [blog entry, last accessed on 2019-01-29].

[25] K. Zyp F. Galiegue. JSON Schema: core definitions and terminology. (Draft), 2013. `http://tools.ietf.org/html/draft-zyp-json-schema-04`.

[26] David C. Fallside and Priscilla Walmsley. XML Schema part 0: Primer. Technical report, W3C, 2004. `http://www.w3.org/TR/xmlschema-0/`.

[27] Mario Faschang, Stephan Cejka, Mark Stefan, Albin Frischenschlager, Alfred Einfalt, Konrad Diwold, Filip Pröstl Andrén, Thomas Strasser, and Friederich Kupzog. Provisioning, deployment, and operation of smart grid applications on substation level. *Computer Science - Research and Development*, 2016.

[28] Diwaker Gupta. Thrift: The Missing Guide. Technical report. `http://diwakergupta.github.io/thrift-missing-guide/thrift.pdf` [online, last accessed on 2019-01-29].

[29] Thomas Hamböck. Towards a Toolchain for Asynchronous Embedded Programming based on the Peer-Model. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2015.

[30] 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, 1973.

[31] Georg Holasek. Evaluation of the Peer Model Framework on a RCM4300 Evaluation Board. Bachelor thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2014.

[32] Vojin Jovanovic and Philipp Haller. The scala actors migration guide. `http://docs.scala-lang.org/overviews/core/actors-migration-guide.html`. [online, last accessed on 2019-01-29].

[33] Rajesh K. Karmani and Gul Agha. *Actors*, pages 1–11. 2011.

[34] Martin Kleppmann. Schema evolution in Avro, Protocol Buffers and Thrift, 2012. `http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html` [blog entry, last accessed on 2019-01-29].

[35] Eva Kühn. Reusable coordination components: Reliable development of cooperative information systems. *International Journal of Cooperative Information Systems*, 25(04):1740001, 2016.

[36] eva Kühn. Flexible transactional coordination in the peer model. In *Fundamentals of Software Engineering*, pages 116–131, 2017.

[37] eva Kühn, Stefan Craß, and Thomas Hambock. Approaching Coordination in Distributed Embedded Applications with the Peer Model DSL. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 64–68, 2014.

[38] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 121–135. 2013.

[39] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible Modeling of Policy-driven Upstream Notification Strategies. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1352–1354, 2014.

[40] eva Kühn, Sophie Therese Radschek, and Nahla Elaraby. Distributed coordination runtime assertions for the peer model. In *Coordination Models and Languages*, pages 200–219, 2018.

[41] eva Kühn, Johannes Riemer, and Gerson Joskowicz. XVSM (eXtensible Virtual Shared Memory) architecture and application. Technical report, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2005.

[42] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, 2005.

[43] Matthias Lettmayer. A Public Resource Computing Application based on the Secure Peer Model. Diploma thesis, Space Based Computing Group, Institute of Information Systems Engineering, TU Wien, 2018.

[44] Lightbend Inc. Documentation for akka 2.4.20. `http://doc.akka.io/docs/akka/2.4/AkkaJava.pdf`. [online, last accessed on 2019-01-29].

[45] Maurice Naftalin. *Mastering Lambdas: Java Programming in a Multicore World*. Oracle Press, 2015.

[46] Ted Neward. 5 things you didn't know about ... Java ObjectSerialization. `https://www.ibm.com/developerworks/library/j-5things1/j-5things1-pdf.pdf` [online, last accessed on 2019-01-29].

[47] OASIS. Web services business process execution language version 2.0, 2007.

[48] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, October 2005.

[49] Andrew Prunicki. Apache thrift. Technical report, 2009. `https://objectcomputing.com/resources/publications/sett/june-2009-apache-thrift/` [online, last accessed on 2019-01-29].

[50] Krzysztof Rakowski. *Learning Apache Thrift*. Packt Publishing, 2015.

[51] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. Cpn tools for editing, simulating, and analysing coloured petri nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 450–462, 2003.

[52] Dominik Rauch. Implementing and Evaluating the Peer Model with Focus on API Usability. Master's thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2014.

[53] Gerald Schermann. Extending the Peer Model with Compositional Design Patterns. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2014.

[54] Jörg Schoba. Mobile Peer Model – A mobile peer-to-peer communication and coordination framework - with focus on scalability and security. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2017.

[55] Michael Sperberg-McQueen, Eve Maler, Tim Bray, François Yergeau, Jean Paoli, and John Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). W3C recommendation, W3C, 2006. `http://www.w3.org/TR/xml11/`.

[56] Apache Thrift Language Support. Comparison of Apache Thrift Java Servers, 2018. `https://thrift.apache.org/docs/Languages` [online, last accessed on 2019-01-29].

[57] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. 2006.

[58] Peter Tillian. Mobile Peer Model – A mobile peer-to-peer communication and coordination framework - with focus on mobile design constraints. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2017.

[59] R. Vida and L. Costa. Multicast Listener Discovery Version 2 (MLDv2) for IPv6. RFC 3810, 2004.

[60] Stefan Zischka. A Coordination-Based Framework for Routing Algorithms in Unstructured Peer-to-Peer Networks. Diploma thesis, Space Based Computing Group, Institute of Computer Languages, TU Wien, 2017.