FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

PeerSpace.NET

Implementing and Evaluating the Peer Model with Focus on API Usability

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Dominik Rauch
Matrikelnummer 0825084

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eva Kühn
Mitwirkung: Projektass. Dipl.-Ing. Thomas Scheller

Wien, 23.04.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

PeerSpace.NET

Implementing and Evaluating the Peer Model with Focus on API Usability

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dominik Rauch
Registration Number 0825084

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eva Kühn
Assistance: Projektass. Dipl.-Ing. Thomas Scheller

Vienna, 23.04.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Dominik Rauch
Lilienbrunngasse 13/1/13, 1020 Wien

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

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I would like to thank several people for their contributions to this thesis. First of all, I thank my professor Eva Kühn and her research assistant Thomas Scheller for enabling me to work on the topic, for their useful and constructive feedback and their guidance during my work. Furthermore I'd like to thank my girlfriend and my family for their huge support during the previous year.

Abstract

The Peer Model is a new design approach for distributed software systems. In contrast to other modeling languages it allows to model even complex systems in a very concise and sharp way without being tailored for a specific problem domain. It gains its strengths from a well-defined underlying space based framework, which is able to guarantee several system properties, so that the architect may focus on the actual coordination pattern design. Up to now there has not been a concrete software implementation available which would enable developers to transform Peer Models into executable software components. It has therefore been crucial for the practical use of the Peer Model to provide such an implementation - the PeerSpace.NET framework in hand.

Earlier frameworks in the area of space based computing faced criticism in the area of API usability for developers. Denoted as “hard to learn”, it has been set as a major goal to build a framework with high API usability, offering the same comfort to developers as the Peer Model is providing to software architects. Therefore a fluent interface has been created on top of the core framework to support developers in their work. An extensive evaluation using two completely different methods from the area of qualitative and quantitative API usability assessment ensured that the solution indeed fulfills the given requirements.

Altogether, the PeerSpace.NET framework allows developers to build scalable distributed software components based on the new Peer Model approach in a convenient and error-preventive way.

Kurzfassung

Das Peer Modell ist ein neuartiger Ansatz, um verteilte Softwaresysteme zu designen. Im Gegensatz zu bestehenden Modellierungssprachen erlaubt es auch sehr komplexe Systeme präzise und unkompliziert zu beschreiben ohne auf eine konkrete Problem-Domäne zurechtgeschneidert worden zu sein. Es gewinnt seine Stärke durch das darunterliegenden Space Based Framework, welches bereits eine Reihe von Eigenschaften garantiert, und es der Architektin bzw. dem Architekten ermöglicht, sich auf das eigentliche Koordinationsmuster zu fokusieren. Bis jetzt existierte allerdings noch keine Software-Implementierung, welche es EntwicklerInnen erlaubt, bestehende Peer Modelle ohne Umwege in ausführbare Software-Komponenten zu transformieren. Es war deshalb ein wichtiger Schritt für den praktischen Einsatz des Peer Modells, eine solche Implementierung bereitzustellen - das vorliegende PeerSpace.NET-Framework.

Frühere Frameworks, welche auf dem Prinzip von Space Based Computing basierten, mussten sich mit Kritik im Bereich API-Usabilty auseinandersetzen. Es war deshalb ein wichtiges Ziel dieser Diplomarbeit ein Framework zu kreieren, welches den EntwicklerInnen denselben Komfort zur Verfügung stellt, welchen das Peer Modell den SoftwarearchitektInnen bereitstellt. Aus diesem Grunde wurde ein sogenanntes “fluent interface” geschaffen, welches auf Basis des Kern-Frameworks die Entwicklerin bzw. den Entwickler in seiner Arbeit bestmöglich unterstützt. Eine eingehende Evaluierung unter Verwendung zweier vollkommen verschiedener Methoden aus den Bereichen qualitativer und quantitativer API-Usability-Bewertung zeigte, dass die gefundene Lösung den Anforderungen entspricht.

Zusammengefasst erlaubt das PeerSpace.NET-Framework EntwicklerInnen skalierbare, verteilte Softwarekomponenten in einer bequemen und fehlervermeidenden Umgebung auf Basis des neuen Peer Modell-Ansatzes zu erstellen.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

In today's interconnected computing world, information acquisition, processing and exchange takes place at a global level. Distributed software systems utilize these capabilities and instruct multitudes of standalone processes to collaborate in a highly dynamic manner. Designing and implementing such systems is a very difficult task. Software architects and developers have to face high complexity and many hidden pitfalls. Lots of non-functional requirements like scalability or reliability have to be considered additionally in order to create effective and efficient distributed systems. Research and industry, both started to define common patterns of coordination to reduce the overwhelming complexity. Software frameworks, so-called middleware solutions, have been created to encapsulate frequently occurring functions like transaction management, security, replication and many more. Additionally, modeling languages have been invented to optimally support the design process, even allowing us to give mathematical proofs for some of a system's properties.

While historically client/server architectures, where a central host performs most of the computational work, are very popular, they usually do not offer enough flexibility for modern, increasingly collaborating systems, where each participating node may act as a server and a client at the same time (peer-to-peer systems). Space based middleware implements the peer-to-peer approach by allowing standalone processes to communicate via shared data. Figuratively speaking, peers can post, read, take and remove information from virtual blackboards. Initially introduced in the form of the Linda tuple space [4], the technology has since evolved and resulted in the XVSM [5] (eXtensible Virtual Shared Memory) framework. XVSM enables modern space based computing for C and Java.

Modeling languages have evolved as well. The last decade has witnessed the upcoming of a range of tailored modeling tools for specific application scopes. The Space Based Computing research group, part of the Institute of Computer Languages of the Technical University of Vienna, is currently concentrating on the evermore complex field of modeling distributed software systems as well. In 2013 they presented a new way of modeling coordination patterns and distributed systems, the so-called Peer Model approach [1]. In comparison with currently existing modeling tools, the Peer Model is able to gain simplicity by utilizing a well-defined underlying

space based computing framework. The framework already provides services to overcome many of the problems and obstacles faced by software architects. This allows the software architect to concentrate on the coordination between the distributed components, while non-functional properties are either inherent in the Peer Model (e.g. scalability), or manifest from the underlying framework, e.g. transaction-safety or security. Furthermore, the Peer Model achieves a strict segregation of coordination and domain logic and allows to design distributed software systems without incorporating the data processing logic, thus removing any complicating entanglement.

Currently, the Peer Model is a modeling tool for software architects only. As there is no full-featured software implementation available, the transformation of Peer Models into distributed software components still constitutes a difficult task. It is therefore crucial for the practical use of the Peer Model to provide a software implementation - the PeerSpace.NET framework, which is the topic of this thesis. It enables developers to implement Peer Models straightforward and to transform coordination patterns directly into runnable distributed systems.

On another note, software development complexity can also be reduced by improving the interface of middleware solutions. High API usability helps the users of the corresponding framework, the software developers, to understand the framework more profoundly, thus simplifying the implementation of software components and increasing the quality of the resulting product [6]. In the past, space based technology frameworks have been criticized as "hard-to-use" or "not beginner-friendly", which is why the Space Based Computing research group recently got into the (sparse) research field of API usability. Therefore, the second step of the thesis is to incorporate new insights of this research area and design a developer-friendly API for the PeerSpace framework.

While some evaluation tasks and comparisons can be made using the theoretical Peer Model only, the results of this thesis allow to evaluate more of the practical advantages of the Peer Model, which will help to establish it as a future-proof and enterprise-ready modeling tool for distributed software systems. Altogether, the aim of the thesis is to create the very first comprehensive implementation of the Peer Model with a special focus on API usability.

1.1 Methodology & Expected Results

The methodological approach of this thesis is driven by the aim to create the very first implementation of the Peer Model using .NET and at the same time providing an API with high usability. Furthermore, the resulting API shall provide a basis for future Peer Model implementations in Java or C++. In this section we present the applied methodologies step by step as well as the expected results of each part.

1.1.1 Literature Review

In the beginning an extensive literature review will bring up the state-of-the-art in the following research areas: coordination pattern modeling, the Peer Model itself, space based computing as an underlying framework, alternative solutions for distributed software systems that are currently used in the industry and last but not least the state-of-the-art in the field of framework API design, API usability and API usability evaluation. Although not firsthand required, the literature review

will also look into related work to emphasize the differences and work out the strengths of our approach. This will help later on to make the correct design decisions for the PeerSpace framework.

1.1.2 PeerSpace Core Design & Implementation

After examining the relevant literature, we gather all functional and non-functional requirements of the PeerSpace core framework. The feature-complete core shall enable developers to transform Peer Models into executable code. The focus will not yet be on developer-usability, instead, special attention is given to modern object-oriented framework guidelines. Clear separation of interfaces and their implementations as well as strict usage of the dependency injection pattern allows to replace individual components of the PeerSpace framework easily with more advanced versions later on. A focus on extensibility also ensures that the PeerSpace can adapt if the Peer Model itself is altered by feedback gained in the practice. Although it is not necessarily developer friendly yet, the core framework shall enable possible future automatic code generators to transform Peer Models into PeerSpace code.

Another important point is the integration of the underlying space framework. The space framework defines many of the PeerSpace's non-functional properties by providing the means of information exchange between peers. It is important that developers are able to change the configured space framework easily: choosing a framework, depending on the given requirements, defines which features should manifest in the resulting solution. E.g., it should be possible to use a non-secure and non-reliable space implementation in order to achieve a performance-only-focused solution without changing any of the code written by the developers targeting the PeerSpace framework.

The expected goals of this part are: gathering all functional requirements from the original Peer Model definition, pointing out non-functional requirements (derived mostly from the Space Based Computing group's research), and designing the software architecture for the PeerSpace framework. Last but not least the full-featured core implementation needs to be created, on which all further work in this thesis is based.

1.1.3 PeerSpace API Design & Implementation

Based upon the core implementation we add the usability-focused API. It shall enable the developer to do the majority of the tasks in an easy and immediately comprehensible way. This not only improves development speed but also increases code maintainability and code reuse. The API shall still be able to access the core framework for the most advanced or for very unusual tasks which are not directly supported by the PeerSpace API.

In order to create a modern interface to the PeerSpace with high API usability we combine the latest research results from the literature review with the requirements of the PeerSpace configuration. We compare various possible approaches, choose the best suitable design and in the end present the final API. Furthermore, an example will show how to use the PeerSpace API, which is an important step to train developers and to let them immediately profit from the PeerSpace framework.

1.1.4 Evaluation

At the end of the thesis, the resulting PeerSpace API is evaluated. Qualitative and quantitative approaches will be used to evaluate the API usability of the PeerSpace framework. Among them: the cognitive dimension framework [7] as well as a newly developed automated measurement method for API usability from the Space Based Computing research group [8].

We will apply the same evaluations to an alternative middleware solution as well, in order to compare the PeerSpace framework with its competitors. For the comparison, we will use a framework with the same technology stack (i.e. .NET). As most of the currently used middleware solutions follow the Service Oriented Architecture (SOA) principles, we chose a SOA middleware for our evaluation.

The goal is to determine if the newly designed framework is in fact easier to use and better suited to implement complex distributed interactions than alternatives. The gained feedback will be applied to improve the PeerSpace continuously.

1.2 Structure of the Master's Thesis

The thesis is structured roughly along the succession of methodologies which have been enumerated in the previous section. It starts with the introduction (chapter 1), which gives the motivation behind the thesis, describes the problem at hand, the aim of the thesis, the methodologies and the expected results. Chapter 2 describes the state-of-the-art and important related work. It presents coordination pattern modeling in general, the Peer Model, space based computing, .NET communication middleware solutions and the state-of-the-art in the area of framework usability. Chapter 3 gives a short presentation of the applied technologies, which have been used to implement the PeerSpace. Chapter 4 collects all requirements, functional as well as non-functional, which build the base for the PeerSpace core design and implementation. The actual implementation of the core framework is then discussed in chapter 5, including its architecture as well as presentations of essential parts of the code. The usability-focused API follows in chapter 6, it evaluates various API options and presents the final PeerSpace API. It also includes an example scenario. The qualitative and quantitative evaluation of the API's usability is presented in chapter 7. The thesis ends with the conclusion in chapter 8, also discussing the future road map of the framework, pointing out open issues as well as possible modifications to achieve certain non-functional requirements like transaction-safety.

CHAPTER 2

State of the Art & Related Work

This chapter presents the current state of the art in all the areas relevant for implementing the thesis. Among the covered research fields are: coordination modeling in distributed systems, in particular the Peer Model approach, as well as the field of API usability for frameworks. Furthermore, we point out important related work, especially alternative .NET middleware solutions. This will support the identification of the Peer Model's specific advantages, which helps to focus on its strengths during design and implementation of the PeerSpace.

2.1 Coordination Modeling in Distributed Systems

Advanced distributed software systems unite a multitude of concurrently running components to form one consistent piece of software, solving one or even multiple tasks within a given problem domain. For achieving this goal it is necessary to integrate various, possibly heterogeneous, information sources (e.g. sensory data or traffic data providers), which are delivered by other distributed software systems [1]. Those other systems are entirely independent from the original system and are possibly designed with different goals in mind.

While coordinating the integration of such a large amount of processes is already hard enough, the underlying nature of distributed systems increases the complexity of designing such systems even more by adding a great deal of exceptional corner cases and pitfalls in system communication. Among them are the well-known eight fallacies of distributed computing [9]:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.

6. There is one administrator.

7. Transport cost is zero.

8. The network is homogeneous.

Well-designed, robust and scalable distributed systems need to tackle each one of those fallacies in order to build systems which are able to guarantee consistent behavior under all circumstances. One can easily imagine that the design, implementation, analysis and control of distributed concurrent systems is therefore an extraordinarily complex task [1].

Defining the coordination structure of a large distributed system as one monolithic giant is not reasonable. Designers are encouraged to decompose the system into smaller distributed parts, optimally representable by existing, industry-proven coordination patterns. Nevertheless, the application of modeling techniques is necessary to control complexity.

The first part of this chapter will show current modeling techniques for distributed systems. Afterwards we will present the new Peer Model approach and compare it with already existing techniques to find out its strengths and advantages. This enables us to focus on the important areas when implementing the Peer Model.

2.1.1 Colored Petri Nets (CPN)

Colored Petri nets are most probably the best known modeling language for distributed systems today. Basically, Petri nets are a mathematical model for the description of distributed systems. More precisely, a Petri net is a directed bipartite graph in which nodes represent transitions (signified by bars) and places (signified by circles). Arcs connect a place with a transition or vice versa [10]. Tokens are markers, which can be put on places and are then transported through the system via transitions.

Like other industry standards (e.g. UML activity diagrams), Petri nets offer a graphical notation for stepwise processes that include choice and iteration as well as concurrent execution [10]. Figure 2.1 presents a small example. The transition t activates ("fires") whenever there are at least two markers on p1 and one marker on p2. When t fires it will put two markers into p3. After the transition t fired, the Petri net looks like shown in figure 2.2. This graphical language is supported by tooling applications which help the designer to easily model a concurrent system. The underlying mathematical definition can then be used to proof various properties (e.g. liveness of transitions) of the created system.

Colored Petri nets are an extension to Petri nets where each token is additionally assigned a color, indicating the identity of the token. Also places and transitions have an attached set of colors. A transition can fire with respect to each of its colors, moving only tokens of the corresponding color. The transition may also change the color of the involved tokens [11].

Typically Petri nets are used for system testing and performance analysis [12]. However, the abstraction level of Petri nets is very low-level. On the one hand this enables users to model systems for many problem domains, on the other hand it does not provide any higher-level facilities to overcome complexity in large systems. [1] shows that even for minor service-oriented architecture patterns corresponding Petri nets may result in very complex graphical models (see figure 2.3). If we increase the number of services, the Petri net becomes even more complex.

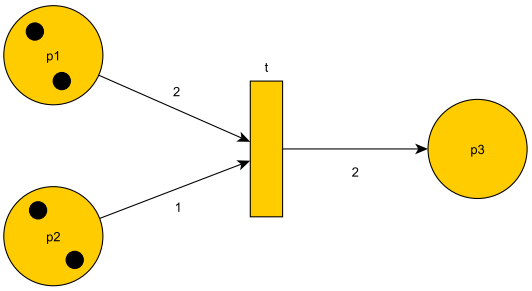

Figure 2.1: A simple Petri net (before firing)

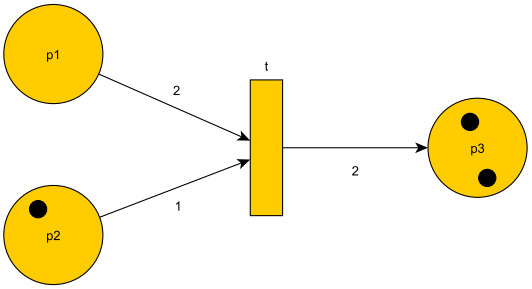

Figure 2.2: A simple Petri net (after firing)

In summary we can see that component design with Petri nets does not scale very well, because each new component must be modeled explicitly.

2.1.2 Other Modeling Languages

In addition to colored Petri nets there are many other modeling languages on the market. In this section we present the most common ones and afterwards summarize their advantages and disadvantages. The selection of languages is based upon the evaluation work in [3].

OMG MARTE

Modeling and Analysis of Real-Time and Embedded Systems (short: MARTE) is a UML extension developed by the Object Management Group (OMG) [13]. It adds timing information to UML diagrams and allows specifying timing constraints. Though UML is a widely adopted industry standard, the MARTE extension is not perfectly suitable for the given problem domain of distributed, parallel software systems. UML modeling of states is limited to local transitions via state charts which makes it very difficult to analyze concurrency behavior [14].

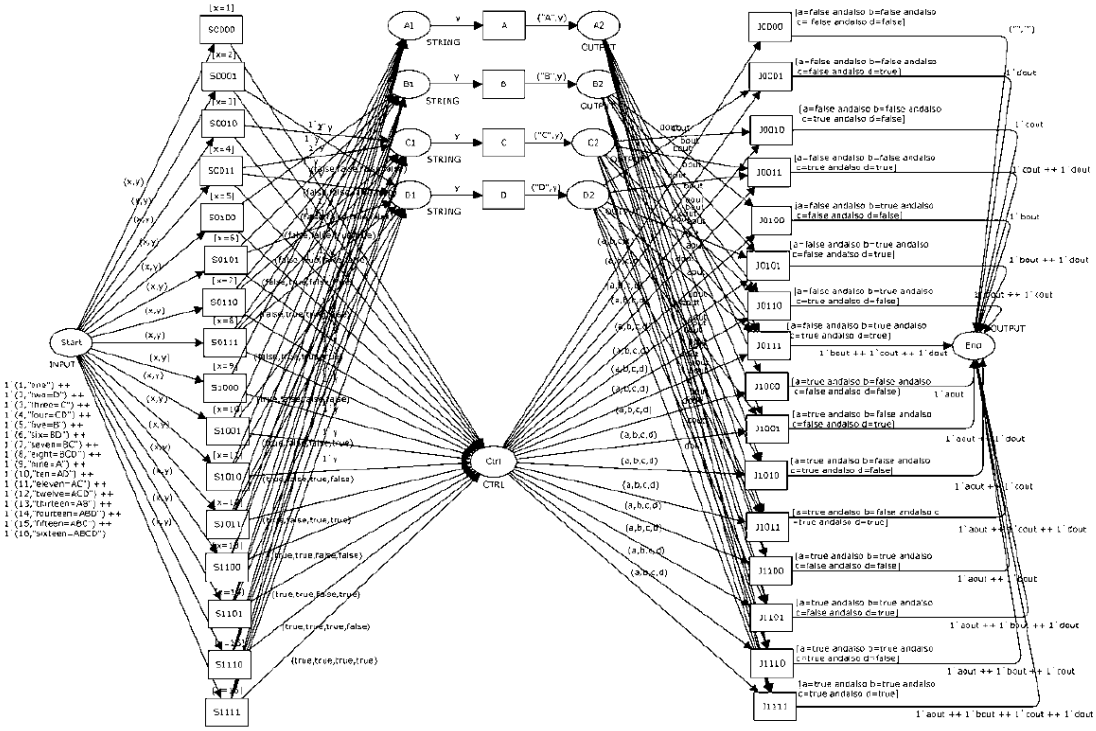

Figure 2.3: Simple Split/Join pattern with four services resulting in a very complex CPN (picture taken from [1])

REO

REO is a modeling tool suited for service-based business workflows [15]. The main aim is to model all aspects of such a business process, so that the model can then be verified and optimized by the toolset [14]. The designer models components and services via a channel-based modeling language which (in contrast to Petri nets) is data driven and supports time semantics. Pre-defined coordination primitives such as filter, transform or FIFO allow specifying when and how data is transferred between services and components.

Workflow Languages

Various workflow languages like BPEL, BPMN or YAWL [16] also claim to be suited for modeling coordination between services. However, very much like OMG MARTE, those modeling languages are intended to design enterprise business processes and are not geared towards distributed coordination [14]. This is also the topic of [17], which states that workflow languages need yet to be extended by so-called event stream processing units to overcome disadvantages in the area of decoupling and decentralized coordination.

Actor Model

The Actor Model [18] describes the eponymous "actor" as its fundamental unit of computation. Each actor consists of three parts: processing, data storage and communication capabilities. Actors communicate asynchronously via messages. After receiving a message, an actor can respond with three different actions: it may create a number of additional actors, send messages (including messages to itself) and influence its behavior by specifying how the next received message should be handled (i.e. change its state). The Actor Model had direct influence upon the development of some programming languages, amongst them Erlang and Scala. In contrast to complex workflow languages like WS-BPEL it is a much simpler model, however, it has no way to incorporate time as a factor: messages may take arbitrarily long to reach a receiver and there are no timed interrupt capabilities, e.g. to resend a message.

2.1.3 Summary

The previous chapter has given an overview over the most commonly used modeling languages. Petri nets and their more expressive colored counterpart are a modeling tool for general tasks, capable of designing any distributed system as of today. However, the low-level approach lacks design scalability and mixes coordination with application logic, two important properties for modeling complex systems. The Actor Model is an alternative general modeling language, which is very advanced in its features, but it fails to incorporate the factor of time. Specific needs in various areas of distributed computing have therefore led to the development of more tailored modeling tools like REO or workflow languages like BPEL.

Gerald Schermann, in his thesis [3], performed an in-depth evaluation of the most common modeling languages and compared them in regards to ten criteria. Table 2.1 presents the summarized results, already including his evaluation of the new Peer Model approach.

	CPN	**REO**	**BPMN**	**WS-BPEL**	**Actor Model**	**Peer Model**
Composition	+	+	+	+	+	+
Reuse	+	+	+	+	+	+
Parametrization	∼	+	+	+	+	+
Sep. of Concerns	−	+	+	+	+	+
Dynamics	−	+	∼	+	+	+
Addressing	−	∼	∼	+	+	+
Scalability	−	−	+	+	+	+
Time	∼	∼	∼	∼	∼	+
Toolchain & Docs	∼	∼	+	+	∼	∼
Simplicity	+	−	∼	−	+	+

Table 2.1: Summarized evaluation results of modelling languages (taken from [3])

2.2 Space Based Computing

Middleware frameworks for distributed systems simplify the communication and coordination of components which have to collaborate across machine and network boundaries. The aim is to simplify development and to reduce the need of "reinventing the wheel" for various different problems of distributed computing. A common approach is to make a lot of tasks and challenges transparent to the developer, e.g. a component's location, replication, migration or access control are decoupled from the actual component behavior. Furthermore the middleware provides services like transaction management, reliability services, load balancing and many more. Most middleware frameworks utilize the concept of message passing via queues in order to exchange commands and data between processes. The main disadvantage of the FIFO-based client/server approach is that it is not well-suited for all tasks. Imagine multiple waiters in a restaurant. All of them bring orders to the kitchen: of course a simple FIFO algorithm would not result in a very efficient kitchen [2]. It is therefore favorable to support multiple communication paradigms.

The concept of space based computing has been developed by David Gelernter in the form of Linda tuple spaces [4]. He introduced shared data spaces, one can picture such a space like a blackboard where entries are put to and read or taken from. Autonomous components are then able to use the information on the blackboard, process the information, put additional information there, etc. The concept is easily scalable by adding new processes, there is no need of a special coordination process and it allows a multitude of communication paradigms instead of just a few [2].

Typically space implementations use a centralized server as their shared space (e.g. the Tuple Space framework or JavaSpaces [19]). However, this creates a single point of failure and does not support scalability. Furthermore, information from the space can only be received by means of a single concept termed "template matching", i.e. processes block until an entry has been put into the shared space that looks like the provided template entry. This approach often requires developers to write complex code instead of utilizing a simple query language.

Due to this drawbacks, the Space Based Computing group at the Institute for Computer Languages has started to develop space based systems which do not rely on a central server but instead create a virtual server on the basis of distributed shared memory [20]. The resulting framework is called XVSM (eXtensible Virtual Shared Memory) [5]. It features pluggable and extensible function profiles on the basis of a space core (see figure 2.4). The core contains the distributed shared memory which is further divided into so-called containers. A container is the blackboard-like structure into which entries can be put. XVSM features various deployment scenarios, most importantly, space instances are able to replicate between each other and therewith solve the single point of failure problem. Furthermore, it allows to utilize multiple coordination patterns in addition to the Linda tuple matching approach, i.e. FIFO or key-based selection. Automatic discovery of XVSM instances in the same network is also already built-in. On top of the XVSM core, many plug-able function profiles can be deployed. They provide functionality which is usually found in other middleware solutions as well. Over the last years, the XVSM framework has proved itself successful in various application scenarios (e.g. see [21] or [22]).

Space based computing is the underlying concept of the Peer Model. As we will see in more detail in the next section, the space based framework provides various services to the Peer Model,

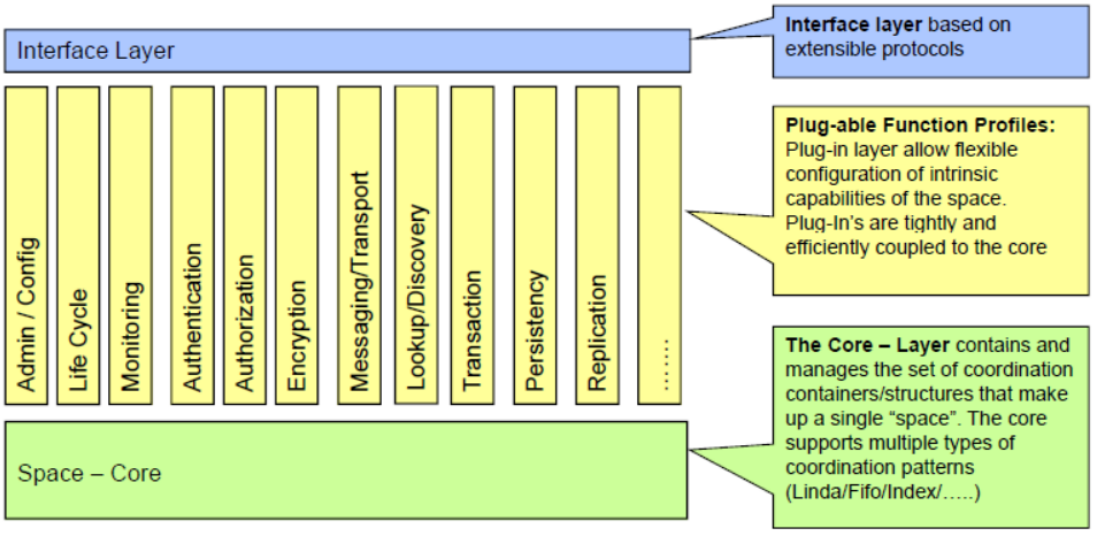

Figure 2.4: The XVSM architecture (picture taken from [2])

such that the architect may concentrate on the coordination between the distributed components (the so-called peers) only. All the guarantees of the space framework can be taken for granted and thereby greatly simplify the required design work.

2.3 Peer Model

This section briefly summarizes the Peer Model that has been introduced in [1]. In the initial section we introduced common methodologies for modeling concurrent and/or distributed software systems. We have seen that many modeling tools follow one of two approaches. Either the designer *can model everything*, however, the designer also *has to model everything* [1] or the modeling tool is tailored to the very specific requirements of a single problem domain. The Peer Model is founded on space based coordination principles, which allows developers to model in an environment with a higher level of abstraction. Furthermore, a clear separation of the coordination logic and services (data logic) enables designers to concentrate on the coordination model only.

2.3.1 Peers and Containers

The basic building block of the Peer Model that was introduced in [1] is the eponymous Peer. It represents a structured, re-usable and addressable component. The graphical notation of a peer can be seen in figure 2.5. It is represented by a central rectangle displaying its behavior (which we will discuss later), a rectangle on the top stating its name, and two more rectangles on the side representing the peer's inbound container (PIC) and its outbound container (POC). Between them the internal logic of the peer takes place: data enters the peer in the PIC, is used and/or transformed by its behavior and put into another container (typically the POC) in the end.

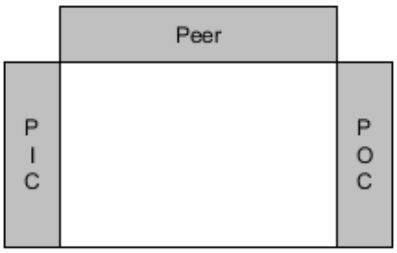

Figure 2.5: Graphical notation of a peer

A single peer instance always runs on a single computing resource, high scalability is reached by running multiple peer instances on multiple computers. This also dictates that peers can only have local state, but are otherwise standalone components which are only defined by their behavioral logic and the incoming data. If stateful peer behavior is required, peers have to exchange data with a so-called "space peer", a special type of peer with no own behavior and only a single container, where data may be posted like on a blackboard.

So-called wirings model the peer's internal behavior as well as communication between peers, we will discuss this in the next sub section. They describe how the data (modeled as entries) are routed through the peer network. Entries may have application- as well as coordination-specific data. The latter holds additional system-internal information like time-to-live meta-information or transaction identifiers. Coordination data is managed by a label/value store and is extensible - a good extension point for add-ons.

A very important aspect of the Peer Model is the possibility to use peers within other peers as sub components (called sub-peers), as initially proposed in [1]. The theoretical model goes one step further and places all peers inside a so-called universal peer which is hosting all other peers and thus explaining remote inter-peer wirings. The option to use peers as predefined components within bigger structures is a vital feature: it enables the Peer Model to scale for large systems by incorporating already proven patterns into bigger components without adding complexity.

2.3.2 Wirings

Wirings are the active part of the Peer Model. Data (so-called entries) are routed through the peer network according to the wiring rules. In other words: wirings describe when, which and how many entries are moved between PIC and POC containers. All wirings consist of three parts:

1. A **guard** (consisting of one or more guard links)
2. Zero or more **service calls**
3. An **action** (consisting of zero or more action links)

The graphical notation of a wiring can be seen in figure 2.6. It displays the wiring within an example peer as a rectangle (W1) - which is also, much like a peer, framed by its components. Arrows pointing from a container to the wiring represent guard links, the link's details can be found on top of the arrow. On top of the wiring we can see the connected services. Arrows pointing away from the wiring represent action links. A detailed description of how a wiring works follows shortly. Beforehand some additional notes: two special guard links exist: the star (*) "init guard" and an X "shutdown guard" are "fulfilled" so to say, when the system starts or shuts down respectively. See the second wiring W2 for an example of an init guard. Furthermore, action links can be found on the left side as well, if the emitted entries are put into the PIC instead of the POC container.

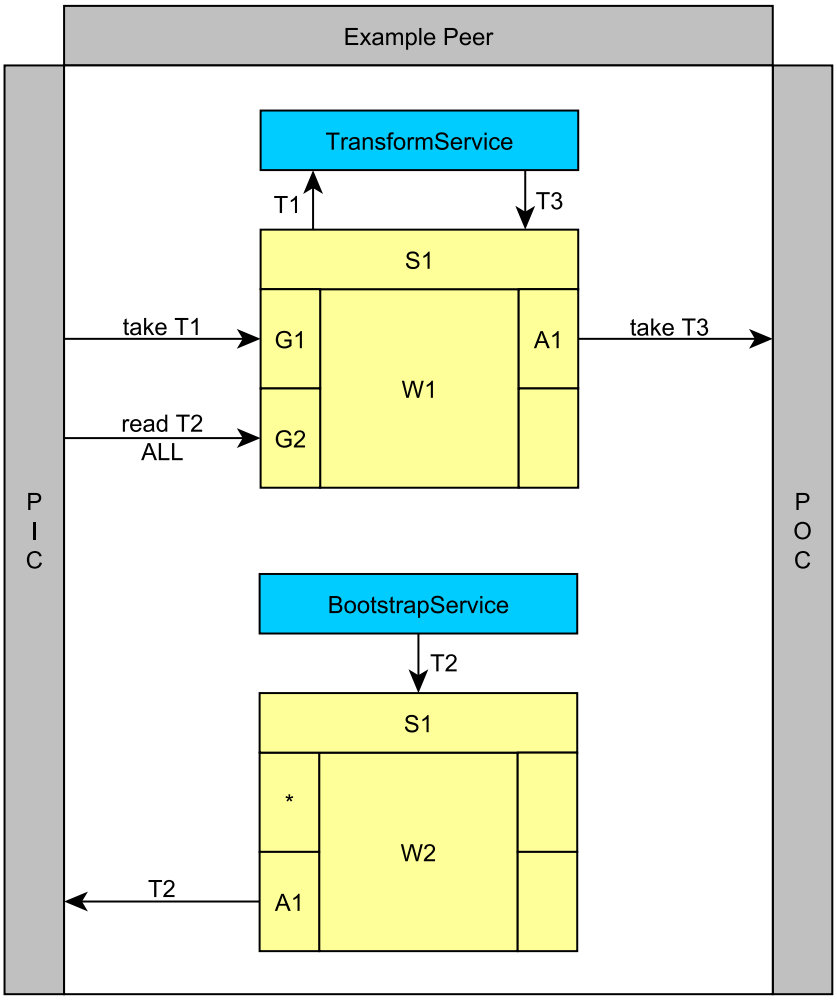

Figure 2.6: Graphical notation of wirings

The description on an arrow describes an entry query clause. For example, the guard link G1 says: take a single entry of type T1. The second guard link G2 says: read all available entries of type T2. A wiring always takes as many entries out of the container as possible, i.e. “take one or more” is equivalent to “take all available but at least one”. Note also, that simply writing a type on an arrow is equivalent to specifying “take one item of type T”.

The wiring behavior is as follows: whenever all of the guard’s links can be fulfilled simultaneously, the wiring fires. All selected entries are put into the entry collection, a container-like bag for entries which lives only during the execution of the wiring. All connected services are now called sequentially and are allowed to work with the entries in the entry collection, take entries out of it and also put new entries into the entry collection (see the arrows between the S1 box and the actual service). After all services have completed their work, the system tries to fulfill the action links of the wirings’s action. In contrast to the wiring’s guard however, action links don’t need to be satisfied. In case one link cannot be fulfilled, it is ignored and the next action link is executed. All remaining entries in the entry collection are discarded in the end. Altogether, the fired wiring has transported entries from one container to another, possibly processed/transformed them in the meantime and probably even created some new entries. As you can see, wirings concentrate on the coordination logic only, the data processing is outsourced to services which are completely independent of the Peer Model. This enables high decoupling and provides a strong focus on the communication between peers.

If transactionality is enabled, each wiring is associated with a wiring transaction. The whole wiring is executed in an atomic step with a single commit at the end. However, the wiring may decide to commit the withdrawal of entries from the in-container immediately after the guard or also after the service phase in order to remove locks and allow higher concurrency. All wirings whose guards are fulfilled may run concurrently, also the model allows a single wiring to fire multiple times in parallel. An implementation of the Peer Model is allowed to add possibilities of enforcing a maximum grade of parallelism if desirable.

An example of a complete coordination pattern designed with the Peer Model can be seen later on, e.g. in section 6.5.

2.3.3 Flows

A flow is the entirety of all wirings that together constitute a business process [1]. A flow is started by an initial entry bearing a special flow ID. Flow IDs trigger a behavior very much like the color in colored Petri nets - only entries with compatible flow IDs can fulfill a wiring’s guard (note: entries without a flow ID are compatible with all flows). All further entries created by such a wiring execution are also bearing the same flow ID. Flows may finish through either an explicit success or failure result entry or after reaching a maximum time-to-live. Eventually all entries belonging to this flow will be recognized by wirings and automatically removed. This allows architects to model concurrent business processes with the help of the Peer Model.

2.3.4 Summary

In this chapter we examined the fundamentals of the Peer Model and its graphical notation. We can see some major advantages over other modeling tools which are:

Design Scalability
In contrast to colored Petri nets, the Peer Model scales naturally to any number of services without exponential complexity.

Composability
The Peer Model allows to break down problems into smaller, re-usable components.

Transparent Remoting
The physical distribution of processes does not cause a difference in design compared to local processes [1].

Furthermore it is less general than colored Petri nets and takes specific assumptions on its domain. It conceptionally assumes an underlying space based middleware - whose features define many capabilities (the "quality") of the resulting Peer Model system.

2.4 Alternative Communication Frameworks for .NET

In the previous sections we discussed the Peer Model as a modeling language and compared it to related alternatives. In this section we take a look on the implementation-side: what are the existing .NET middleware and communication framework alternatives, what are their specific advantages and disadvantages, etc. Four areas are of particular interest for this thesis:

Support for complex coordination patterns
We're looking for a coordination middleware which is able to implement complex coordination patterns. It should particularly not be restricted to a few predefined patterns. The middleware should also include support for asynchronous communication.

Component composition & Reuse
The middleware should support composition of existing components. This allows to reuse existing coordination patterns within a bigger system and prevents "reinvention of the wheel"-solutions.

Maturity & Feature completeness
The framework should be enterprise-ready, mature and feature complete in regards to its requirements.

High API usability
It is very important that the middleware features a high API usability. This allows to write cleaner and more maintainable code, furthermore it enables developers to learn and understand the technology faster.

In the following sections we examine all major .NET communication frameworks and especially focus on these four points of interest.

2.4.1 Windows Communication Foundation

Due to Microsoft's huge influence to the .NET world, it is no surprise that their own - specially designed for .NET - WCF framework is the widely accepted standard for network communication. The Windows Communication Foundation has been introduced with the .NET framework 3.0 in 2006 and has become the de-facto standard to develop service-oriented applications [23]. It features a consistent interface to underlying communication technologies like DCOM, MSMQ, WSE and WebServices of all kinds (SOAP- as well as REST-based). HTTP-based web services allow C# applications to communicate with other frameworks, written in different languages and running on completely different systems (i.e. non-Windows, non-x86, etc.).

The main concept of WCF is the *endpoint*, which combines three sub-elements (the so-called "WCF ABCs") into a service:

- **A**ddress: a URI which defines how the endpoint can be accessed
- **B**inding: describes how the communication takes place, e.g. communication protocol, security mechanisms, timeouts, etc.
- **C**ontract: defines the interface of the service, i.e. which operations are exposed.

While you can easily write your own bindings, the default WCF implementation comes with a variety of bindings to support the most common scenarios, among them is SOAP over HTTP, SOAP over TCP, REST over HTTP (since WCF 4.0) as well as the MSMQ protocol. Furthermore so-called behaviors can be put on almost all concepts of the WCF framework and are able to extend the existing functionality. Here too, the WCF framework already comes with many default behaviors: metadata publishing, message transformation, message validation, etc.

In comparison to space based frameworks, WCF focuses on server-client systems in a service-oriented architecture style. Although WCF supports peer-to-peer systems since version 3.5, the possibilities are very minimalistic and are merely more than a publish/subscribe architecture [24]. Space based frameworks are far less server-client-specific and resemble more of a teamwork system in which the peers are cooperating within a shared space - although server-client applications can be implemented with space based frameworks as well. This restriction on server-client systems prevents us from using WCF as a Peer Model implementation basis. WCF's usability has been criticised by various sources [25], [26] & [27], still, it is certainly the standard for distributed components in .NET and we will compare it to the PeerSpace in our API usability evaluation.

2.4.2 NServiceBus

Initially developed in form of an open source project by Udi Dahan, a well-known .NET software developer and architect, NServiceBus features an enterprise service bus implementation, foremost concentrating on the publish/subscribe pattern [28]. NServiceBus runs on top of various communication layers (among others: MSMQ and RabbitMQ) and also features scalability, reliable integration as well as a workflow engine and auditing options.

NServiceBus follows a so-called "pit of success" principle [29], where the framework makes it very easy to implement certain scenarios, which are deemed as enterprise-proven, while trying to prevent developers from applying different, probably less adequate, patterns. E.g., publishing the same type of event from two different logical publishers is very hard to implement because it is not considered a best-practice in the enterprise service bus world [30]. Such restrictions may have negative consequences when creating off-mainstream distributed systems. If the supported communication patterns are sufficient, NServiceBus is a viable, enterprise-ready alternative to WCF. Regarding usability, NServiceBus chose an interface-based configuration approach. By implementing various interfaces, the framework is able to recognize the developer-intended behavior.

2.4.3 MassTransit

MassTransit is another distributed application framework for .NET. Similar to NServiceBus it provides a service bus implementation concentrating on the publish/subscribe pattern [31]. It features RabbitMQ as well as MSMQ support on the transport layer. Although MassTransit prominently features so-called "sagas", a state machine for long-term transactions, which can be configured with a handy fluent API, it has much less features than NServiceBus. Initially the two authors' goals have been to learn about distributed messaging systems, and it is still more of a private project than an enterprise-ready solution.

2.4.4 MSMQ

Microsoft Message Queueing is a low-level, message-oriented middleware developed by Microsoft and initially released two decades ago with Windows NT 4. MSMQ is a message queue implementation which allows asynchronous communication between two parties in a failsafe manner [32], i.e. communication is also possible if not both parties are online at the same time. Due to its age, the system is excessively stable, proven in the industry and has been incorporated into multiple Windows-based middleware solutions as the underlying communication technology (amongst them: WCF, NServiceBus and MassTransit). It supports various security models, priority messages and dead letter queues for timed-out messages. The Microsoft Distributed Transaction Controller also supports MSMQ and allows to have transactions spanning multiple queues at once. Nowadays the technology is, however, too low-level to be used directly from an application and instead serves as the basis for more high-level middleware.

2.4.5 Xcoordination Application Space

The Xcoordination Application Space is a coordination middleware for software components. It builds upon Microsoft's Concurrency and Coordination Runtime to connect worker threads. In contrast to using the CCR, the Application Space extended the concept from locally-only to remotely connected workers in a transparent way. As we use the Application Space as a foundation for the PeerSpace.NET framework, we go into more detail in section 3.2.2.

	Approach	Primary pattern	Communication
WCF	Service-oriented Architecture	Request/Response	Various
NServiceBus	Enterprise Service Bus	Publish/Subscribe	Queues
MassTransit	Enterprise Service Bus	Publish/Subscribe	Queues
MSMQ	Queuing	Request/Response	Queues
AppSpace	Peer-to-peer	Any pattern	Space based
PeerSpace.NET	Peer-to-peer	Any pattern	Space based

Table 2.2: Communication Frameworks in .NET - Architectural comparison

	Underlying paradigm	sync/async	Main configuration approach
WCF	RPC	sync still dominant	XML configuration
NServiceBus	Message passing	async only	Interfaces & FluentAPI
MassTransit	Message passing	async only	FluentAPI
MSMQ	Message passing	async only	UI Tools
AppSpace	Message passing	async only	Config string & Attributes
PeerSpace.NET	Space	async only	(to be decided)

Table 2.3: Communication Frameworks in .NET - Technology & Configuration approach

2.4.6 Technology Comparison

In this section we compare the various options with our planned Peer Model implementation (PeerSpace.NET) to get insights about the strengths of PeerSpace.NET and find its unique characteristics. In contrast to other frameworks we want to take advantage of space based middleware and to allow using any communication pattern. Peers should be highly scalable and therefore not have any shared state (which is also the recommended way to implement WCF services nowadays). Tables 2.2 and 2.3 summarize the most important differences of the various frameworks.

While WCF seems to be Microsoft's attempt to bring all messaging technologies under one roof, it is still (or perhaps because of that) very heavily RPC-oriented. RPC supports synchronous messaging only and mixes application-level with messaging-level responsibilities [33]. Furthermore, by disguising a remote call as a local call, developers easily fall for the eight fallacies of distributed computing (particularly for latency is zero or transport cost is zero). NServiceBus overcomes those issues by providing a publish/subscribe-based async-only framework, however, it has a steep learning curve and tries to restrict developers to certain patterns. MassTransit is the smaller brother of NServiceBus, however, not enterprise-ready yet. MSMQ is operating on a very low abstraction level, furthermore it does not support any other pattern than request/response. The PeerSpace.NET framework will be somewhere in between: featuring a high level of abstraction (like NServiceBus) but supporting many different patterns. In contrast to WCF or all queuing-based frameworks, communication will be based on top of space technology only, no WS_-technology [34] integration is planned. The framework which in the end is most comparable with our new PeerSpace.NET is the Application Space with which we will share most of the properties. As we will see at a later point however, the PeerSpace.NET framework will be based on top of the Application Space and extend it with additional capabilities.

	WCF	NSB	MTT	MSMQ	AppSpace	PeerSpace
Complex patterns	−	∼	∼	−	∼	+
Composition	+	+	+	∼	∼	+
Maturity	+	+	−	+	∼	−
High usability	−	∼	∼	−	+	+

Table 2.4: Framework support for the thesis' focus points

Table 2.4 returns to the four focus point which we defined at the beginning of this section. WCF and MSMQ are too specifically designed for a single purpose in order to support complex coordination patterns, furthermore the usability is - owed to the age of the frameworks - not state-of-the-art. NServiceBus, as presented earlier, focuses on some best-practice patterns, MassTransit is a publish/subscribe technology only and has problems to support very simple patterns like request/response. In terms of composition and code reuse the publish/subscribe systems NServiceBus and MassTransit shine, MSMQ has only 1-to-1 connections and is therefore limited in its composition possibilities. The AppSpace focuses on simple patterns like request/response and does not inherently support composition. As development has only just begun, the PeerSpace's weak point is maturity and feature completeness, however, follow-up work has already been initiated and the addition of transaction support is coming up. In contrast to the other focus points, high API usability could not be undermined with scientific references. Unfortunately there is no suitable existing research work, which is why the marks are opinions of the author. However, later on in the thesis we will perform an objective usability evaluation of our new API against an existing .NET middleware system to see if the proposed Peer Model API is indeed superior.

2.5 Framework Usability

Most of us know what usability means, when applied to *user* interfaces. The term is especially popular for graphical user interfaces (GUIs) and in nowadays world wide web and mobile phone app areas. However, the term is much more general. A very handy definition comes from Jakob Nielsen, a famous human-computer interaction consultant, who defined five important goals of usability [35]:

- Learnability - how easy is it for new users to accomplish basic tasks?
- Efficiency - once users have learned the interface, how quickly can they perform tasks?
- Memorability - when users return to the interface after a period of not using it, how easily can they reestablish proficiency?
- Errors - how many errors do users make? How severe are those errors and how easily can they recover from errors?
- Satisfaction - how pleasant is it to use the interface?

These five goals are not limited to graphical user interfaces but can be applied to all interfaces - particularly also to developers as users of framework-interfaces (application programming interfaces) [36]. Unfortunately there has not been a lot of research in the area of framework usability and most APIs do not even follow a published design guideline. In this section we are going to look at well-known qualitative evaluation methods as well as on an upcoming automated quantitative approach.

2.5.1 Qualitative Evaluation: Cognitive Dimensions Framework

The cognitive dimensions framework (CDF) has been originally described by Thomas R. G. Green and Marian Petre back in 1996 [37]. In its core CDF helps to evaluate and rate an existing framework in various so-called "dimensions". CDF is a textual approach and does not conduct any measurements which would result in concrete numbers, instead it is more of a general guideline to look upon specific important points in framework design (hence, a qualitative method). In its original form, thirteen different dimensions have been identified [38], a fourteenth - Juxtaposability has been added by R. G. Green soon after. Over the years more dimensions have been proposed by other researchers. After evaluating a framework in regards to those dimensions, we can immediately see weak spots in the design, which we want to remove (or at least reduce their impact) by applying design maneuvers.

Design maneuvers are changes made by the designer to improve a framework in a particular dimension. Dimensions have been created in such a way that they are pairwise independent, i.e. you can alter one dimension without modifying a second one - however, typically one would modify a third one. Therefore design maneuvers result in trade-offs between the individual dimensions. These trade-offs are inherent to usability, reflecting the assumption that there is no perfect user interface.

In 2004 Steven Clarke from Microsoft refined the list of cognitive dimensions. He adapted the list to meet the requirements of modern API evaluation. The following enumeration presents the new list of twelve dimensions [7], describes each dimension shortly and gives an example to clarify its meaning:

Abstraction level - renamed from "Abstraction gradient"
: What are the minimum and maximum levels of abstraction? Both values should be roughly the same. Can details at least be encapsulated?
Example: a communication API should not offer direct socket access and high-level coordination concepts at the same time, as both concepts are on completely different abstraction levels.

Learning style - new
: How steep is the learning curve of the API? What are the available learning styles?
Example: to work with log4net it is sufficient to read a tutorial, it is not required to take courses or to read a whole book about the framework.

Working framework - new
: How big is the size of the conceptual chunk in order to work effectively?

Example: log4net requires only a single configuration call to initially setup the framework and start working with it, i.e. the working framework is very small.

Work-step unit - new

How much of a programming task must/can be completed in a single step? I.e. how big is a single development step before you have once again an executable component with additional functionality?

Example: adding a service method in WCF requires a change at the interface and the addition of the method in the implementation class before the code is executable again, a reasonable work-step unit size.

Progressive evaluation - no change

How easy is it to evaluate and obtain feedback on an incomplete solution?

Example: many web service frameworks allow to test incomplete solutions and retrieve debug information - even supported by visual tools like soapUI.

Premature commitment - no change

Are there strong constraints on the order with which tasks must be accomplished? Do you have to commit to decisions early on without being able to change it later on?

Example: when using log4net it is required to select a log level system from the very beginning, changing it later in the project is very cumbersome.

Penetrability - new

How does the API facilitate exploration, analysis and understanding of its components?

Example: APIs with fluent interfaces like MassTransit support API exploration due to code completion very well.

API elaboration - new (has been a part of "Abstraction gradient")

To what extent must the API be adapted by the developer?

Example: socket communication users are highly encouraged to wrap the exposed socket API and define a communication protocol instead of sending byte arrays directly over the wire. I.e., the API should/must be adapted before actually using it.

API viscosity - renamed from "Viscosity"

How much effort is required to change code expressed with the designed framework?

Example: WCF allows to define so-called "behaviors" for existing components, enabling developers to change the behavior without the need to modify the existing code at all.

Consistency - no change

After the initial part of the framework has been learned, how much of the rest can be successfully inferred?

Example: as soon as one knows how to log messages with log4net, it is very easy to infer how to log exceptions, a good example of API consistency.

Role expressiveness - no change

How obvious is the role of each component in the solution? Reading a code snippet should

allow the developer to immediately understand it.
Example: for many WCF concepts the documentation must be consulted before fully understanding why they have to be applied to a certain service, unfortunately, it is often not obvious, even for an experienced developer.

Domain correspondence - renamed from "Closeness of mapping"
How close does the API map the problem domain?
Example: the socket API cannot map any complex communication pattern directly due to its low level of abstraction. E.g., the publish/subscribe pattern would have to be implemented from scratch.

This new and updated list provides better results and furthermore simplifies the evaluation of modern APIs in regards to API usability. Note the relation of many points to the five usability goals defined by Nielssen.

2.5.2 Qualitative Evaluation: User Tests and Expert Panels

User tests and expert panels are two qualitative methods to gain insight about framework usability. While user tests are performed on a draft or prototype of the final product, expert panels may already start at an earlier stage during the initial design.

Expert Panels

Expert panels bring together specialists of the topic under discussion. Together they focus on conclusions and recommendations through consensus [39]. In the software usability area we have two kinds of specialists: software architecture experts, who bring in a lot of knowledge and experience in the API design area, and problem domain experts, who know everything about the framework's target domain. After a presentation of the problem domain's key concepts the group can find ideas and solutions in a cooperative way. Various group creativity techniques (like brainstorming or 6-3-5 brainwriting [40]) can help to come up with an initial API design which can then be refined.

To evaluate the API by experts some methods have been proposed by [41] which include cognitive walkthroughs, where the experts simulate the user's problem-solving process and check at each step if potential usability problems are adherent. The addition of domain experts is important to focus on domain-driven design [42] principles like bounded contexts or an ubiquitous language and of course to help the software architects to understand the problem which is solved by the framework.

User Testing

User tests are applied in a later phase of development, after the first initial draft or a prototype is implemented (at least the API must be specified, the underlying implementation can be faked like it is often the case in unit-testing). User testing involves measuring the performance of typical users doing typical tasks in controlled, laboratory-like conditions [41].

In case of framework usability we have developers as our users and we may expect a good understanding of the programming language, as well as some knowledge of the domain and its underlying difficulties, i.e. distributed systems and a basic knowledge of the distributed systems design fallacies. Observing the developers while performing usual tasks with the framework leads to the desired results. Studies [43] [44] have shown that more than five to seven people do not add additional value to the user test, as the additional costs merely bring up any new information. In general, the high time consumption and high costs are the major disadvantage of this method. This led to the development of alternative ideas like measuring the usability using heuristic guidelines [45] and more recently to quantitative usability evaluation methods.

2.5.3 Quantitative Evaluation: An Automated API Usability Evaluation Framework

Qualitative evaluation methods face two major drawbacks: they are very cost- and time expensive. Skilled personnel is required to accomplish good, but still opinion-based [46], results. Furthermore, qualitative methods are often only applicable if a final draft of the API is already available. Hence, an automated usability measurement method would be favorable, which gives us objective and replicable results. Various software tools are available, which output a broad range of so-called code-metrics (e.g. cyclomatic complexity [47] or the object-oriented software metrics by Chidamber & Kemerer [48]). Unfortunately, those metrics concentrate on the inner structure of the code and not on its public API. For users of a framework it is irrelevant how complex the implementation is.

Studies on the usability of public APIs are rare. Jeffrey Stylos has examined very specific aspects of APIs: in [49] he stated that an API should require the users to work with factories only when absolutely necessary due to increased complexity. In [50] he compared different method placements and concludes that it is important that there is some kind of "starting class" within a framework from which other required types can be inferred because they are referenced as method parameters or similarly connected to the "starting class". He also compared default constructors with additional required property-setters and required constructors with each other in [51] and found that the first approach has better usability. Thomas Scheller takes a more methodical approach and identified measurable concepts of API usability in [52] and [53].

The first completely automated API usability evaluation framework is presented in [8]. It allows, even for developers which are inexperienced in this area, to obtain measurable and objective results for a given API. The paper proposes that the complexity of a framework solution can be split into three measurable sub-characteristics:

Interface Complexity

Describes the complexity of the used elements of an API (classes, methods, fields, etc.). The fewer elements a developer must use, the better. Important is how an element is used: from calling a method to implementing a whole class.

Implementation Complexity

Measures the complexity of a framework's underlying implementation. This part can be

approached using existing code metrics, which assess the implementation complexity of a solution.

Setup Complexity
Includes any action that a developer must perform before an API can be used. Installations, configurations, etc.

The newly developed framework [8] concentrates on interface complexity, where today no automated way of evaluation is available. To perform an evaluation, the framework is given a step-by-step code example. Each required step (e.g. implementation of an interface, calling a method, adding an attribute, etc.) is given certain complexity values. The framework uses reflection to obtain additional information, i.e. how many parameters a method call requires or how many different methods an object provides in the first place and thereby assigns a score value to each step according to documented rules. In addition to evaluating these low-level concepts, the framework also supports high-level concepts which alter the total score. Currently supported concepts include the factory pattern, fluent interfaces, XML configuration files and configuration strings. Furthermore the framework allows us to obtain different interface complexity measurements depending on the number of previous usages by the user. Thereby we can see how good the learning experience of a given API design will be. In the end the framework provides us with an objective interface complexity measurement which can be compared with different API designs.

CHAPTER 3

Background Technologies

This chapter draws a rough plan of the implementation approach and presents all the technologies which have been used to implement the PeerSpace framework.

3.1 Core Implementation & Usability-Focused API

The idea of the PeerSpace framework is 1) to be the very first implementation of the Peer Model based on .NET and 2) to provide an API which is easy to use. Optimally it features a measurable good API usability according to qualitative and quantitative evaluation methods. To concentrate on each of the two requirements individually we decided to split the PeerSpace in two parts: a core implementation and a usability-focused API (see figure 3.1).

The PeerSpace core is going to translate the requirements of the Peer Model (see the requirement analysis in section 4.1) into framework components and will focus on feature-completeness and extensibility. Developers on the other hand are not supposed to work directly with the PeerSpace core. To create Peer Models in code we provide an additional API on top of the PeerSpace core, which is based upon state-of-the-art research in the framework usability area.

3.2 Technologies Used

This section will describe all the technologies used by the PeerSpace project and the reasons why they have been chosen.

3.2.1 .NET Framework

The .NET Framework is a software framework developed by Microsoft. It serves as the basis for many applications running on Microsoft Windows [54], however, meanwhile there are also .NET implementations for other operating systems available, e.g. the open source, cross-platform implementation Mono [55]. Initially released in 2002 as a Microsoft alternative for

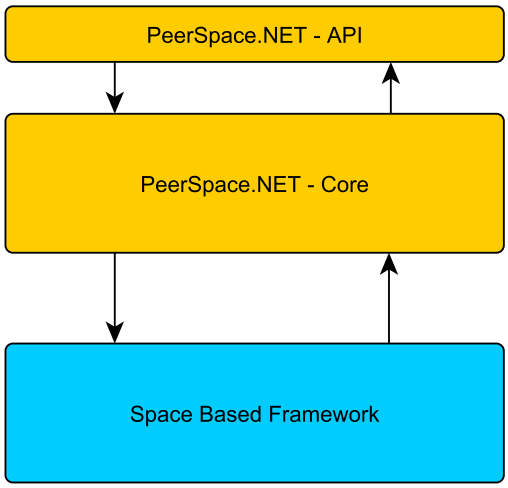

Figure 3.1: The PeerSpace.NET architecture

Sun's successful Java environment (now owned by Oracle) the .NET Framework has become the de facto standard for client application development on the Microsoft Windows platform. Although rival platforms are upcoming, e.g. HTML app development, Microsoft still believes in .NET and version 4.5.1 has been released recently (October 2013) alongside Windows 8.1.

The .NET Framework is formed by two components. The "Common Language Runtime (CLR)" and the "Base Class Library (BCL)". All .NET applications run within the CLR software environment, a virtual machine very much like the Java Virtual Machine, which provides services to the application like memory management, exception handling, etc. The BCL provides base classes like collections, threads or network access to the developer [56]. There is more than one programming language targeting the .NET Framework, and assemblies are capable of being used as part of other .NET languages without the need of special wrappers. Major programming languages which target the .NET language are C#, Visual Basic.NET (VB.NET) and F#.

For the PeerSpace project a bunch of different factors led to the decision to make use of the .NET Framework in combination with its most popular programming language: C#. The PeerSpace shall serve as a general-purpose coordination library and must therefore be addressable by at least one widespread used programming language for application development, which narrowed our selection down to C, C++, C#, Objective-C, Java and Visual Basic according to the well-known TIOBE programming index [57] (excluding all script languages like PHP, Python and JavaScript). Space based implementations of XVSM exist for C and Java, the Application Space is an alternative space based framework for .NET. In contrast to XVSM, the Application Space is a programming model that is truly usability-oriented. Furthermore, C# is the most modern language of the three languages mentioned before, and the resulting assemblies can be used from any other CLR language like VB.NET as well, which is why we decided to go for a .NET-based implementation using C#.

3.2.2 CCR and the Xcoordination Application Space

As there is currently no full-featured .NET-based XVSM implementation available, the PeerSpace implementation had to look for alternatives in order to prevent a re-implementation of XVSM in .NET, which would have gone beyond the scope of this thesis. Luckily, Thomas Scheller, of the very same university institute, has implemented an alternative space based framework which is based on .NET: the Xcoordination Application Space [58]. It does not follow the very same principles as XVSM and does not come with typical tuple space characteristics, but it features the required possibilities to deliver a fully-featured Peer Model implementation. In case a full-fledged XVSM becomes available for .NET, the module-based architecture of the PeerSpace project allows to exchange the underlying space framework easily.

The Application Space is based upon Microsoft's Concurrency and Coordination Runtime (CCR) [59] [60], an asynchronous programming library which originally was developed for the Microsoft Robotics Developer Studio (MRDS) but had a breakthrough into mainstream usage. Basically the CCR implements the dispatcher pattern, which executes so-called "work items" in a thread pool with a fixed number of threads, all of which can of course execute simultaneously. The dispatcher provides a generic "port", which is itself a simple queue where items can be posted, which are then processed asynchronously by the dispatcher in the background. Other work items may consume the results for further processing. One can already see some similarities between this concept and the Peer Model's services and wirings.

While the CCR is a framework which runs on a single local machine only, the major features provided by the Application Space framework are the possibility to connect work items via remote ports and the add-on of an additional abstraction layer which much more resembles the original idea of space based architecture. The PeerSpace implementation will be described in a later chapter, however, as one can already imagine we decided to build wirings between different peers upon those connections.

Also very nice for our purposes is the well-designed software architecture of the Application Space. Its design enables users to replace or extend almost any part of the Application Space with own interfaces, amongst them:

- Direct configuration of the underlying CCR dispatcher
- Different transport services
  (with many useful out-of-the-box transport services like TCP or WCF)
- Different security services
- Different message interchange formats
  (= different serialization mechanisms)
- Different logging mechanisms
- Extensibility of all components

All these features will contribute to the PeerSpace implementation later on. For example, due to the fact that the Application Space enables to use security services, the PeerSpace framework can provide security to its users with little effort.

3.2.3 Utility Libraries

In this subsection we discuss required minor libraries which are used as utilities within the PeerSpace.NET implementation. This completes the enumeration of used technologies.

NuGet

NuGet is a package manager for the Microsoft development platform (particularly used with the .NET framework) [61]. It consists of a client to include packages into a project, automatically resolving any transitive dependencies, and a global package repository from where dependency packages are obtained.

The NuGet client tools also allow to create your own packages and upload them to a repository, most notably the "NuGet gallery", a central package repository managed by the NuGet organization.

NuGet adds handy dependency management to the PeerSpace. During the implementation we made use of NuGet in both possible ways: First, all of our project's dependencies have been obtained via NuGet, secondly, we created a PeerSpace NuGet package and uploaded it to the NuGet gallery to enable developers to easily include the framework into their own projects.

JSON

The JavaScript Object Notation is a lightweight, text-based, language-independent data interchange format. JSON defines a small set of formatting rules for the portable representation of structured data [62] [63].

Within the implementation of the PeerSpace we use JSON for two scenarios. First, we use JSON as the default data interchange format between remote components. Secondly, we use JSON for all kind of auditing tasks. Exchanged messages (which are essentially lists of C# objects) can be looked on easily when presented to the auditor as JSON.

The usage of a JSON serializer (e.g. in comparison to a binary serializer) enables much easier integration of incompatible external data sources into the system if necessary. If performance is a major issue, the Application Space can be reconfigured to use a binary serializer instead.

log4net

log4net is the most common logging framework for the Microsoft .NET Runtime. Based on the even more successful log4j project for Java, the developers cloned the project and tuned it for better use with .NET languages like C# [64]. In particular, it enables fine-grained logging. This enables users of the PeerSpace framework to obtain different levels of log message details, depending on their requirements.

CHAPTER 4

Requirements Analysis

In the previous chapter we gave a rough overview about the approach to split the PeerSpace in two components: the framework core is responsible for providing a feature complete Peer Model implementation, the developer-usability-focused API is responsible for providing an easy-to-use and future-proof interface in favor of the very simplistic core interface. This chapter collects all functional and non-functional requirements for the PeerSpace core and therewith prepares the actual implementation.

4.1 Core Requirements & Architecture

The core framework should provide a full-fledged implementation of the Peer Model. On the basis of this premise we derived five design goals for the core:

- Feature-complete Peer Model implementation
- Using an underlying (and replaceable) .NET-based middleware technology
- Modern .NET framework design (especially extensibility)
- Possibility to plug a usability-focused API on top
- Usage of future-proof .NET technologies

Only the first of these goals targets the area of functional requirements, the other four are non-functional goals. By analyzing the Peer Model specification and following our five design goals we will derive the functional and non-functional requirements in the next sections.

4.2 Functional Requirements

The Peer Model consists of four important main entities: peers, containers, wirings and entries (see section 2). In this section we will first derive the functional requirements of the four main entities, afterwards we discuss all mechanisms which require a more thorough analysis in detail (e.g. the concrete communication layer requirements which are not specified in the Peer Model at all).

4.2.1 Peers

Peers are autonomous software components and must be able to run as standalone components on any .NET-capable system.

Definition of a Peer

Peers have four important properties, defining its main requirements:

Self-reliance
: Peers are standalone components, they do not require any other service or application to run in order to fulfill their work.

Structure
: Peers are configured in terms of their wirings and a few additional properties (e.g. logical name or physical address).

Reusability
: Peers may be scaled-out multiple times. Therefore the structure of a peer is comparable to a templating mechanism: you can create any desired number of peer instances from a single peer definition (i.e. peer configuration). This also means that peer instances do not share any state between each other.

Addressability
: Each peer instance must be uniquely addressable such that communication with other peers via a local or global network is possible.

Although we use the term “peer” for the unity of the addressable peer, its containers and a number of wirings, the peer itself is only a passive resource. The only active part of the peer are its wirings.

Types of Peers

In theory, the Peer Model distinguishes various types of peers: application peers, space peers, coordination peers, runtime peers and a special “universe peer” which bootstraps the distributed system.

The implementation will only support application peers, all other types are either special manifestations of application peers or theoretical models. That is, a space peer is an application

peer with only a single container and no inherent logic. Coordination peers are similar to application peers in functionality, however, they do not contain any application logic. Their purpose is to fulfill coordination logic only. One can imagine a future coordination pattern library, which already contains a range of coordination peers for all kind of purposes. Runtime peers encapsulate all peers running on a local site, however, as this does not help the implementation, we decided to not implement the concept.

The universe peer is the extension of the runtime peers on a global level. In theory, all wirings between peers run within a so-called universe peer, however, this is a virtual concept: software may only run on a single computer. We will address this sophisticated issue in section 4.2.5 after the other components have been discussed. Furthermore, universe peers initialize the system, e.g. by putting initial data into peer containers. Such an initializer is not required in a real environment. Input originates either directly from a user, another system (e.g. a sensor, etc.) or from data sources like files or databases. We need to provide a way for programmers to fill a peer's container with such input data instead.

Another important characteristic is that peers can be nested: peers may contain sub-peers. This enables software architects and developers to reuse complicated coordination patterns within a peer by combining existing, functionally reliant and proven peers. In this thesis we have introduced a small simplification regarding sub-peer addressability: sub-peers are only visible inside their immediate parent peer, outside peers cannot communicate with them directly.

4.2.2 Containers

Basically, containers are mere data holders for entry instances. The concept originates from the XVSM framework [5]. Containers provide an interface to add, query and remove entries in various ways. Peers receive and emit data via containers. Per default, each application peer instance consists of two containers: an in-container (PIC) and an out-container (POC). Space peers are an exception, they combine these two containers into a single one.

The only active component of the system, the wirings, are responsible for moving entries between containers. Wirings activate ("fire") as soon as their corresponding guard can be fulfilled. At the end of the wiring execution, the resulting entries are once more moved to one or more containers (if a completely different peer is addressed by a wiring, the entry is moved into that peer's PIC container).

Containers may be implemented in various ways, manifesting in different container properties. A container implementation's capabilities affect the capabilities of the whole PeerSpace behavior. Containers may implement persistence and support transactions, which allows the whole PeerSpace to be persistent and transactional and therefore reliable even if the peer or the peer's host system crashes. Many more possible features of containers include: authentication, authorization and other security features [65], replication [66], auditing, etc. To support extensibility (one of the non-functional goals), we expect containers to be "decorable" [67], i.e. we can add additional features to a simple container implementation by wrapping them.

It is easy to see that containers play an important role in the PeerSpace and it is essential that the specific container implementation is easily exchangeable via configuration options. The initial PeerSpace core implementation provides an in-memory container with full support for the

later presented Container Query Language (see section 4.2.8). Thread-safety and basic auditing using the log4net logging framework are added by two container decorators.

4.2.3 Wirings

Wirings are the only active component of the Peer Model and therefore play a very important part. Although the theoretical Peer Model does not distinguish between them, there are actually two types of wirings:

Inner-peer wirings
: Wirings which conceptually run within a peer (typically they move entries from the PIC container to the POC container, i.e. they use a peer's PIC container as input source). Also moving entries from an enclosing parent peer to its sub-peers and back is handled by inner-peer wirings.

Inter-peer wirings
: Wirings which connect multiple peers with each other (they move entries from one peer to another, i.e. they use a peer's POC container as input source and move the entries to another peer).

In theory, all wirings run within a peer, at least within the theoretical concept of the universe peer. Therefore no inter-peer wirings were necessary in the original Peer Model definition, however, this is not practical for a real implementation. Inter-peer wirings are fundamentally different from inner-peer wirings as they need to cross process borders, probably even system and network borders. A big design question for wirings is how to integrate inter-peer wirings, which is why we discuss it in an extra section later on (see section 4.2.5).

In contrast to peers and containers, the design of the wiring interfaces is pretty straightforward. Wirings have a very precisely defined behavior in the Peer Model and use only defined interfaces (e.g. the container interface) without providing any own interfaces to the rest of the system. Each wiring consist of three parts:

- Guard
  (consisting of one or more guard links)
- Service(s)
- Action
  (consisting of zero or more action links)

Each guard link is essentially a blocking query against a container. Multiple wirings may of course query a container concurrently. Whenever all guard links of a wiring are fulfilled, the wiring "fires", and moves the queried entries from the container into an entry collection. The entry collection is a simple in-memory container which serves as a data container during the execution of a wiring.

Note: actual implementations will not use blocking queries, instead it is faster to check if the container can fulfill the guard whenever a new entry is put into the wiring's source container. The details are presented in chapter 5.

Wirings may incorporate one or more services. Services retrieve parts of the entry collection as input and may emit additional entries into the entry collection as output. Services encapsulate all the application logic within the Peer Model and therefore ensure a very good separation of coordination logic and application/domain logic. In the PeerSpace implementation services may be arbitrary .NET methods. The parameters of a registered service method are automatically filled by the system with matching entries from the entry collection.

After all services have been executed, the action links are executed one after the other. They are very much like the guard links, however, they use the entry collection as input source. Each action link has a destination property, which defines where to put an entry (e.g. into the own PIC/POC, or into the PIC of a sub-peer). The framework tries to fulfill the action link, and in contrast to guard links, if an action link cannot be fulfilled it is simply ignored (and does not block). Entries which are not taken out of the entry collection by any action link are dropped and not moved into any container. In contrast to inner-peer wirings, which use the PIC container as input source, inter-peer wirings use the POC container instead. They have action links with destinations at possibly remote peers.

Please note, that in this thesis we assume that each wiring execution creates a new transaction. The transaction scope spans from removing the very first entry from a container until the last put operation of an action link has finished. Therefore, if a wiring fails for some reason and the container would support transactions, a peer could crash or be restarted at any point without losing entries.

4.2.4 Entries

An entry represents the information which is moved around in the Peer Model. In other frameworks for distributed systems, entries would most likely be called messages, however, the term is deliberately avoided within the Peer Model. First, it is a well-defined term in the area of space based computing and secondly, entries represent state within the container, unlike a message, which only exists during the transfer from one point to another.

Entries consist of two parts: the application data (app-data) and the coordination data (co-data). While the former represents the actual information, the latter is a label-value dictionary which holds meta information about the entry. Users of the Peer Model may put user-defined properties into the co-data dictionary. It is also a place where extensions of the PeerSpace may put data, e.g. for flow control or for implementing distributed transactions. The core implementation also puts some default co-data fields into an entry which are used for e.g. the auditing system. Three additional co-data properties (Destination, TimeToStart and TimeToLive) fulfill special purposes:

- Origin - the peer which created the entry in the first place
- LastSender - the peer which last sent the entry
- Created - timestamp for the creation of the entry

- LastSent - timestamp at which the entry has been last sent
- LastReceived - timestamp at which the entry has been last received
- Hops - number of hops the entry has travelled so far
  (whenever the entry has been moved to another peer, the number of hops is incremented)
- Destination - routing information (discussed below)
- TimeToStart - earliest point in time at which the entry is eligible for selection by a wiring
- TimeToLive - point in time at which the entry has expired and is not eligible for selection by a wiring anymore.

Normally the wiring (i.e. the wiring's action) decides whereto an entry is routed. This has one major drawback: action links are of static nature and do not change the destination during the peer's lifetime. It is favorable in some situations to let a service decide whereto an entry should be routed [1], [14]. E.g. imagine a peer which should distribute entries to a number of other peers using a round-robin algorithm. This is achieved by setting the destination property of an entry within the service. If the destination property is set, the entry is moved before any action links are executed (i.e. it triggers a special system wiring, which dispatches the entry to the given destination). This is considered a major feature, and the core framework will even provide an API for setting the destination property of an entry.

Another possible extension is the usage of a "flow ID" property, which identifies the entry as corresponding to a flow. Wirings only fire using entries which are flow compatible as described in section 2.3.3.

4.2.5 Inter-peer Communication

As mentioned above, the Peer Model does not distinguish between inner-peer and inter-peer wirings at the communication layer level. This is a consequence of the fact that in theory all communication is within a conceptual enclosing peer, i.e. a runtime peer or at least the universe peer. This is technically not feasible: peers may run on different hardware machines and do not necessarily have an enclosing context.

Whenever two peers need to communicate with each other (or better: a wiring moves an entry from a source peer to a different destination peer) we need to transfer entries over one of the following two borders:

- Local peer barrier - messages (entries) are exchanged between two peer processes on the same machine (process boundary)
- Remote peer barrier - messages (entries) are exchanged between two peers on different machines (machine boundary)

The underlying communication technology may be able to hide the barrier, e.g. the XVSM framework is even capable of making all those barriers completely transparent. However, we

cannot assume that every technology used for the PeerSpace is able to do so without major performance losses. Therefore, whenever the destination of an entry is another peer, the PeerSpace core makes use of a so-called communication layer to move the entry. The communication layer abstracts away the underlying communication framework, more details are presented in a subsequent section (see section 4.2.6).

Inter-peer Communication: Use a PUSH or PULL Approach?

In this section we concentrate on another issue of inter-peer wirings: as an active component they must execute on a single machine, however, there are multiple options:

- Run the wiring on the source peer's machine (within the source peer's process) -> **push** the entries to the destination peer
- Run the wiring on one of the destination peer's machine (within the destination peer's process) -> **pull** the entries from the source peer

Both options have their advantages and disadvantages, which requires us to do some research before making a final decision. We will discuss technical as well as organizational consequences and then decide which option is better for the PeerSpace core implementation.

Technical consequences of PUSH vs. PULL approach: The basic differences between push-based and pull-based approaches are already common knowledge, and may be found in various papers (e.g. [68]). In a nutshell: a push-based approach reduces the latency by immediately transferring an entry to the receiver and keeps network traffic at a minimum (it does not force any additional communication). Pull-based approaches in contrast will always slow down the PeerSpace a bit by introducing small latencies between the point in time when the entry is ready for transfer and the point in time when the receiver actually pulls the data. In addition, it is a major issue to find an adequate time span between two pulls by the receiver. Pulling too often, rises network traffic with useless pull requests, pulling too rarely slows down the PeerSpace, as entries are not immediately transferred. Pull-based approaches have advantages in two scenarios:

- Asynchronous communication, i.e. the receiver is not always online (e.g. receiving mails from a mail server)
- Throttling communication, i.e. the receiver is currently not able to process all messages from the producer (e.g. master/worker design pattern)

However, even in those two scenarios we are easily able to model a pull-based mechanism with a push-based PeerSpace core implementation. The Peer Model in figure 4.1 presents the required wirings for such a system design. The wirings allow us to fire only if there is an additional "pull token" (PT) by the polling peer. This activates the pushing behavior only on special request, which is the definition of a pull-based system.

As handling communication errors is necessary for both, push- and pull-based mechanisms, in similar ways we do not discuss it in this section.

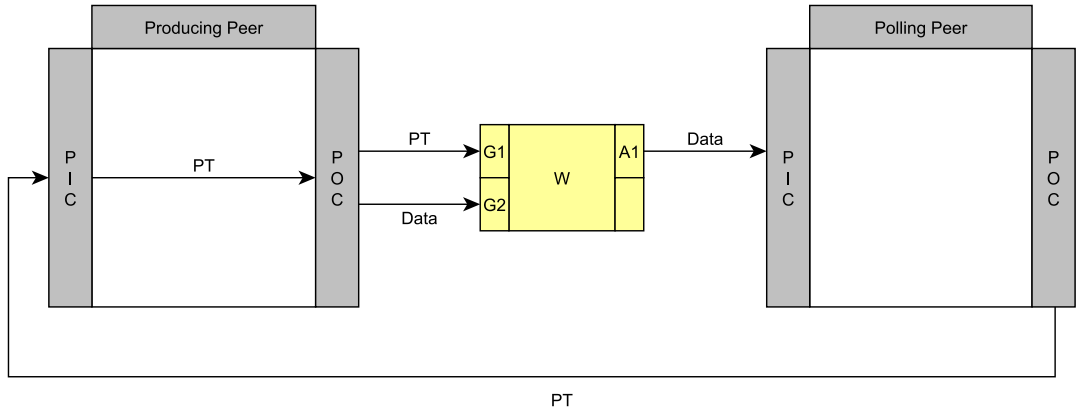

Figure 4.1: Modeling pull-based communication with a push-based mechanism

Organizational consequences of PUSH vs. PULL approach: As the inter-peer wiring runs within the context of a peer, the developer is forced to place the wiring in the configuration of the corresponding peer. This results in configuration differences between push- and pull-based implementations. If the PeerSpace core is implemented using a push-based mechanism, the developer needs to install the inter-peer wiring at the source peer, if the PeerSpace core is implemented using a pull-based mechanism, the developer needs to install the inter-peer wiring at the destination peer.

The pull-based wiring would need an additional mechanism to know if it should fire and try to pull data from the source. This concept has not been discussed in the theoretical Peer Model and would require additional functionality and increase the complexity of the system. Push-based implementations embed conceptually more natural into the Peer Model, as inter-peer wirings can use the exact same mechanism as inner-peer wirings, they just operate on a remote container.

Atomicity of wirings also favors the push-based mechanism: because all the logic of a single wiring (including the final transfer of an entry to a given destination) runs within a single process. Furthermore, although distributed transactions are not yet part of the core framework, we also looked on the effects of push- vs. pull-based mechanisms when implementing distributed transactions: no important differences have been found, both are required to implement a distributed commit protocol (e.g. two-phase commit) or make use of a distributed transaction controller. A follow-up thesis will add distributed transaction support to the PeerSpace framework.

Conclusion: We discussed advantages and disadvantages of inter-peer communication and showed that a push-based mechanism integrates more naturally into the PeerSpace framework. Furthermore we presented a way to imitate a pull-based mechanism with the push-based implementation if required. On that basis we decided to implement the PeerSpace core with push-based inter-peer communication.

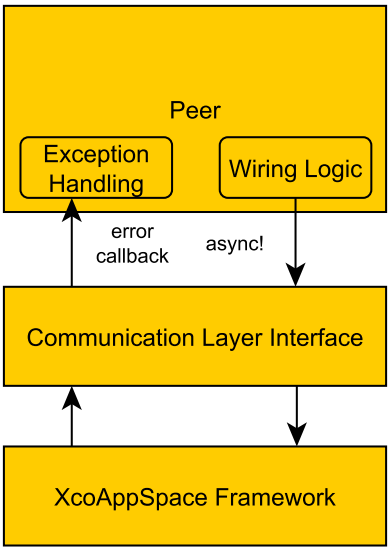

Figure 4.2: Inter-peer communication layering

4.2.6 Communication Layer

This section discusses the communication layer, an abstraction (see figure 4.2) used by the PeerSpace core to exchange entries between peers over process and machine boundaries. In contrast to previous sections, the requirements in this section are not derived from the theoretical Peer Model, as the communication layer is not mentioned and process and machine boundaries are expected to be transparent in the Peer Model (by putting all peers within a conceptual universe peer).

The PeerModel makes various demands regarding the underlying communication technology in order to provide a higher abstraction level compared to other modeling environments. More specifically, the Peer Model requires space based technology, which enables the PeerSpace framework to rely on all the guarantees made by the underlying space (e.g. security, reliable transmission of data, etc.).

When planning to implement the PeerSpace framework with .NET we wanted to use an existing space based framework, otherwise this task would exceed the scope of this thesis. Unfortunately there is currently no up-to-date .NET implementation of XVSM (eXtensible Virtual Shared Memory, the institutes's own space based framework). There is however, another space based framework developed by the Xcoordination group which sufficiently fulfills our requirements: the Xcoordination Application Space framework (see section 3.2). Very like the possibility to configure which container implementation is used, we are also able to exchange the actual communication layer at any point by changing the configuration. Among other things, this enables us to switch back to XVSM, if the .NET implementation becomes up-to-date again.

4.2.7 Handling Communication Errors

The theoretical Peer Model has no specification for handling communication errors occurring during inter-peer communication. Although a concept of error entries exists, the Peer Model does not go into much detail about error handling. When dealing with processes on different machines and network communication it is however of utter importance to have clear requirements for error handling. No system or network is reliable enough to guarantee error-free communication at all times.

The original Peer Model paper defines a rule that each entry whose time-to-live (TTL) expires, is automatically wrapped into an error entry and put into the peer's POC container. System wirings should let the error "roll back" until it is back at the client that originated the flow [1]. This concept is not sufficiently detailed to derive functional requirements. Therefore in this thesis a solution for error treatment needed to be designed. Among the open questions were:

- Which peer is the origin of a flow?
- What happens if the roll-back-wiring itself fails to forward the error entry?
- Is there an alternative routing system to bring the entry back to the origin or does it have to follow its original route?

At first it sounds like a good idea to shift the responsibility of error handling from the peer to an outside entity. However, in a distributed system each entity must be able to react to exceptional situations by its own, e.g. if the network is down and the peer cannot reach any other peers anymore, it is forced to react itself. Possible reactions are trying again after a short time span, choosing an alternative destination, logging the error and/or simply dropping the entry. The concept of putting an error entry into the POC container is apparently not enough.

In order to enable each peer to react to communication failures, each wiring may define communication handlers: callback methods which are called either on success or on error. The full context is given to the callbacks (including the thrown exceptions) and provides them with various control mechanisms to create an adequate reaction (e.g. initiating a retry).

4.2.8 Container Query Language

The Container Query Language is a small SQL-like query language which allows to select a collection of entries from a container. If the queries' demands are fulfilled, the query returns and the corresponding wiring may fire.

Such a query consists of one or more query clauses. Each clause targets entries of a different entry type. For each type we can furthermore select a quantity, specified using a relational operator (either "Exactly", "MoreThan", "MoreThanOrExcatly", "LessThan" or "LessThanOrExactly"). In the end we can specify if we want to read the selected entries from the container or take them out of the container. A wiring must at least have one "take" operation in order to prevent infinite firings.

In addition to the original requirements in the Peer Model paper, extra mechanisms have been introduced: a user-defined predicate has been added to the query clause. Each entry is

matched against the predicate and only selected if it fulfills the predicate. This can be used for example to select only entries with specific co-data. Another feature is an operation termed "none", which checks that no entry of the given type is available in the container.

In order to give a better impression of the possibilities of the Container Query Language, we present some example queries in informal language:

- Query with a single clause:

```
take exactly one entry of type WeatherData
```

- Query with two clauses and different relational operators:

```
take exactly one entry of type SensorDataDemandToken
take at least one (but as much as available entries)
  of type SensorData
```

- Query with clauses including predicates:

```
take exactly one entry of type TempSensorData
  where entry.ID = 543
take exactly one entry of type PressureSensorData
  where entry.ID = 543
```

Note, that entries can only be selected by a query if there TimeToStart has been reached and their TimeToLive has not yet been reached. Furthermore a query can only be fulfilled by entries with compatible flows as described in section 2.3.3.

4.2.9 Summary

In this section we summarized the functional requirements to fulfill the original Peer Model's specification and extended it where necessary in order to be able to implement a sound framework.

4.3 Non-functional Requirements

In this section we derive explicit non-functional requirements for the PeerSpace framework. Instead of focusing on performance only, most non-functional requirements are from the area of quality software engineering. This ensures that the implemented framework enables future development.

4.3.1 Extensibility

Extensibility is important for the core framework due to various reasons. We already identified some parts in the PeerSpace core which are very likely to be changed by future users, e.g. configuration of another container implementation class or addition of further entry properties (user- or system-defined). Furthermore, there are already planned future features like transaction support, which should not require a reimplementation of the framework.

4.3.2 Maintainability & Documentation

The Peer Model itself is still in research and development and future change requests to the PeerSpace are inevitable. It is therefore important that the PeerSpace implementation is able to react to changes and adapt quickly without breaking already-used interfaces. Documentation throughout the whole code base (also using C# XML documentation comments) as well as the thesis itself should provide enough information for future developers to maintain and enhance the PeerSpace framework. Usage of enterprise-ready technology as well as software design according to established guidelines underline the effort. During the implementation code inspection tools like ReSharper [69] are used to maintain high standards.

4.3.3 Fault Tolerance

Another important requirement for distributed systems is fault tolerance. Failures cannot be eradicated and their possible occurrence must be taken into concern. As we have already seen in the previous section about handling communication errors (see section 4.2.7), responding to errors in different ways is a core requirement of the PeerSpace implementation. It is also important, that any error which occurs in the application logic (i.e. the services) does not interfere with the coordination logic of the Peer Model.

4.3.4 Testability

Nowadays it is also important to ensure testability of code. This allows to write unit tests for individual components as well as integration tests for the whole system in order to guarantee the correct functionality of all components. Another option would have been to drive the whole implementation using a test-first approach (called “test-driven development” [70]), however, this is not recommended for research projects due to their frequent changes during initial development.

CHAPTER 5

Framework Design & Implementation

After collecting all requirements in the previous chapter we will now present the final architecture of the PeerSpace core framework and give implementation notes where necessary. The chapter concludes with a small sample on how to use the core implementation, which also shows the necessity of the additional user-friendly API.

5.1 Architecture

The PeerSpace architecture resembles the functional and non-functional requirements, which have been established in the requirements analysis. The initial section will describe the architecture, supported by UML class & package diagrams. Figure 5.1 shows the four main interfaces of the PeerSpace implementation: `IPeer`, `IContainer`, `IWiring` and `IEntry`. Additionally important interfaces of the core framework are shown in figure 5.2. In general, this chapter will only present important implementation details and does not go into the details of every method and property of each class. The C# XML documentation of the given interfaces and their respective implementation classes provide sufficient documentation, if more detailed information is required.

The `IPeerConfiguration` interface holds all required data to instantiate a new peer instance and run it: an address to name and locate the new peer, all sub-peers, of course all of its wirings, its initial local state (if any), a startup and a shutdown action to be executed immediately after startup and immediately before shutdown as well as two callbacks which are called in case of succeeded communication or in case of an exception during communication. While normally a peer's wirings are checked each time a new entry is put into the container, a wiring check interval allows for additional checks at regular time intervals. This is required only, if the TimeToStart property is used (see section 6.4.5). The `IPeerConfiguration` interface allows developers to easily scale out a peer definition by instantiating the peer multiple times with different `Address` values.

Three factory interfaces serve the extensibility and maintainability goal. All PeerSpace core interfaces are completely technology independent. When instantiating a peer from the given

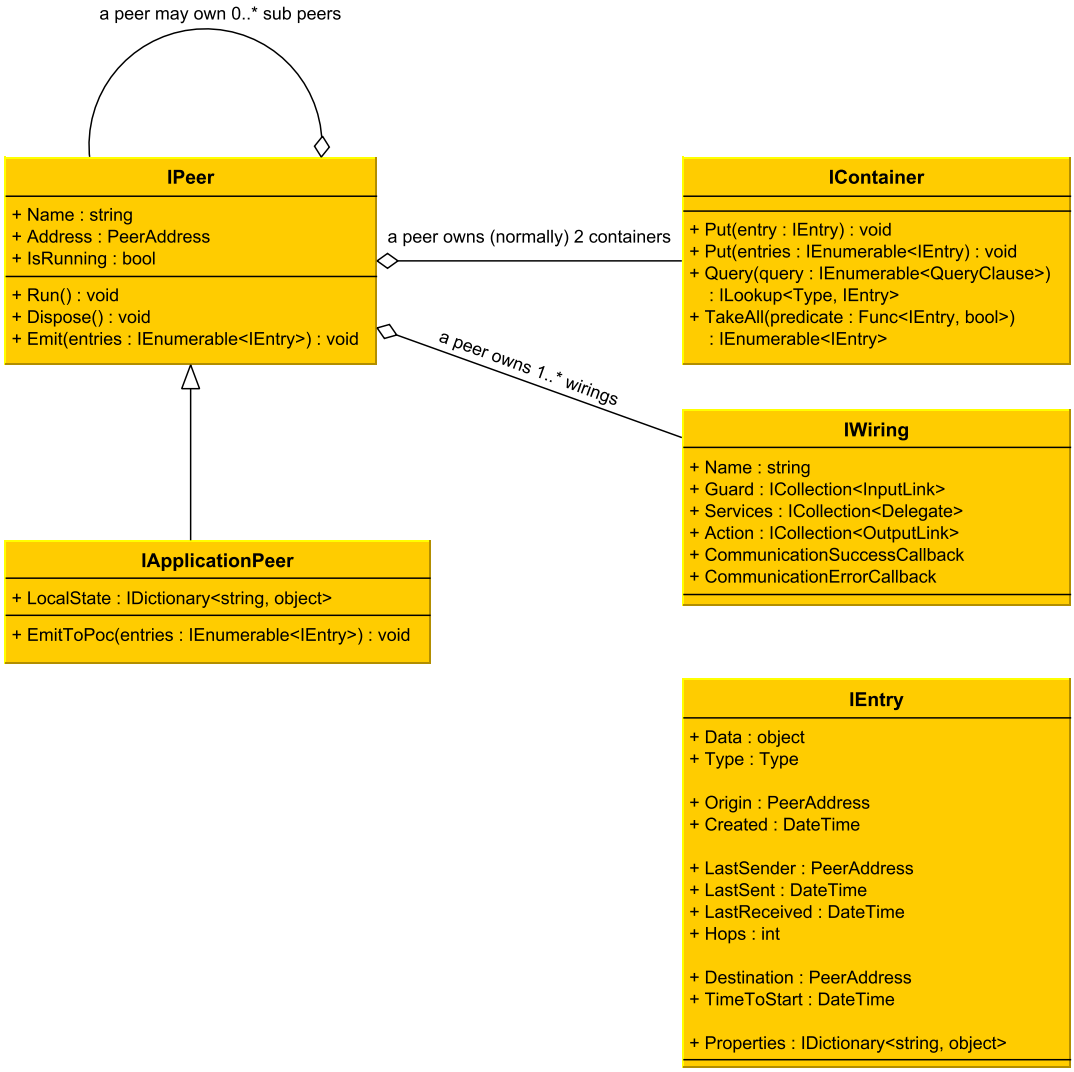

Figure 5.1: UML class diagram of main interfaces

configuration, concrete implementations have to be selected. An `IPeerFactory` combines an `IContainerFactory` and an `ICommunicationLayerFactory` which in turn select the underlying implementation classes by returning concrete class instances. It is also possible to register those implementations at a dependency injection framework. The framework core provides a `DefaultPeerFactory` which is configured to use the FIFO container and the XcoAppSpace communication layer.

The `IWiring` interface holds all the information of a single wiring: an optional name, a guard (list of guard links, called “input links” in the source code), a list of services to be called, an action (list of action links, called “output links” in the source code) and also two callback

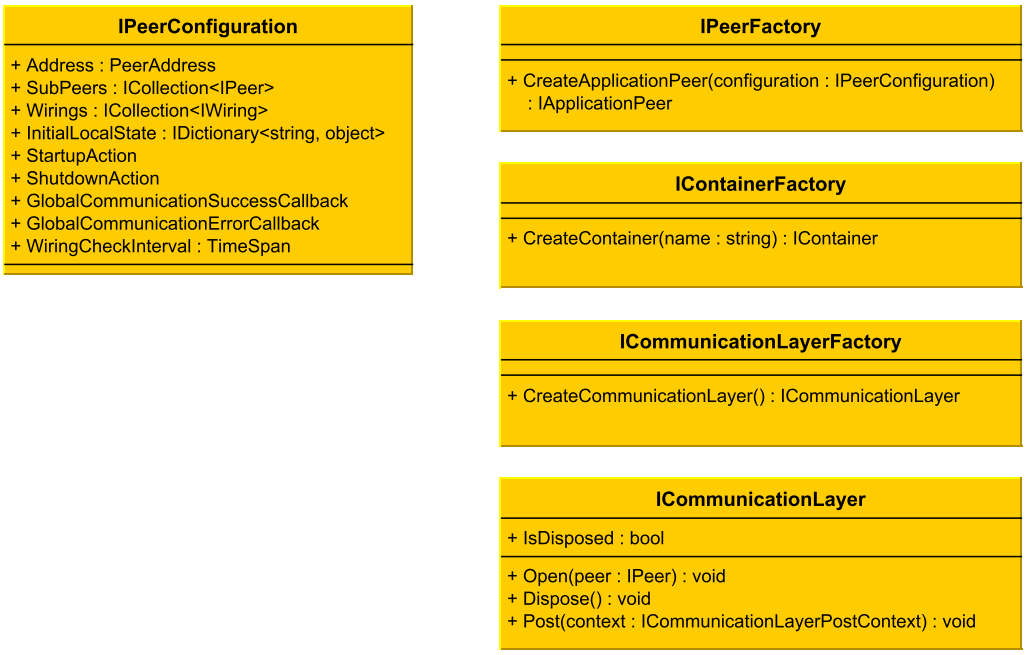

Figure 5.2: UML class diagram of additionally important interfaces

functions which are called in case the remote communication started by this specific wiring succeeded or failed. Guard and action links are simple DTO types which each hold a query clause for a container query. Guard links furthermore have a source property, identifying the source container, action links have an additional destination property, identifying the destination container or peer. Note that `IWiring` does not contain a method like `RunWiring(...)`, as the actual logic resides inside the `IPeer` implementation class. More on the implementation of wirings can be found in section 5.3.

`IContainer` defines the interface to a container as specified in the functional requirements. It offers a method to put new entries into the container and two more to query entries. The input argument of the query method, the `QueryClause` class, is a simple DTO type (see figure 5.3).

The core framework currently provides a single container implementation: the `FifoContainer`, an in-memory container which returns queried entries in a first-in-first-out ordering. Multiple decorators extend its functionality with thread-safe access, auditing and sanity checks on new entries (e.g. if an entry implements `ISerializable`). In the future we will see more containers coming up, most probably a persistent, database-backed container which guarantees fault tolerance by enabling transactions.

The last of the four basic interfaces of the PeerSpace core framework is the `IEntry` interface which is also merely more than a DTO class containing an entry's application and coordi-

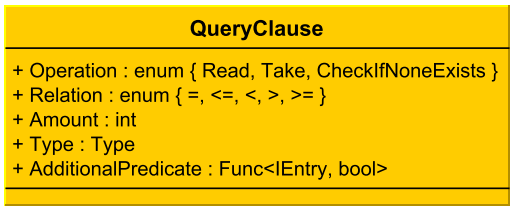

Figure 5.3: UML class diagram of `QueryClause`

nation data. The interface has a single implementation class `Entry` only.

The UML package diagram in figure 5.4 shows all PeerSpace.NET core packages (= .NET namespaces). The main package contains all interfaces which are often used by developers. The communication layer package holds all communication layer related classes and the AppSpace-based default implementation in the sub package XcoAppSpace. All configuration-related interfaces and classes are within the configuration package. Container- and peer-related interfaces and classes are inside their corresponding packages, however, some of the interfaces reside in the main package due to their importance (e.g. `IPeer`). Last but not least a utilities package contains various auxiliary classes.

5.2 Peer Implementation Details

In this section we present important details of the application peer implementation class. `ApplicationPeer` derives from `PeerBase` and implements `IPeer` as well as `IApplicationPeer`. With the name goes the behavior of application peers, that is: having two containers, namely PIC and POC, as well as having inner-peer wirings between those two and having some inter-peer wirings with push-behavior in addition. Application peers may also own any number of sub peers. Last but not least all application peers hold so-called local state, a simple replacement for space peers, however, the state is only kept in the local peer instance. This state may later be read and updated by services. All the necessary configuration options are provided to the implementation class by the `IPeerConfiguration`. Additionally the class receives a container and a communication layer factory. Those are necessary to completely abstract the `ApplicationPeer` class from any underlying technology, which reduces the cohesion between peer and container. Note, that sub peers initially do not contain any inter-peer wirings. It is the parent peer developer's responsibility to define wirings from the sub peer to the parent peer within the parent peer configuration. On system startup, those wirings are moved from the parent peer to the sub peer and they are converted by the framework from "pull-wirings" to "push-wirings". Altogether, the following steps are executed when a peer starts:

- Create the PIC and POC container
- Move "pull-wirings" of parent peer to corresponding sub-peers in order to create "push-wirings" (enables inner-peer wirings between a sub-peer's POC and the parent peer)

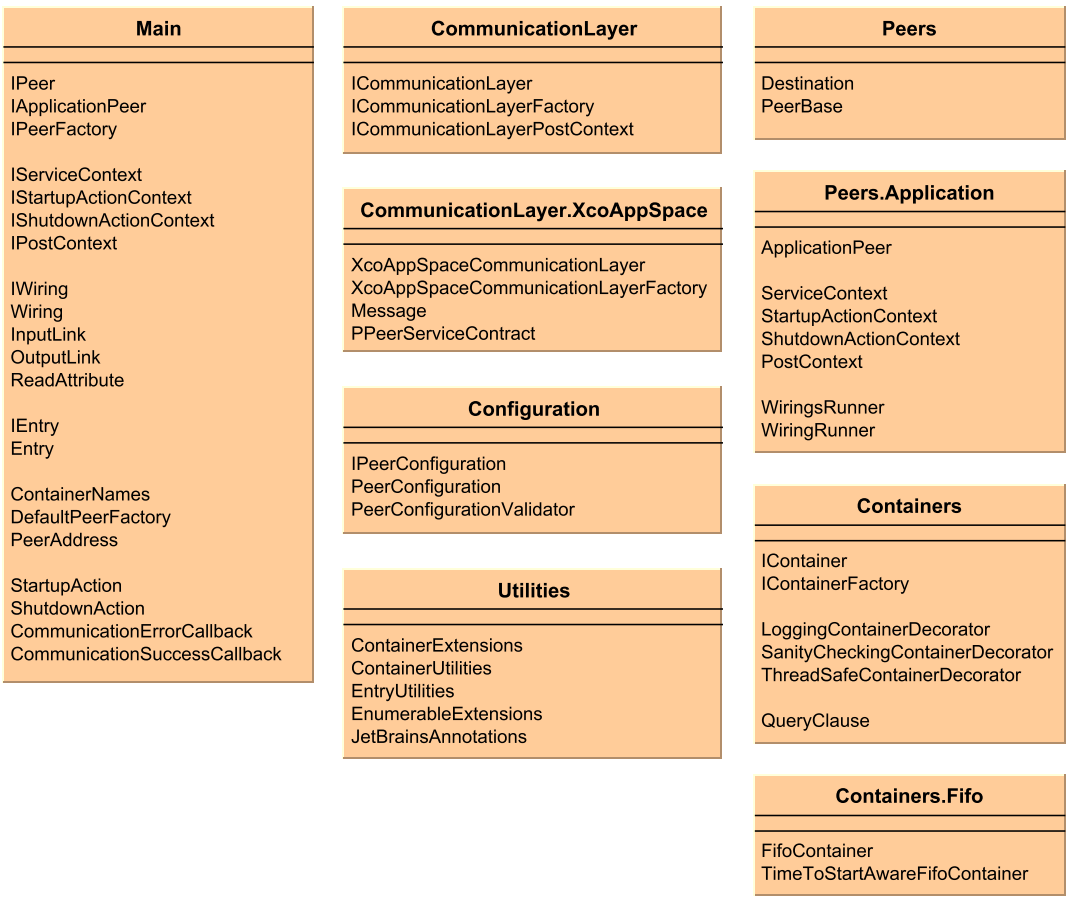

Figure 5.4: UML package diagram of the PeerSpace core

- Split wirings into inner-peer and inter-peer wirings
- Connect the wiring's source/destination information (e.g. "take from PIC" or "move to POC") to actual container instances (i.e. replace logical names like "PIC" with physical container addresses).

Afterwards the peer is ready, however, not yet running. A call to the peer's `Run()` method additionally performs:

- Initialize the communication layer (`ICommunicationLayer.Open(...)`)
- Run `StartupAction` method of the peer and all its sub-peers
- Start the wiring check interval timer

Now the peer is actually running and wirings fire as soon as their respective guard is fulfilled. Note, that the framework's wirings runner engine runs within its own thread to increase multi core usage. Last but not least all peers implement the `IDisposable` interface. The `ApplicationPeer` simply disposes all existing sub-peers and closes the communication layer on disposal.

5.3 Wiring Implementation Details

This section discusses the details of the wiring implementation. Both types of wirings (inner-peer and inter-peer) are basically handled equally in the PeerSpace implementation. The only difference is, which container is used as the guard's source container. As mentioned in an earlier section, a guard's input links are nothing more than container query clauses. Therefore the only difference between inner-peer and inter-peer wirings is to which container the query belongs: PIC or POC.

As "stationary queries" (blocking queries against a data source) are not really suitable and are not very well performing, the PeerSpace approach is to check all guards after each insertion into the corresponding container, i.e. if new entries are put into the PIC, the framework ensures that all guards of inner-peer wirings are checked for fulfillment. This task is handled by the wirings runner engine (encapsulated by the `WiringsRunner` class). In case it finds a fulfilled guard, the corresponding wiring is executed. A wiring execution consists of the following sequence of actions:

- The query results of the guard are put into an entry collection (essentially, an in-memory FifoContainer)
- The wirings' services are executed and may use entries from the entry collection as well as emit new entries into the entry collection
- The wiring's action (a collection of action links) is executed. In contrast to the guard, action links do not block if they cannot be fulfilled, instead they are skipped.

All these steps are executed sequentially, afterwards the next wiring's guard is checked until all wirings have been examined. Note: as a performance improvement we tried to check the guards of all wirings in parallel, however, this approach failed to meet the desired performance gain due to the vast thread management overhead.

If at least one wiring fired during the first loop iteration, all the wirings are checked again - as they might be able to fire due to newly emitted entries (or as they are able to fire multiple times anyway). An example: five entries of type A are put into the PIC container, there are three inner-peer wirings with the following guards: take less than or exactly 3 A entries, take exactly one A entry and the third wiring queries another type. This leads to an execution of wiring 1 taking 3 entries, wiring 2 taking 1 entry and then again wiring 1 taking 1 entry. All guards are checked a third time, no wiring fires anymore and the `WiringsRunner` rests until new entries are put into a container.

The whole wirings runner engine is triggered asynchronously after new entries have been put into the container. This also entails that the container has to be thread-safe. One more note: if the container does not support transactions, unexpected behavior may occur: during the execution of a wiring, the container is not locked and additional entries can be added to the container which may be observed by the following wiring in the same loop iteration.

5.4 Communication Layer Implementation Details

The communication layer has been implemented using the Xcoordination Application Space framework [58]. The Application Space is based upon Microsoft's Concurrency and Coordination Runtime (CCR) [59], more information can be found in section 3.2.2. Implementing the `ICommunicationLayer` interface using the Application Space has been straightforward. First, the CCR port is opened:

```
_space = new XcoAppSpace(configString);
_space.RunWorker((PPeerServiceContract)this, peer.Name);
```

Listing 5.1: XcoAppSpace initialization

Instead of opening various workers for different entry types, a single worker of type `PPeerServiceContract` is opened, which listens for objects of type `Message`. A message is a collection of `IEntry` objects. On receipt of a message, we update the `LastReceived` and `Hops` properties of the received entries and forward them to the peer's PIC container.

The asynchronous posting of entries to other peers guarantees - even on disposal of the communication layer - that either the specified success callback or error callback action is called. This is realized using atomic increments and decrements of a `_currentlyTransmittedMessages` integer field. A call to `commLayer.Dispose()` will block until all posts have finished.

Another interesting implementation detail is that it is necessary to obtain the Application Space communication port in a thread-safe way:

```
lock(_resolveOrConnectLock)
{
  var url = peer.Address.Url;
  return
   _space.Resolve<PPeerServiceContract>(url) ??
   _space.ConnectWorker<PPeerServiceContract>(url, url);
}
```

Listing 5.2: Thread-safely obtaining the XcoAppSpace communication port

Unfortunately, the `ConnectWorker<TContract>(url)` method is not thread-safe. If we would not use any locking, two parallel posts, could try to connect to the same worker at the same time. The second attempt would throw an unexpected exception.

5.5 Error Handling

There are two possible locations where errors may occur:

- Domain logic: exceptions during the execution of a service
- Coordination logic: exceptions during the exchange of entries (e.g. another container cannot be reached)

First, errors may occur during service execution. This, however, is deliberately not the realm of the Peer Model, which is responsible for handling coordination logic only. Therefore, the PeerSpace core framework simply ignores all service exceptions and thereby forces services to do all the necessary domain logic exception handling themselves. Nevertheless, for better debuggability, all errors are logged into the log file. Furthermore, services are able to emit service exception entries, however, in the current PeerSpace implementation, this is not done automatically.

Secondly, errors may occur in the communication layer due to all the well-known reasons of network unreliability, e.g. due to network partitioning another peer may become unreachable. Instead of automatically creating error entries, we decided to introduce a new error handling mechanism (see section 4.2.7), which is more robust and which is able to dynamically decide at runtime how to react in case of an exception. To react, developers are able to add an error callback method for each wiring or one for the whole peer. The registered method is given the exception and a `IPostContext` (see listing 5.3), which allows the user to (a) retrieve extended information about the failed communication attempt (first four properties) and (b) to react in a certain way (e.g. retrying the delivery immediately or at a later point, sending the entry to a different location instead, or maybe even by adding an error entry to the POC).

```
public interface IPostContext
{
  IPeer Peer { get; }
  PeerAddress Destination { get; }
  IEnumerable<IEntry> Entries { get; }
  int Attempt { get; }

  void RetryImmediately(PeerAddress newDestination = null);
  void RetryAfter(TimeSpan timeSpan, PeerAddress
     newDestination = null);
}
```

Listing 5.3: The IPostContext interface

5.6 Usage Example for the Core Framework

Finally we will present a small usage example of the core framework. We use a very simple scenario: the well-known request/response pattern. The aim in this section is to show, how to

create peers and run them using the core framework. Also the section will make it clear that a developer-friendly API is necessary.

First we will present how to structure the main program of a peer process:

```
var name = args[0];
var port = int.Parse(args[1]);

var peerFactory = new DefaultPeerFactory();

var config = CreatePeerConfig(name, port);
using(var peer = peerFactory.CreateApplicationPeer(config))
{
  peer.Run();

  // Peer is now running...
  // you may put entries from an input source into the PIC:
  var line = Console.ReadLine();
  peer.Emit(line); // add entry of type string

  Console.ReadKey(); // exit after next key stroke
}
```

Listing 5.4: Structure of the main routine of a peer process

Name and port are input parameters of the process. The `CreatePeerConfig` methods for the request and the reply peer are shown in the following two listings. Note that the peer is automatically converting the emitted string `line` into an `IEntry<string>` with all co-data properties set accordingly. To point out that Peer Models can be transformed into PeerSpace applications very directly, we show the Peer Model of the request/reply pattern in figure 5.5 before presenting the source code.

The request peer features two inner-peer wirings: one that is simply forwarding received string entries to the POC (this type of wiring is called “move wiring”), from where an additional inter-peer wiring forwards the string entry to the reply peer. When implementing the peer, we are able to achieve a minor simplification by putting the input line directly into the request peer’s POC, and thereby save the move wiring. The second wiring (W2) uses a service to output received entries of type “newStr”.

The reply peer consists of only a single inner-peer wiring (W3) which makes use of a transformation service to transform received string entries into entries of type “newStr” (i.e. it provides some service, e.g. converting the string, translating it, etc.) and then immediately replies the “newStr” to the origin of the originally received string entry. The service utilizes the Destination property of the “newStr” entry, and set it to the origin address of the received “str” entry.

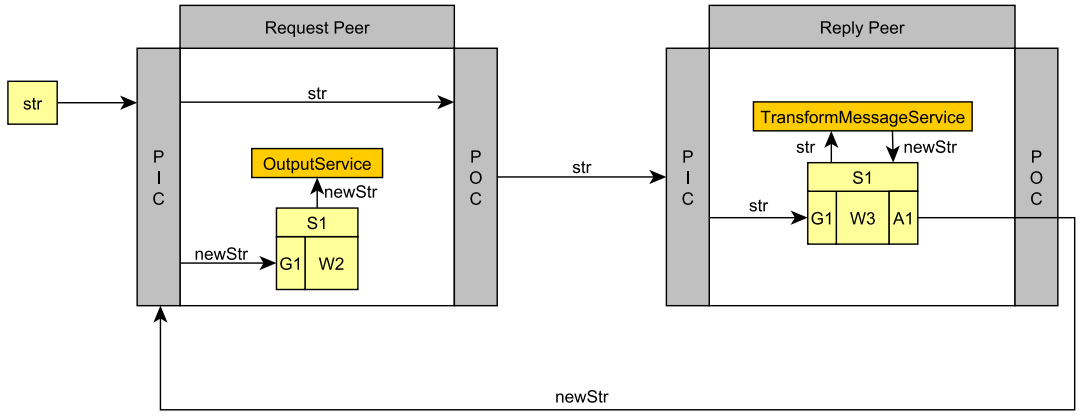

Figure 5.5: Peer Model of the request/reply peer pattern

Now, the implementation simply converts the Peer Model wirings into peer configuration settings:

```
// Request peer:
var rp = new PeerConfiguration{Address=requestPeerAddress};

var w1 = new Wiring("inter-peer wiring");
w1.Guard.Add(new InputLink{Type=typeof(string), Source="POC"});
w1.Action.Add(new OutputLink{Type=typeof(string),
   Destination=replyPeerAddress});
rp.Wirings.Add(w1);

var w2 = new Wiring("inner-peer wiring W");
w2.Guard.Add(new InputLink{Type=typeof(TransformedString),
   Source="PIC"});
w2.Services.Add((Action<TransformedString>) OutputFunction);
rp.Wirings.Add(w2);

// Request peer's OutputFunction implementation:
void OutputFunction(TransformedString str)
{
  Console.WriteLine("Response of reply peer: " + str.String);
}

// Reply peer:
var rp = new PeerConfiguration{Address=replyPeerAddress};

var w = new Wiring();
```

```
w.Guard.Add(new InputLink{Type=typeof(string), Source="PIC"});
w.Services.Add((Action<IServiceContext, IEntry<string>>)
   ServiceDelegate);
rp.Wirings.Add(w);

// Reply peer's ServiceDelegate implementation:
void ServiceDelegate(IServiceContext ctx, IEntry<string> str)
{
  var newStr = Transform(str);

  // ReplyTo sets the Destination property to the Origin
  // property of the first argument
  ctx.ReplyTo(str, newStr);
}
```

Listing 5.5: Echo (request/reply) peer system example

One immediately notes, that the configuration effort for such a simple example is huge and even comparable to directly programming against a socket library. However, the core framework does not target the developer, but code generation tools and wants to provide a stable but extensible interface instead of a very intuitive one. The following chapter will show how the usability-focused API will improve the code and reduce the given example configurations to mere one-liners.

CHAPTER 6

API Design & Implementation

The previous chapter covered the design and implementation of the PeerSpace core framework. This chapter discusses the usability-focused PeerSpace API. Beginning with a requirements analysis and an evaluation of various design approaches, the finally selected implementation will be presented. A detailed example completes the chapter.

6.1 Requirements of the Peer Space API

Previous space based technology frameworks often suffered from a lack of API usability. This played a major part in why existent space based frameworks did not reach mainstream adoption yet [53], which is why TU Vienna's Space Based Computing research group started to delve into the research field of API usability measurement recently.

A main focus of this master thesis is to combine the two research areas of space based computing and API usability and to equip the PeerSpace implementation with a usability-focused API. This will improve the developer-friendliness of space technology and at the same time provides testing material for the research group's approach of measuring and comparing API usability with quantitative evaluation methods.

An API should not only be developer friendly, it should also comply with additional characteristics. Altogether the literature [71] [72] [73] [74] most prominently features the following rules and heuristics, which we will discuss briefly. Some of those general framework guidelines have to do with API usability as well, others are guidelines of a different kind:

Stability of the public API
: Stability is very important for APIs. Once released to the public it is very hard to make changes without breaking existing code.

Structure should follow use cases
: A focus on use cases helps the developer to use and understand the API, furthermore it gives the API additional stability.

Consistency (e.g. parameter ordering)
Inherent consistency of the API as well as consistency with conventions of the environment (e.g. .NET conventions) further improve API usability.

Small ("as small as possible, but no smaller")
Additional features can always be added, however, rarely removed. Therefore a small API helps to reach stability.

Names are important
Giving the right name to an entity is an important task for self-documenting source code. It supports the readability of the source code very much. APIs additionally gain usability as well, because a developer may easier write new code if the interface to use is "well-named".

Hide implementation details
Implementation details should not manifest in the public API, this would undermine the whole purpose of separating the interface from the implementation.

Easy to use - hard to misuse
This item promotes API usability. Easy to use & hard to misuse is a basic rule for any interface design.

Follow the principle of least astonishment
Developers should never be surprised by the API. Whenever developers are forced to read the documentation in detail (e.g. because the first idea coming to the mind when reading code written against the API is ambiguous or even plainly wrong) the API designers failed.

Fail fast
Whenever the user misuses the API, errors should be reported as soon as possible (during compile-time is better than during deployment-time, however, during deployment or on first start is still better than during run-time).

Prevent string parsing (provide programmatic access)
Never force API users to parse information they need, instead provide programmatic access. String parsing is the source of many errors and although the API looks stable, even minor changes within the string layout may cause failures in existing code.

Self-documenting and good documentation in general
An API should be self-documenting for most tasks, particularly for standard use cases. Good documentation should support the API and give further help for rare use cases as well as background information.

All of those characteristics have been considered for the design of the PeerSpace API. In the following chapter we are going to compare various technical approaches in order to find the best suited option.

6.2 Technological Approach

In chapter 5 we've already seen that the API must be able to build an instance of type `IPeerConfiguration` which is then used to create `IPeer` instances. Therefore we are looking for a configuration approach which *builds* a *configuration object*. A list of different configuration builder approaches and their respective advantages and disadvantages in regards to API usability can be found in [75], amongst them:

- Configuration object
- XML configuration file
- Annotation-based configuration
- Fluent interfaces

A simple configuration object is already implemented in the core framework (`IPeerConfiguration`), it is however very clumsy to use and its verbosity for simple tasks is not favorable. The only advantage of using the configuration object directly is, that it is by definition providing full access to all features of the PeerSpace core. If any feature is added to the core and the API is not updated yet - or worse: not in development anymore - the API would immediately be rendered useless if it is not possible to combine both worlds. Therefore, we decided that the API should improve the usability but still allow access to the actual configuration object before instantiating the peer. In [75] the XML configuration file approach, the annotation-based approach and the fluent interface approach are compared with each other in regards to API usability. The results show that in general API usability is highest with annotations and lowest with XML configuration files. Fluent interfaces are somewhere in between.

XML configuration has one additional major disadvantage considering the PeerSpace.NET API. The structure of an XML file is very static, it kind of resembles the C# configuration object for the core implementation. Of course there is the advantage that changes of the configuration are possible without the need to recompile the code, however, this is not a primary use case for the PeerSpace. As it is also the approach with the lowest API usability in general, we chose to not implement the PeerSpace.NET API with XML configuration files.

Although the annotation-based approach has the best overall results, unfortunately current high-level languages are not powerful enough to express most of the PeerSpace's use cases in an easily and understandable way (especially without using the string parsing anti-pattern). The following attempt to make use of a pure annotation-based API immediately points out the problems with annotations:

```
// Field with annotations
[Peer(Name="MyPeer", Port=80)]
[Wiring(Guard=???, ...)]
private Peer _myPeer;
```

Listing 6.1: Attempt of a pure annotation-based API

In many high-level languages, especially Java and C# the language construct of annotations (called "attributes" in C#) allow only compile-time constant literals (i.e. no objects) as argument values. We would have to put everything in strings which is (a) not very developer friendly and (b) even worse: not verifiable during compile-time. Furthermore, the annotations are very static and would not allow using peers within different configurations, which would not be desirable.

This leaves us with the very promising fluent approach. A side note in the aforementioned paper points out that fluent APIs resemble the builder pattern best. As the PeerSpace API's aim is to *build* a peer configuration object (for the builder pattern see [67]), it is likely to be a good option. Fluent interfaces are not yet common knowledge in the software development world, which is why the following section will give a short introduction.

6.3 Fluent Interfaces

Conventional APIs provide a set of methods with each method standing individually. Such an API style is called command-query API. In some problem domains (amongst them also the builder pattern) readability increases noticeably by using a fluent API instead of a command-query API [76].

Fluent interfaces are designed with the goal of increased readability of a whole expression. Very often the fluent interface translates internally to the traditional command-query API. A little example immediately shows the benefits of this approach (see listing 6.2). Both examples create a calendar event and set various properties. While the command-query API looks clunky and very verbose, the fluent interface can be read like an English sentence. Very rare methods with no fluent equivalent can still be called in the traditional way after the initial creation of the event.

```
// Traditional command-query API:
var ev = new CalendarEvent();
ev.SetTitle("My lecture");
ev.SetFromDate(2010, 9, 4);
ev.SetToDate(2010, 9, 5);
ev.AddAttendee("Dominik");
ev.AddAttendee("Andrea");
...

// Fluent API:
var ev = Event.Create("My lecture")
              .From(2010, 9, 4).To(2010, 9, 5)
              .With("Andrea", "Dominik");
ev.RarelyUsedProperty = "x509";
```

Listing 6.2: Comparison: command-query API vs. fluent API

A fluent interface is implemented using multiple interfaces. The event creation interface could look like follows:

```
interface IFluentEvent
{
  IFluentEventWithFromInfo From (int year, int month, int day);
}

interface IFluentEventWithFromInfo
{
  IFluentEventWithToInfo To (int year, int month, int day);
}

class Event : IFluentEvent
            , IFluentEventWithFromInfo
            , IFluentEventWithToInfo
{
  public static IFluentEvent Create (string eventName) {
    var cev = new CalendarEvent();
    cev.SetTitle(eventName);
    return new Event(cev);
  }

  private readonly CalendarEvent _cev;
  private Event(CalendarEvent cev) {  _cev = cev; }

  public IFluentEventWithFromInfo From (int year, int month,
     int day)
  {
    _cev.SetFromDate(year, month, day);
    return this;
  }

  ...

  // .With() finally returns _cev instead of this.
}
```

Listing 6.3: Implementing a fluent interface

The `From` method from the `IFluentEvent` interface returns another interface of the different type `IFluentEventWithFromInfo`, and so on. Due to intelligent code completion in modern IDEs and code editors the developer is guided through the event creation process and is not required to consult the documentation. All the interfaces may be implemented by the same implementation class, like shown in the code listing above (class `Event`).

6.4 API Design

The final API design is a combination of a fluent API with some additional attributes. This mixed approach enables us to give the user the best of both worlds: fluent, sentence-like APIs for wiring configurations and easy-to-use attributes for tagging certain code elements. According to [76] it is recommended to design fluent interfaces using a state machine-like diagram as shown in figure 6.1. Such a state machine shows the transitions from one interface to another and by which method calls those interfaces may be reached.

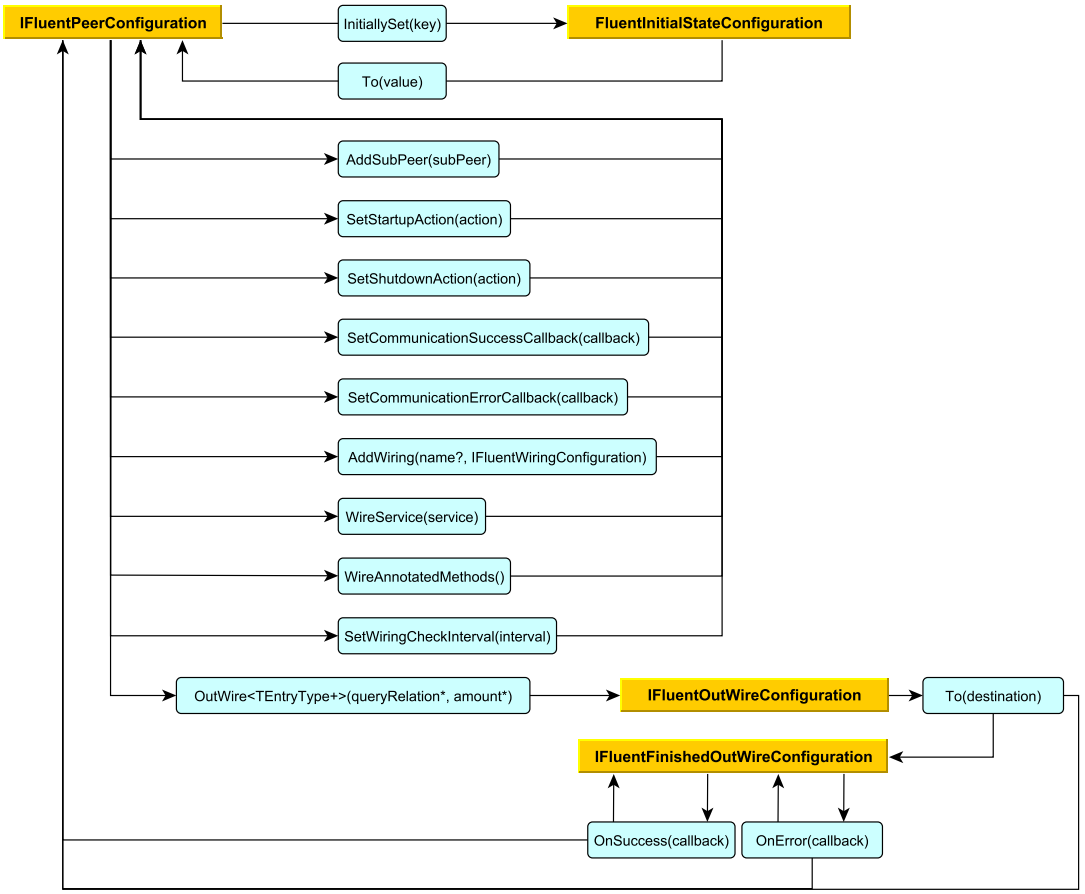

Figure 6.1: State machine of the fluent interface

The main reason for the development of the fluent interface is the complicated and verbose way of defining wirings in the core framework. All other configuration aspects of the PeerSpace core are already very simple and are therefore 1 to 1 mappings in the fluent API (among them: `AddSubPeer`, `SetStartupAction`, `SetShutdownAction`, `SetCommunicationSuccessCallback` as well as `SetCommunicationErrorCallback`. To add initial local state to the peer you can use the pretty straightforward `InitiallySet(key).To(value)`

method chain. The `SetWiringCheckInterval(interval)` allows you to set an interval in which the wiring guards are additionally checked (required for TimeToStart only, see section 6.4.5 for more information).

In the following sub sections we will discuss how to configure wirings of different kinds in a developer-friendly way using the new PeerSpace API.

6.4.1 Configuring simple Inner-Peer Wirings

There are various ways to configure wirings with the PeerSpace API. The easiest way is to use the `WireService` method which allows to define a method delegate (= service method) as argument. The method delegate is automatically converted into a wiring as follows:

- Each `IEntry<T>` or `T` argument is converted into an Exactly-One-Of-Type-T input link
- Each `IEnumerable<IEntry<T>>` or `IEnumerable<T>` argument is converted into an Exactly-One-Or-More-Of-Type-T guard link.
- If the `[Read]` attribute is used in front of a parameter, the entry is only read (not taken) from the entry collection.
- If the `[ReadFromPic]` attribute is used in front of a parameter, the guard link is set to `QueryOperation.Read` and the entry is only read (not taken) from the source container (PIC).
- If the `[IfNone]` attribute is used in front of a parameter, the guard link is set to `QueryOperation.CheckIfNoneExists` and the parameter is always filled with `default(T)`.
- An `IServiceContext` argument is filled by the framework. It allows the developer to query context information and to emit new entries from within a service.

Of course the method delegate itself is established as the service call. All entries which are emitted by the method delegate (i.e. which are in the entry collection at the end) are by default put into the POC. Note: `IEntry<T>` allows to query the co-data of an entry as well. Also, there is no way to set success and error callback methods for inner-peer wirings in the fluent API as this is a no-use-case. The following two listings show examples to better underline the service wiring mechanism:

```
// Reads 1 string from the PIC into the EC
// Service takes 1 string from the EC
void Service1 (IServiceContext ctx, [ReadFromPic] string s);
```

Listing 6.4: Automatic service wiring example 1

```
// Takes one or more strings from the PIC into the EC
// Services takes one or more strings from the EC
// Service needs access to the co-data of the entries
void Service2 (IServiceContext ctx,
   IEnumerable<IEntry<string>> strs);
```

Listing 6.5: Automatic service wiring example 2

Note how the additional attributes make it very easy for developers to express certain statements. If you only wire a single service (e.g. using `WireService`) there is no need to utilize the `Read` attribute - if a wiring includes only a single service, there is no need to only read from the entry collection. See section 6.4.4 for an example wiring multiple services.

6.4.2 Auto-Discovery of Inner-Peer Wirings & Callback Actions

Additional attributes remove the need of calling `WireService` multiple times and the need to call the various `Set*Action` and `Set*Callback` methods. It is possible to put one of the following attributes on any method in your own class:

- `[Service(EmitTo="...")]` - wires the method automatically according to the rules in the previous section 6.4.1.
- `[StartupAction]` - service which is executed once after the peer has started.
- `[ShutdownAction]` - service which is executed once before the peer is shut down.
- `[CommunicationSuccessCallback]` - method which is set as the global callback for all successful communication attempts.
- `[CommunicationErrorCallback]` - method which is set as the global callback for all unsuccessful communication attempts.

Afterwards a simple call of `WireAnnotatedMethods(objOfClass)` will auto-wire all discovered methods accordingly. This is a very handy way to wire almost any aspect of your peer, it also allows you to define the action link's destination property for services (in contrast to `WireService()`). Note that the methods must have the following signatures in order to successfully discover them:

```
[StartupAction]
void OnStartup (IStartupActionContext ctx) { ... }

[ShutdownAction]
void OnShutdown (IShutdownActionContext ctx) { ... }

[CommunicationSuccessCallback]
void OnCommSuccess (IPostContext pc) { ... }
```

```
[CommunicationErrorCallback]
void OnCommError (IPostContext pc, Exception e) { ... }
```

Listing 6.6: Required signatures for auto-wired methods

6.4.3 Configuring Inter-Peer-Wirings

Inter-peer wirings can be created using the `OutWire` fluent method. It is possible to specify the type, the query relation (defaulting to `Exactly`), the amount of entries (defaulting to 1) and where to move them. Furthermore one can specify wiring-specific success and error callbacks. Some examples will describe the process even better:

```
// Move exactly one entry of Type EntryType to destinationPeer
.OutWire<EntryType>()
  .To(destinationPeer)

// Move at least two entries of EntryType to destinationPeer,
// however, leave them in the EntryCollection
// on error call the given error callback
.OutWire<EntryType>(QueryRelation.MoreThanOrExactly, 2)
  .To(destinationPeer, QueryOperation.Read)
  .OnError(MyErrorCallback)

// Overload for multiple types
.OutWire<EntryType1, EntryType2>(
    QueryRelation.Exactly, 1, // EntryType1 query
    QueryRelation.MoreThanOrExactly, 2) // EntryType2 query
  .To(destinationPeer)
```

Listing 6.7: Inter-peer wiring examples

The last of the three examples wires exactly one entry of type EntryType1 and at least two entries of type EntryType2 to the given destinationPeer address.

6.4.4 Configuring Advanced Wirings

In case the wiring options presented up to now are not powerful enough there is another option which is a little bit more verbose, however, allows even the most complicated wirings. It starts by using the `AddWiring()` fluent method which in turn is the starting point for an additional configuration API. Once more we will start with an example:

```
.AddWiring("myInnerPeerWiring", w => {
  // Start with the guard
  // (ET ... EntryType)
  w.Take<ET1>(); // Take exactly one ET1
```

```
  w.Take<ET2>(QueryRelation.Exactly, 3); // Take exactly 3 ET2
  w.Read<ET3>(); // Read exactly one ET3
  w.IfNone<ET4>(); // Fire only if there is no ET4

  // Call multiple services
  w.Call<ET1, IEnumerable<ET2>>(FirstServiceMethod);
  w.Call<ET3>(SecondServiceMethod);

  // Finish with the action
  w.Move<RT1>().ToPic(); // move exactly one RT1 to the PIC
  w.MoveToPoc(); // move the rest of the results to the POC
});
```

Listing 6.8: Advanced wiring example

Note that it is also possible to create inter-peer wirings and wirings incorporating sub-peers using the very same API (see the following listing for examples). The complete state machine for the advanced wiring possibilities is shown in figure 6.2.

```
.AddWiring("myInterPeerWiring", w => {
  // Take exactly one RT1 from the POC
  // (which has set the DoNotSend property to false)
  w.Take<RT1>(e => e.Properties["DoNotSend"] == false)
   .FromPoc();
  // Fire only if there is no RT4 in the POC
  w.IfNone<RT4>().FromPoc();
  // Move the selected RT1 entry to the given peer
  w.MoveTo("peer://url");
  // On any communication error call the MyErrorCallback
  w.OnError(MyErrorCallback);
});

.AddWiring("mySubPeerWiring", w => {
  // Take exactly one RT1 from MySubPeer's POC
  w.Take<RT1>().From("MySubPeer");
  // Move it to our own POC
  w.MoveToPoc();
});
```

Listing 6.9: More advanced wiring examples

6.4.5 Timer-triggered Wirings

The `TimeToStart` property allows to create timer-triggered wirings. Such wirings can be used for various purposes, most importantly for delayed processing and creating loops. In the

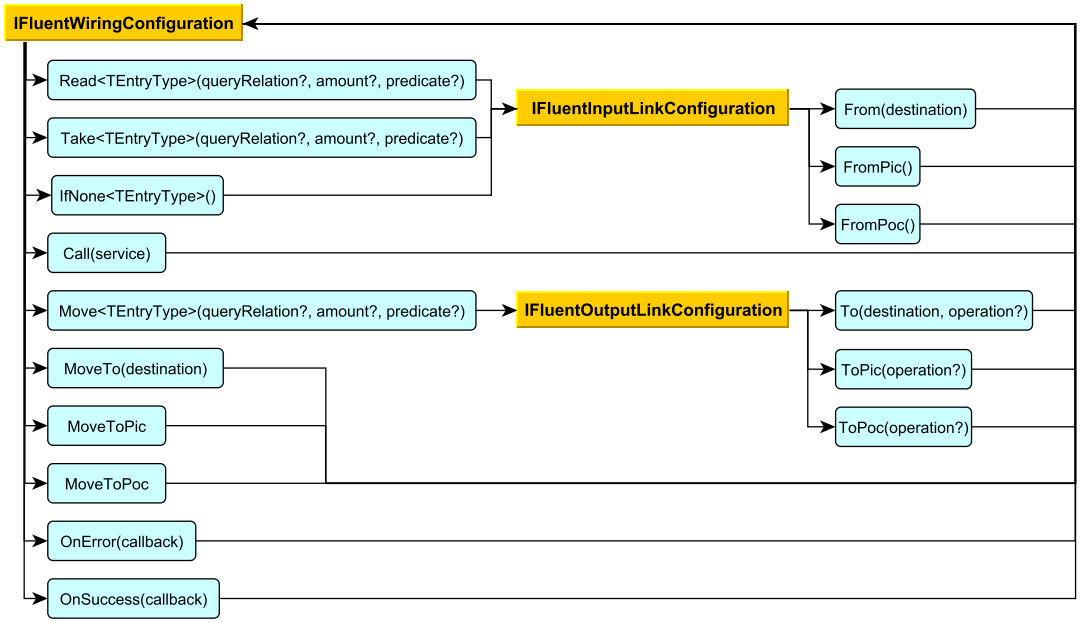

Figure 6.2: State machine of the wiring branch

following example we will show a peer which emits data every five seconds by utilizing the `TimeToStart` property:

```
// Required configuration:
.SetWiringCheckInterval(TimeSpan.FromSeconds(1))

// Auto-discovered methods:
[StartupAction]
void OnStartup(IStartupActionContext ctx)
{
  ctx.Emit(new Token());
}

[Service(EmitTo="PIC")]
void Process(IServiceContext ctx, Token t)
{
  var data = ...;
  ctx.EmitToDestination(receiverAddress, data);

  var inFiveSeconds =
     DateTime.Now.Add(TimeSpan.FromSeconds(5));
  ctx.EmitWithProperties(new Token(), new {TimeToStart =
```

```
      inFiveSeconds});
}
```

Listing 6.10: Loop with timer-triggered wiring

The auto-wired service emits the data and a fresh token with the TimeToStart property set to five seconds in the future. This delays the next activation of the wiring for five seconds and creates a loop. In order for TimeToStart-wirings to work properly, it is required to set a suitable wiring check interval. The interval defines the precision with which the systems works and should of course be smaller than the given time to start of the entries.

6.4.6 The ApplicationPeerBase Class

Most application peers will have common properties and a common structure. Therefore it makes sense to provide a base class for application peers which supports the following tasks:

- Save the own peer's address in a property
- Start building the peer configuration using this address
- Calling `WireAnnotatedServices(this)`
- Running & disposing the peer

The fluent API provides the following base class from which one can easily extend own peer classes:

```
abstract class ApplicationPeerBase<TImpl> : IDisposable
  where TImpl : IPeerFactory, new()
{
  ApplicationPeerBase(PeerAddress peerAddress)
  {
    PeerAddress = peerAddress;
  }

  PeerAddress PeerAddress { get; private set; }
  IApplicationPeer Peer { get; private set; }

  void Run(bool asSubPeer = false)
  {
    var peerConfig = PeerAPI.Create(PeerAddress)
                            .WireAnnotatedMethods(this);
    BuildPeerConfiguration(peerConfig);

    var peerFactory = new TImpl();
    Peer = peerFactory.CreatePeer(peerConfig.Finish());
```

```
    if(!asSubPeer)
      Peer.Run();
  }

  void Dispose()
  {
    if (Peer != null)
      Peer.Dispose();
  }

  virtual void BuildPeerConfiguration(IFluentPeerConfiguration
     peerConfig)
  {
    // derived classes override BuildPeerConfiguration for
    // additional configuration
  }
}
```

Listing 6.11: ApplicationPeerBase class

The `Run` method starts the peer configuration and calls the virtual `BuildPeerConfiguration` method which allows derived peers to extend the fluent configuration. Afterwards it creates and runs the peer. On disposal, the underlying peer is also disposed.

6.5 PeerSpace API Example

This chapter ends with a small example for the PeerSpace API, which should help future users and developers to use the PeerSpace API with ease. To illustrate the API we will implement the well-known master/worker pattern (according to the Peer Model given in figure 6.3) step by step.

Of course the Worker peer will be scaled out and instanced multiple times. On startup, each worker emits a so-called WorkerToken (WT) entry to the master peer. WorkItem (WI) entries are put into the Master peer's PIC from an external source (e.g. user input). The wiring W1 takes a WorkItem entry and a WorkerToken entry from the PIC and forwards the WorkItem entry to the corresponding worker peer. After the worker has finished processing the WorkItem entry using its WorkService, it returns a WorkResult (WR) entry as well as a new WorkerToken entry. Finally the master peer takes the WorkResult entry and outputs it in some form (using its Output service).

We presume that our three entry classes (WI, WR and WT) are already existent. The only requirement is, that they all implement the `ISerializable` interface. The worker peer will be configured first, as it is a little bit easier to implement. The code for the WorkerPeer looks like this (using the application peer base class):

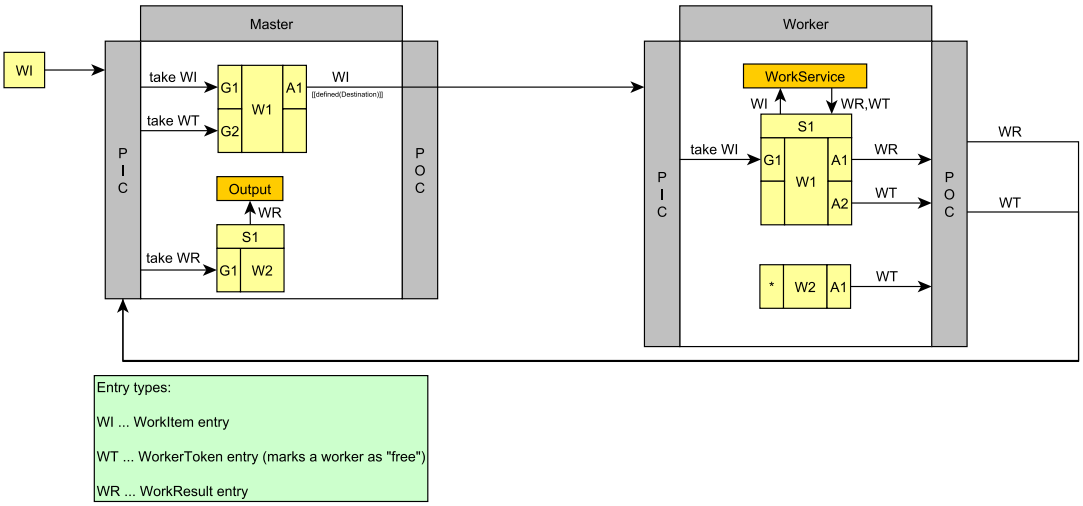

Figure 6.3: Peer Model of the master/worker pattern

```
class WorkerPeer<TImpl> : ApplicationPeerBase<TImpl>
{
  PeerAddress _masterAddress;

  WorkerPeer(PeerAddress address, PeerAddress masterAddress)
    : base(address)
  {
    _masterAddress = masterAddress;
  }

  void BuildPeerConfiguration(IPeerConfiguration configuration)
  {
    configuration
      .OutWire<WR>().To(_masterAddress) // inter-peer wirings
      .OutWire<WT>().To(_masterAddress);
  }

  [StartupAction]
  void Startup(IStartupActionContext ctx)
  {
    ctx.EmitTo("POC", new WorkerToken()); // W2
  }

  // W1
```

```
  [Service(EmitTo="POC" /* W1.A1 & W1.A2 */)]
  void ProcessWorkItem (IServiceContext ctx, WI wi /* W1.G1 */)
  {
    // W1.S1
    var wr = DoActualWork(wi);
    ctx.Emit(wr);
    ctx.Emit(new WT());
  }
}
```

Listing 6.12: Worker peer implementation

The peer is derived from a utility class in the fluent API which allows us to concentrate on the peer configuration. The peer's constructor retrieves all data that should be different in each peer instance. The `BuildPeerConfiguration` method configures the peer using the PeerSpace API. Observe the following properties:

1. The WorkerPeer class is completely independent from the actual PeerSpace implementation (e.g. which container implementation is used, which communication technology is used, etc.). This is defined by the generic parameter `TImpl`, where we can put any peer factory.

2. The WorkerPeer can be used as a template, by just modifying the constructor arguments we can instantiate multiple worker peers (= scale-out) without any other required modifications.

We are now able to instantiate as many worker peers using the `DefaultPeerFactory` as we like using the following code:

```
var address = new PeerAddress("peer://localhost:5010/Worker1");
var master = new PeerAddress("peer://localhost:5000/Master");
using(var worker = new WorkerPeer<DefaultPeerFactory>(address,
   master))
{
    worker.Run();
    Console.ReadKey(); // run worker until key has been pressed
}
```

Listing 6.13: Worker peer instantiation

Note that we could put the `worker.Run()` call already into the constructor of the worker peer if we wanted to. Also we do not have to specify `EmitTo="POC"` in the `Service` attribute as this is the default. Also note that we cannot combine the two `OutWire` calls into a single `OutWire<WR, WT>().To(...)` as this would create only a single inter-peer wiring expecting both a WR and a WT in order to fire. Therefore the wiring would not fire after the initial startup action which only produces a WT entry! Now let's implement the master peer alike:

```
class MasterPeer : ApplicationPeerBase<DefaultPeerFactory>
// directly used DefaultPeerFactory (for convenience)
{
  MasterPeer(PeerAddress address) : base(address)
  {
    Run();
  }

  void EmitNewWorkItem(WorkItem wi)
  {
    Peer.Emit(wi);
  }

  [Service] // W1
  void ProcessWorkItem(IServiceContext ctx,
     IEntry<WorkerToken> wt, WorkItem wi)
  {
    ctx.EmitWithDestination(wt.Origin, wi);
  }

  [Service] // W2
  void ProcessWorkItemResult(WorkItemResult wr)
  {
    // output result
  }
}
```

Listing 6.14: Master peer implementation

Due to the handy base class we automatically configure the wirings W1 and W2 according to the Peer Model shown in figure 6.3 using annotations on the service methods. The non-service method `EmitNewWorkItem` can be used from the main application to feed the peer with new work items (e.g. with user input or with work items read from a file).

We now already have a full-scaling master/worker pattern implemented. It would be nice to get additional robustness into the system: if the master cannot send the work item to the worker peer it should retry again after a minute. If however the second attempt also failed and the worker is still not reachable, the work item should be placed back in the PIC such that it will be scheduled for another worker.

Fortunately the PeerSpace has good support for error handling in inter-peer wirings, see section 4.2.7 for more information. Using the `ApplicationPeerBase` adding error handling is nothing more than adding an additional method.

```
...
[CommunicationErrorCallback]
void ProcessCommunicationError(IPostContext ctx, Exception ex)
{
  if (ctx.Attempt == 1)
  {
    // Try again
    ctx.RetryAfter(TimeSpan.FromMinutes(1));
  }
  else
  {
    // Second attempt also failed => put the work item back in
       the PIC
    var workItem = ctx.Entries
                      .Single(e => e.Type == typeof(WorkItem));
    ctx.Peer.Emit(workItem);
  }
}
...
```

Listing 6.15: Communication error handling

CHAPTER 7

Evaluation

This chapter evaluates the new PeerSpace API with regards to developer usability. This serves two objectives: first and foremost we can verify if the attempt to create a developer friendly API for the PeerSpace.NET framework has succeeded. Secondly, we test out the relatively new API usability measurement approach of Thomas Scheller [8] in order to provide empirical information for his PhD thesis.

7.1 Evaluation Setup

From the variety of evaluation methods presented in the Related Work chapter 2, we picked one from each of the two areas. From the area of qualitative methods we chose the modernized cognitive dimensions framework by Steven Clarke [7]. From the area of quantitative methods we chose Thomas Scheller's measurement approach [8]. Furthermore we will do an evaluation with some additional Visual Studio code metrics [77] to underline our results.

7.1.1 Technologies Used

We've presented various different .NET middleware frameworks in chapter 2, however, many of them were specialized for very specific use cases. WCF has been the only framework which claims to be a comprehensive communication framework. This lead to the decision that we are comparing the PeerSpace with the Windows Communication Foundation. In the final month of my thesis, Gernot Rumpold [26] compared the created PeerSpace API with other frameworks as well.

7.1.2 Implemented Patterns

For the purpose of the evaluation we're going to implement two different coordination patterns (both taken from [2]). The very common master/worker pattern (see example in the previous

chapter 6.3) will show how well coordination between components and scalability can be implemented with the respective framework. As a second example we go for a more traditional pattern and try to implement a request/multiple-responses example with both frameworks. We will see that developers face high complexity when implementing coordination patterns using standard SOA frameworks like WCF. It is much easier to use a coordination-focused framework to model such distributed systems. However, even the SOA-focused request/multiple-responses pattern, as we see in the next sections, is easier to develop using the new PeerSpace API.

Master/Worker Pattern

The typical master/worker pattern consists of a single master node which receives work requests (e.g. from user input) and distributes the work amongst several registered worker nodes in order to process the work items. An important feature is that the worker nodes are not tightly coupled to the master node, i.e. additional (distributed) workers are allowed to join and leave at any time. For the purpose of this example, we're not handling reprocessing of work items from crashed worker nodes. This pattern allows to scale-out very easily by simply adding more worker nodes. The corresponding Peer Model design has already been presented in the previous chapter (see figure 6.3).

Request/Multiple-Responses

This pattern fits more the SOA architecture than the typical coordination of distributed nodes for which the PeerSpace is designed. Nevertheless, in order to create a fair evaluation we need to evaluate such a pattern as well. The pattern consists of a client node which requests multiple responses from a server node. While the first response may be returned synchronously, further responses must be sent asynchronously back to the client. It is also allowed to send all responses asynchronously to the client. I.e. for a long running operation, the client wants to display a status bar, so multiple progress messages should be sent back before the actual response.

7.2 Implementation Notes

In this section we give some insight into the actual development process and discuss the implementation's complexity and the required development effort. We do not show the full source code of the patterns as all source code can be found as example projects within the PeerSpace framework Visual Studio solution.

The PeerSpace API allowed us to implement both patterns straightforwardly. The source code of the master/worker pattern has already been shown in the previous chapter. The request/multiple-responses pattern implementation just had to put additional co-data into the entries in order to connect the responses to a specific request.

The WCF framework on the other hand immediately showed, that the framework has been refined for the service oriented architecture paradigm. Although the learning process has been very fast and did not require a lot of effort, soon the limit of WCF's capabilities has been hit. To implement the master/worker pattern it has been necessary to make use of the rarely used duplex channel feature with so-called callback services. The request/multiple-responses pattern

	MI	CC	Coupling	LOC
Peer-Contract	96	11	0	11
Peer-Master	66	12	24	32
Peer-Worker	70	11	22	24
WCF-Contract	97	13	6	10
WCF-Master	66	21	31	72
WCF-Worker	66	12	20	34
MI . . . maintainability index (higher is better)				
CC . . . cyclomatic complexity (lower is better)				
Coupling . . . class coupling (lower is better)				
LOC . . . lines of code (lower is better)				

Table 7.1: Visual Studio code metrics for the master/worker pattern

	MI	CC	Coupling	LOC
Peer	82	22	16	38
WCF	87	22	23	44

Table 7.2: Visual Studio code metrics for the request/response pattern

should be favoring WCF, however, without the use of complex duplex channels WCF can only return a single response for a single service call. Altogether, WCF required us to do much more work manually and as soon as the default RPC-structure of service oriented client/server communication has been left, things became immediately complicated.

The Visual Studio code metrics shown in table 7.1 and 7.2 detected those complications (note, Visual Studio code metrics point out implementation complexity, they are not focusing on interface usability!). While the maintainability index (based on the Halstead volume [78]) is good for WCF and the PeerSpace (the value should be between 20 and 100), the WCF master node implementation looks worse in terms of cyclomatic complexity (measures the number of linearly independent paths in a given program [79]) and class coupling (describing how low the system's cohesion is, i.e. how interwoven the classes are [48]). Finally, the lines of code measurement clearly shows that you have to put in much more effort when implementing more complicated patterns (see master/worker results) with WCF.

7.3 Qualitative Evaluation

The qualitative evaluation is based upon the cognitive dimensions framework (see section 2.5.1, more precisely upon the tailored version for APIs by Steven Clarke [7]. The evaluation has been performed by the thesis author and has been reviewed by colleagues afterwards. Normally a number from 1 (fully applies) to 10 (does not apply at all) is assigned to each dimension. In order to retrieve more measurable results, we have realized a direct comparison between WCF and the PeerSpace and assigned either a "++" (PeerSpace is much better), "+" (PeerSpace is slightly better), "$\sim$" (no significant difference) "$-$" (WCF is slightly better) or "$--$" (WCF is

much better). In the end we can see if there are significant differences between both frameworks.

1. **Abstraction level**: The abstraction level in the PeerSpace is very consistently based on the underlying theoretical concept of the Peer Model. The API provides some higher level abstractions for usability reasons, however, access to the basis is always possible. In contrast, the WCF abstraction level is inconsistent: on one side it is quite high, because the framework hides all the underlying WS_-technologies, however, it does not do a very good job in hiding the required coordination effort when implementing cooperating distributed systems. In the implementation examples we had to configure too much details manually.
   Rating: $+$

2. **Learning style**: The initial required learning effort for the PeerSpace framework is manageable if you follow the provided examples and start with small coordination patterns. Bigger Peer Models are just "more of the same". WCF's learning curve is not steep either, a simple web service project is easily implemented. More complicated examples require to read the MSDN documentation more carefully.
   Rating: $\sim$

3. **Working framework**: The minimal required PeerSpace working framework is a single peer with no wirings. WCF requires a single service and a corresponding client. The service client may even be auto-generated. Both frameworks have a reasonable small working framework.
   Rating: $\sim$

4. **Work-step unit**: The work-step unit of the PeerSpace corresponds to adding a new wiring to the peer. This is comparable with WCF's work-step unit of adding a new service or service method. However, the integration of behaviors in WCF allows to realize cross-cutting concerns a little bit better.
   Rating: $-$

5. **Progressive evaluation**: As every peer is executable by its own and can run and be tested without any other peer, progressive evaluation is at its maximum. This is one of the strongest points of the PeerSpace API. The WCF code remains executable for most of the time. The corresponding other side (i.e. the server or the client) can be faked for testing purposes.
   Rating: $\sim$

6. **Premature commitment**: PeerSpace wirings can easily be adapted later on while keeping the data logic capsuled in services without requiring any changes. This is also a strong point of the PeerSpace API. While adding additional WCF services or behaviors is unproblematic, WCF has its downsides when implementing more complex patterns. E.g., the decision to make use of callbacks must be taken early on.
   Rating: $+$

7. **Penetrability**: Due to the fluent interface the PeerSpace API supports very well to be explored by the developer without consulting the documentation all the time. In contrast,

WCF does not come with much IntelliSense support. Due to many attributes and the use of XML configuration files it is required to read the documentation for retrieving additional information most of the time.
Rating: ++

8. **API elaboration**: The PeerSpace is able to directly transform Peer Models into running software components, an adaption of the API is not necessary. The WCF API must not be adapted very much either, however, minor problems arise when implementing advanced coordination concepts. E.g., the retrieval of the callback channel is a mechanism which a developer might want to abstract away.
   Rating: +

9. **API viscosity**: The PeerSpace is suitable for a very broad range of patterns and furthermore provides extensibility points to the user, e.g. replacing the container implementation. Due to the possibility to apply behaviors to various parts of the system, WCF also fulfills this dimension very well.
   Rating: $\sim$

10. **Consistency**: Although all terms are used consistently throughout the PeerSpace API, the different complexity of wirings forced us to give up some consistency in the wiring mechanism. I.e. simple auto-generated wirings vs. manually created advanced wirings. The WCF API itself is also very consistent to use, however, the XML configuration of services, especially in the areas of error handling or security are impenetrable in some cases. Both frameworks could still improve in this area.
    Rating: $\sim$

11. **Role expressiveness**: Peers, wirings, containers and entries are separated sharply from each other. The usage of the ubiquitous domain language from the Peer Model enabled good expressiveness. In contrast, when looking at the WCF master/worker implementation, the developer is forced to consult the documentation to see each component's specific purpose.
    Rating: +

12. **Domain correspondence**: The PeerSpace allows to implement coordination patterns straightforwardly. For the request/response pattern we still need a connecting ID in the entries. The only real domain covered by WCF on the other side, is the SOA service pattern. Other patterns are very hard to implement and sometimes require the developer to create complex workarounds.
    Rating: ++

The Visual Studio code metrics have already pointed out the direction: altogether the qualitative comparison results show that the PeerSpace is significantly better than the Windows Communication Framework (a total of 8 "+" for the PeerSpace vs. only 1 "−"). Of course, qualitative evaluations are always subjective up to a certain point, however, the difference in this case is large. While the WCF framework is suitable for SOA-style applications, it is no match for the

PeerSpace in the world of coordinated distributed systems. Other reports and articles have come to similar results, e.g. [26] or [25].

7.4 Quantitative Evaluation

The quantitative evaluation is based upon Thomas Scheller's API Usability Evaluation framework which has been presented in section 2.5.3. Although the Visual Studio code metrics in combination with our cognitive dimensions analysis already show a pattern, we want to get additional evidence that the PeerSpace performs better than WCF in terms of API usability when implementing coordination patterns. The tool outputs the number of different API concepts (e.g. class usage, method call, etc.) a developer must use, as well as how often they have to be used. The number of complex patterns (e.g. fluent interface usage, dependency injection usage, etc.) are also listed in the overview. There is of course also a more detailed output of the measurement tool, listing the scores of each used interface concept in detail, however, for brevity we've moved them to the appendix. The summarized results are as follows:

```
WCF Master/Worker results
=========================
Evaluation details: (41 concepts, 0 pattern complexity data)
Total -> 1265,0
Total number of independent actions (concepts*usages): 37

PeerSpace Master/Worker results
===============================
Evaluation details: (20 concepts, 1 pattern complexity data)
Total -> 768,2
Total number of independent actions (concepts*usages): 24
```

The total interface complexity value of the PeerSpace API is only sixty percent of WCF's value. Afterwards we even added error handling to the PeerSpace implementation (in case the worker is not reachable anymore, a retry is attempted, if the retry also fails, the work item is put back into the list of open work items), the evaluation still favored the PeerSpace. Implementing the same amount of error handling in WCF was not even feasible, if one side of the duplex link is lost, the connection has to be re-established manually and the lines of code would have exploded.

```
PeerSpace Master/Worker results including error handling
========================================================
Evaluation details: (27 concepts, 1 pattern complexity data)
Total -> 942,4
Total number of independent actions (concepts*usages): 32
```

The results of the second pattern are very much alike:

```
WCF Request/Multiple-Responses results
======================================
Evaluation details: (34 concepts, 0 pattern complexity data)
Total -> 1043,7
Total number of independent actions (concepts*usages): 30

PeerSpace Request/Multiple-Responses results
============================================
Evaluation details: (18 concepts, 0 pattern complexity data)
Total -> 651,5
Total number of independent actions (concepts*usages): 18
```

Even in the Request/Multiple-Responses scenario, where it was thought that WCF would shine with its SOA capabilities, we have received clearly better results for the PeerSpace.

7.5 Evaluation Results

Altogether we can see that the API usability of the PeerSpace is not only reaching the level of the Windows Communication Foundation but supersedes it in many use cases, especially for systems, that require complex coordination capabilities. The qualitative evaluation as well as the quantitative evaluation have come to equivalent results. Another report has also shown that the PeerSpace has good usability in comparison with various other frameworks [26]. The goal to create a good developer experience has been achieved.

Moreover, the API fulfills many important API design guidelines: the API's structure follows the wiring use cases, all API elements are consistent with regards to naming and structure and the API has been kept as small as possible. All class, method and attribute names match the ubiquitous domain language to prevent any developer confusion. Furthermore, the use of the factory pattern hides all implementation details. The fluent interface is more or less self-describing due to its sentence-like structure, it does not require any string parsing and included sanity checks made a fail-fast configuration system possible.

7.6 Feedback on the Quantitative API Usability Measurement Framework

Last but not least this section will give some feedback on the usage of the quantitative API usability evaluation framework of Thomas Scheller, which we have used in the previous sections to obtain an objective interface complexity score. This will hopefully support him to further improve his own work and complete his PhD thesis.

The method has especially proven helpful when comparing two or more completely different approaches targeting the same problem domain. The systematic breakdown of all required concepts and patterns, which are necessary to achieve a solution using a given approach, immediately points out developer pain-points. The framework identifies the score of each and every

concept and helps to find the problematic areas. In addition it allows to take a “previous usages” counter as input which allows us to determine how good the learnability factor of the API is.

Possible improvements are minor missing features, e.g. working with abstract classes is not possible yet: one has to remove the abstract keyword for the evaluation in order to assess the costs of calling a base constructor in a derived class. Furthermore, the measurement tool currently does not increase the interface complexity score if the developer needs to work with huge enumeration values. Last but not least, ellipsis arguments (i.e. `params object[]`) are currently very high rated, although if properly used, a very simple concept to understand.

Another possible high-level concept would be to assess all required thread synchronization. If a framework forces the developer to do synchronization herself/himself, the API is probably easier to understand, however, it is still harder to work with such a framework from the developer’s point of view.

Altogether the framework is already very stable and the detailed output allows to explore the results in a very comfortable way. This is important, so that also developers who are inexperienced in the area of API usability immediately understand what and why a certain construct is too complex for the users of the framework.

CHAPTER 8

Conclusion & Future Work

8.1 Conclusion

This thesis presented the very first Peer Model implementation based on the .NET framework and the Xcoordination Application Space as well as a proposal for a highly usable API. The resulting PeerSpace.NET framework is split into two parts: the PeerSpace core component and the PeerSpace API component, both achieving different goals of the initial problem description.

The motivation behind the PeerSpace core component was to provide a feature-complete Peer Model implementation in .NET. After collecting all functional and non-functional requirements in 4 we've succeeded in implementing the core framework. It is now possible to transform Peer Models into reusable components using either directly the PeerSpace core framework or the developer-friendly PeerSpace API. The implementation is feature-complete with the exception of transactions, and offers additional error handling capabilities. By following the SOLID principles [80] of object-oriented software design we've also managed to meet the non-functional goals of the core framework: it is well-structured and supports extensibility and maintainability. Especially the usage of the single responsibility principle and the dependency inversion principle have proven to be foundations of a good software architecture. We've managed to make all underlying implementation decisions, i.e. the usage of the Application Space or the specific container implementation interchangeable with very little effort for the developer.

The PeerSpace API component on the other hand had completely different goals. Based upon the core, its task was to provide an interface to the developer with a high API usability. The usage of a fluent API in combination with additional supportive attributes has resulted in very favorable results for the newly designed framework. Our evaluation used two methodologies: a qualitative and a quantitative method, both coming to the same results. A direct comparison with the WCF framework has shown that the space framework myth, that space based frameworks are hard-to-use, is not true. The PeerSpace.NET framework is better suited to implement complex distributed systems than the SOA-focused WCF framework, however, even in case of SOA-patterns like request/multiple-response the PeerSpace could keep up and even surpass the WCF framework's API usability.

Altogether the variety of patterns which can be implemented with the PeerSpace (see also [3]), emphasizes its strength: even complex coordination patterns can be implemented using the Peer Model and the PeerSpace.NET framework with reasonable effort and developers are able to implement them using clear and self-documenting code without the need to create any workarounds or hacks.

8.2 Future Work

The very first implementation of the Peer Model is feature-complete in regards to the original Peer Model paper [1] with the exception of transactions and has the desired amount of API usability. Nonetheless, during development we already found some interesting possible future work. Additionally it is required to add functional tests (i.e. unit tests and integration tests) and clear some important non-functional tests in order to make the PeerSpace enterprise-ready.

Automatic generation from Peer Model
: If Peer Model diagrams are saved in a well-known file format, probably based upon an XML standard, a generator could be created that automatically transforms Peer Models into PeerSpace code using the core framework.

Replacing the Application Space with a full-blown XVSM Space
: As soon as there is a full-blown XVSM Space for .NET available, providing additional container-features like transactions and replication, the current Application Space could be replaced.

Transaction safety
: With the current Application Space it is hard to implement transactions. One would have to implement a two-phase commit system at the peer-level which is not the inherent way in the Peer Model. Instead, as previously mentioned, it would be better to add this feature at container-level (e.g. by replacing the in-memory container with a full-blown space container).

Coordination Pattern Library
: Peers are re-usable and can be nested (sub-peers), therefore we could already implement a common library of well-known and often-needed patterns in form of a coordination peer framework. This approach could be extended into a workflow framework.

Multicast
: During the implementation of various patterns we found that it would be favorable to specify multiple destinations for a single action link (multicast).

Additional Hooks into the Wirings Runner Engine
: For even more extensibility we could add additional hooks into the wirings runner engine, e.g. Pre-PutIntoContainer, Before-Query, After-Query, etc.

Additional Request/Reply capabilities

Although not a main concern for the Peer Space, it would be nice to have built-in request/reply capabilities, i.e. a direct response should be clearly mappable to a preceding request. Currently additional IDs are required.

Long running wirings

If the developer marks a wiring as long-running, it should run asynchronously in an own thread. Basically very easy to implement, however, the wirings runner engine has to be reworked.

8.3 Closing Words

The Peer Model architecture has been successfully transformed into a modern development framework for distributed systems and has been enriched with a state-of-the-art API optimized for API usability. The PeerSpace has become a very good starting point for creating an enterprise-ready framework for re-usable, scalable and future-proof distributed software systems.

APPENDIX A

Detailed Evaluation Results

This appendix contains the detailed quantitative API usability evaluation results for the master/-worker as well as the request/response pattern. The presented output is the result of running Thomas Scheller's automated API usability measurement tool [8].

(due to the huge size, each output of the measurement tool is on its own page)

```
Evaluation details: (41 concepts, 0 pattern complexity data)
Cost S  Cost U  Cost Sum  Concept                 Usages  Prev Usages  Description
    30,0     29,0      59,0  ClassUsage                 1            0  ServiceContractAttribute
     0,0      4,0       4,0  Instantiation              1            0  [ServiceContractAttribute()]
    10,7      7,0      17,7  FieldAccess                1            0  SessionMode ServiceContractAttribute.SessionMode
    10,7      7,0      17,7  FieldAccess                1            0  Type ServiceContractAttribute.CallbackContract
    30,0     52,2      82,2  ClassUsage                 3            0  OperationContractAttribute
     0,0      7,2       7,2  Instantiation              3            0  [OperationContractAttribute()]
    10,9     12,6      23,5  FieldAccess                3            0  Boolean OperationContractAttribute.IsOneWay
    10,9      7,0      17,9  FieldAccess                1            0  Boolean OperationContractAttribute.IsInitiating
    30,0     40,6      70,6  ClassUsage                 2            0  DataContractAttribute
     0,0      5,6       5,6  Instantiation              2            0  [DataContractAttribute()]
    30,0     63,8      93,8  ClassUsage                 4            0  DataMemberAttribute
     0,0      8,8       8,8  Instantiation              4            0  [DataMemberAttribute()]
    30,0     21,0      51,0  ClassUsage                 1            0  ServiceHost
     1,0     19,5      20,5  Instantiation              1            0  new ServiceHost(Type serviceType,Uri[] baseAddresses)
                                                                          InformationLookup Pattern: 0 / 8 (lookup of external information (1 lookups))
    12,9     13,0      25,9  MethodCall                 1            0  ReadOnlyCollection<ServiceEndpoint> ServiceHost.AddDefaultEndpoints()
    13,9      5,0      18,9  MethodCall                 1            0  void ServiceHost.Open()
    13,9      5,0      18,9  MethodCall                 1            0  void ServiceHost.Close()
    12,9      5,0      17,9  MethodCall                 1            0  void ServiceHost.Abort()
    30,0     29,0      59,0  ClassUsage                 1            0  ServiceBehaviorAttribute
     0,0      4,0       4,0  Instantiation              1            0  [ServiceBehaviorAttribute()]
    11,9      7,0      18,9  FieldAccess                1            0  InstanceContextMode ServiceBehaviorAttribute.InstanceContextMode
    30,0     29,4      59,4  ClassUsage                 2            0  OperationContext
    10,0      9,8      19,8  FieldAccess                2            0  OperationContext OperationContext.Current
    11,8      7,0      18,8  FieldAccess                1            0  MessageProperties OperationContext.IncomingMessageProperties
    30,0     21,0      51,0  ClassUsage                 1            0  MessageProperties
    11,1      7,0      18,1  FieldAccess                1            0  Object Dictionary<String,Object>.Item (indexer property)
    30,0     29,4      59,4  ClassUsage                 2            0  RemoteEndpointMessageProperty
    10,1      7,0      17,1  FieldAccess                1            0  String RemoteEndpointMessageProperty.Address
    10,1      7,0      17,1  FieldAccess                1            0  Int32 RemoteEndpointMessageProperty.Port
    12,3     15,5      27,8  MethodCall                 1            0  String OperationContext.GetCallbackChannel<>()
    30,0     21,0      51,0  ClassUsage                 1            0  NetTcpBinding
     0,0      4,0       4,0  Instantiation              1            0  new NetTcpBinding()
    30,0     21,0      51,0  ClassUsage                 1            0  EndpointAddress
     0,0      6,5       6,5  Instantiation              1            0  new EndpointAddress(String uri)
    30,0     21,0      51,0  ClassUsage                 1            0  InstanceContext
     2,0     11,5      13,5  Instantiation              1            0  new InstanceContext(ServiceHostBase host,Object implementation)
    30,0     21,0      51,0  ClassUsage                 1            0  DuplexChannelFactory<String>
    13,0     27,0      40,0  Instantiation              1            0  new DuplexChannelFactory<String>(InstanceContext callbackInstance,Binding binding,String remoteAddress)
                                                                          InformationLookup Pattern: 0 / 8 (lookup of external information (1 lookups))
    17,3     13,0      30,3  MethodCall                 1            0  String DuplexChannelFactory<String>.CreateChannel()
    13,3      5,0      18,3  MethodCall                 1            0  void DuplexChannelFactory<String>.Close()
    12,3      5,0      17,3  MethodCall                 1            0  void DuplexChannelFactory<String>.Abort()
     0,0                0,0                             0               InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.

Total -> 1265,0
Total number of independent actions (concepts*usages): 37
```

Figure A.1: API usability results for the WCF master/worker implementation

```
Evaluation details: (20 concepts, 1 pattern complexity data)
Cost S  Cost U  Cost Sum  Concept          Usages  Prev Usages  Description
   30,0     52,2      82,2  ClassUsage          3            0  SerializableAttribute
    0,0      7,2       7,2  Instantiation       3            0  [SerializableAttribute()]
   30,0     37,8      67,8  ClassUsage          3            0  PeerAddress
    1,0     36,7      37,7  Instantiation       3            0  new PeerAddress(String name,Int32 port)
                                                                  InformationLookup Pattern: 0 / 16 (lookup of external information (1 lookups))
   30,0     29,4      59,4  ClassUsage          2            0  DefaultPeerFactory
   30,0     29,4      59,4  ClassUsage          2            0  ApplicationPeerBase<DefaultPeerFactory>
    0,0     24,5      24,5  Inheritance         2            0  new ApplicationPeerBase<DefaultPeerFactory>(PeerAddress address)
   10,1     10,5      20,6  MethodCall          2            0  void ApplicationPeerBase<DefaultPeerFactory>.Run(Boolean asSubPeer)
   10,1      7,5      17,6  MethodCall          1            0  void ApplicationPeerBase<DefaultPeerFactory>.Emit(Object[] objects)
   30,0     52,2      82,2  ClassUsage          3            0  ServiceAttribute
    0,0      7,2       7,2  Instantiation       3            0  [ServiceAttribute()]
   30,0     37,8      67,8  ClassUsage          3            0  IServiceContext
   11,0     12,5      23,5  MethodCall          1            0  void IServiceContext.EmitWithDestination(PeerAddress destination,Object[] objects)
   20,7     21,7      42,4  MethodCall          2            0  IFluentOutWireConfiguration IFluentPeerConfiguration.OutWire<>()
   10,0     23,9      33,9  MethodCall          2            0  IFluentFinishedOutWireConfiguration IFluentOutWireConfiguration.To(PeerAddress destination)
                                                                  FluentInterface Pattern: 0 / 2,2 (less relevance of return parameter, increased complexity of first usage)
   30,0     29,0      59,0  ClassUsage          1            0  StartupActionAttribute
    0,0      4,0       4,0  Instantiation       1            0  [StartupActionAttribute()]
   10,9     12,5      23,4  MethodCall          1            0  void IStartupActionContext.EmitTo(String internalDestination,Object[] objects)
   10,3      7,0      17,3  FieldAccess         1            0  String ServiceAttribute.EmitTo
   10,5     10,5      21,0  MethodCall          2            0  void IServiceContext.Emit(Object[] objects)
                       0,0                                     FluentInterface Pattern
    0,0     10,0      10,0  FluentInterface                  Complex Method chain: OutWire.To.OutWire.To (3 class changes, 1 usages, 0 prev)
                       0,0                                     InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
Total -> 768,2
Total number of independent actions (conceptsxusages): 24
```

Figure A.2: API usability results for the PeerSpace master/worker implementation

```
Evaluation details: (34 concepts, 0 pattern complexity data)
Cost %  Cost U  Cost Sum  Concept                 Usages   Prev Usages  Description
  30,0    29,0      59,0  ClassUsage                   1            0  ServiceContractAttribute
   0,0     4,0       4,0  Instantiation                1            0  [ServiceContractAttribute()]
  10,7     7,0      17,7  FieldAccess                  1            0  SessionMode ServiceContractAttribute.SessionMode
  10,7     7,0      17,7  FieldAccess                  1            0  Type ServiceContractAttribute.CallBackContract
  30,0    40,6      70,6  ClassUsage                   2            0  OperationContractAttribute
   0,0     5,6       5,6  Instantiation                2            0  [OperationContractAttribute()]
  10,9     9,8      20,7  FieldAccess                  2            0  Boolean OperationContractAttribute.IsOneWay
  10,9     7,0      17,9  FieldAccess                  1            0  Boolean OperationContractAttribute.IsInitiating
  30,0    40,6      70,6  ClassUsage                   2            0  DataContractAttribute
   0,0     5,6       5,6  Instantiation                2            0  [DataContractAttribute()]
  30,0    75,4     105,4  ClassUsage                   5            0  DataMemberAttribute
   0,0    10,4      10,4  Instantiation                5            0  [DataMemberAttribute()]
  30,0    29,0      59,0  ClassUsage                   1            0  ServiceBehaviorAttribute
   0,0     4,0       4,0  Instantiation                1            0  [ServiceBehaviorAttribute()]
  11,9     7,0      18,9  FieldAccess                  1            0  InstanceContextMode ServiceBehaviorAttribute.InstanceContextMode
  30,0    21,0      51,0  ClassUsage                   1            0  OperationContext
  30,0    21,0      51,0  ClassUsage                   1            0  OperationContext OperationContext.Current
  10,0     7,0      17,0  FieldAccess                  1            0  String OperationContext.GetCallbackChannel<>()
  12,3    15,5      27,8  MethodCall                   1            0  ServiceHost
  30,0    21,0      51,0  ClassUsage                   1            0  new ServiceHost(Type serviceType,Uri[] baseAddresses)
   1,0    19,5      20,5  Instantiation                1            0  InformationLookup Pattern: 0 / 8 (lookup of external information (1 lookups))
  13,9     5,0      18,9  MethodCall                   1            0  void ServiceHost.Open()
  13,9     5,0      18,9  MethodCall                   1            0  void ServiceHost.Close()
  12,9     5,0      17,9  MethodCall                   1            0  void ServiceHost.Abort()
  30,0    21,0      51,0  ClassUsage                   1            0  NetTcpBinding
   0,0     4,0       4,0  Instantiation                1            0  new NetTcpBinding()
  30,0    21,0      51,0  ClassUsage                   1            0  EndpointAddress
   0,0    14,5      14,5  Instantiation                1            0  new EndpointAddress(String uri)
                                                                       InformationLookup Pattern: 0 / 8 (lookup of external information (1 lookups))
  30,0    21,0      51,0  ClassUsage                   1            0  InstanceContext
   2,0    11,5      13,5  Instantiation                1            0  new InstanceContext(ServiceHostBase host,Object implementation)
  30,0    21,0      51,0  ClassUsage                   1            0  DuplexChannelFactory<String>
  13,0    19,0      32,0  Instantiation                1            0  new DuplexChannelFactory<String>(InstanceContext callbackInstance,Binding binding,String remoteAddress)
  17,3    13,0      30,3  MethodCall                   1            0  String DuplexChannelFactory<String>.CreateChannel()
  13,3     5,0      18,3  MethodCall                   1            0  void DuplexChannelFactory<String>.Close()
  12,3     5,0      17,3  MethodCall                   1            0  void DuplexChannelFactory<String>.Abort()
                     0,0                               1              InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.

Total -> 1043,7
Total number of independent actions (conceptsxusages): 30
```

Figure A.3: API usability results for the WCF request/response implementation

```
Evaluation details: (18 concepts, 0 pattern complexity data)
Cost $  Cost U  Cost Sum  Concept          Usages  Prev Usages  Description
  30,0    40,6      70,6  ClassUsage            2            0  SerializableAttribute
   0,0     5,6       5,6  Instantiation         2            0  [SerializableAttribute()]
  30,0    29,4      59,4  ClassUsage            2            0  DefaultPeerFactory
  30,0    29,4      59,4  ClassUsage            2            0  ApplicationPeerBase<DefaultPeerFactory>
   0,0    24,5      24,5  Inheritance           2            0  new ApplicationPeerBase<DefaultPeerFactory>(PeerAddress address)
  10,1    10,5      20,6  MethodCall            2            0  void ApplicationPeerBase<DefaultPeerFactory>.Run(Boolean asSubPeer)
  20,7    15,5      36,2  MethodCall            1            0  IFluentOutWireConfiguration IFluentPeerConfiguration.OutWire<>()
  10,0    18,5      28,5  MethodCall            1            0  IFluentFinishedOutWireConfiguration IFluentOutWireConfiguration.To(PeerAddress destination)
                                                                  FluentInterface Pattern: 0 / 3 (less relevance of return parameter, increased complexity of first usage)
  10,1     7,5      17,6  MethodCall            1            0  void ApplicationPeerBase<DefaultPeerFactory>.Emit(Object[] objects)
  30,0    40,6      70,6  ClassUsage            2            0  ServiceAttribute
   0,0     5,6       5,6  Instantiation         2            0  [ServiceAttribute()]
  30,0    29,4      59,4  ClassUsage            2            0  IServiceContext
  11,0    17,5      28,5  MethodCall            2            0  void IServiceContext.EmitWithDestination(PeerAddress destination,Object[] objects)
  30,0    21,0      51,0  ClassUsage            1            0  IEntry
  10,0     7,0      17,0  FieldAccess           1            0  Object IEntry<Object>.TypedData
  10,7     9,8      20,5  FieldAccess           2            0  PeerAddress IEntry.Origin
  30,0    29,4      59,4  ClassUsage            2            0  PeerAddress
   1,0    16,1      17,1  Instantiation         2            0  new PeerAddress(String name,Int32 port)
                      0,0                                      FluentInterface Pattern

Total -> 651,5
Total number of independent actions (concepts*usages): 18
```

Figure A.4: API usability results for the PeerSpace request/response implementation

Bibliography

[1] eva Kühn, Stefan Craß, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-Based Programming Model for Coordination Patterns. In *Coordination Models and Languages*, pages 121–135. Springer, 2013.

[2] eva Kühn. Distributed Programming with Space Based Computing Middleware, lecture at TU Vienna, lecture material, 2011.

[3] Gerald Schermann. Extending the Peer Model with Composable Design Patterns. Master's thesis, TU Vienna, E185/1, 2014. to appear.

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

[5] E Kühn, J Riemer, and G Joskowicz. XVSM (eXtensible Virtual Shared Memory) Architecture and Application. Technical report, TU-Vienna E185/1, 2005.

[6] Steven Clarke. Measuring API usability. *Doctor Dobbs Journal*, 29(5):S1–S5, 2004.

[7] Steven Clarke and Curtis Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In *Proceedings of the First Joint Conference of EASE PPIG (PPIG 15)*, 2003.

[8] Thomas Scheller and eva Kühn. A Framework for the Automated Measurement of API Usability. *submitted for publication*, 2014.

[9] Max Goff. *Network distributed computing: fitscapes and fallacies*. Prentice Hall Professional Technical Reference, 2003.

[10] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[11] Kurt Jensen. Coloured petri nets and the invariant-method. *Theoretical Computer Science*, 14(3):317 – 336, 1981.

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

[13] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert De Simone. The Time Model of Logical Clocks available in the OMG MARTE profile. In *Synthesis of Embedded Software*, pages 201–227. Springer, 2010.

[14] eva Kühn, Stefan Craß, Gerson Joskowicz, and Martin Novak. Flexible Modeling of Policy-Driven Upstream Notification Strategies. In *29th Symposium On Applied Computing (SAC)*, Gyeongju, Korea, March 24-28 2014. ACM. to appear.

[15] Dave Clarke. Coordination: Reo, nets, and logic. In *Formal Methods for Components and Objects*, pages 226–256. Springer, 2008.

[16] Wil MP van der Aalst. Formalization and verification of event-driven process chains. *Information and Software technology*, 41(10):639–650, 1999.

[17] Stefan Appel, Sebastian Frischbier, Tobias Freudenreich, and Alejandro Buchmann. Event stream processing units in business processes. In *Business Process Management*, pages 187–202. Springer, 2013.

[18] Carl Hewitt. Actor Model of Computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[19] Eric T Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns and Practices*. Addison-Wesley Professional, 1999.

[20] E. Kühn, R. Mordinyi, H.-D. Goiss, T. Moser, S. Bessler, and S. Tomic. Integration of Shareable Containers with Distributed Hash Tables for Storage of Structured and Dynamic Data. In *Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09. International Conference on*, pages 866–871, 2009.

[21] eva Kühn and Vesna Sesum-Cavic. A space-based generic pattern for self-initiative load balancing agents. In *Engineering Societies in the Agents World X*, pages 17–32. Springer, 2009.

[22] eva Kühn, Alexander Marek, Thomas Scheller, Vesna Sesum-Cavic, Michael Vögler, and Stefan Craß. A space-based generic pattern for self-initiative load clustering agents. In *Coordination Models and Languages*, pages 230–244. Springer, 2012.

[23] Michele Leroux Bustamante. *Learning WCF: A Hands-on Guide*. O'Reilly, 2008.

[24] Amit Bahree and Chris Peiris. Peer-to-Peer Programming with WCF and .NET Framework 3.5. `http://msdn.microsoft.com/en-us/library/cc297274.aspx#WCFP2P_topic1`, 2008. Accessed: 2014-04-17.

[25] Ralf Westphal. Unendliche Weiten - Komponentenorientiert verteilen mit dem Application Space. *dotnetpro*, (07):124–130, 2009.

[26] Gernot Rumpold. Analysis on API-usability for space-based computing frameworks. Technical report, TU Vienna, E185/1, Software Engineering & Internet Computing student's project, 2014.

[27] Ralf Westphal. WCF is not the solution but the problem. http://weblogs.asp.net/ralfw/archive/2010/02/20/wcf-is-not-the-solution-but-the-problem.aspx, 2010. Accessed: 2014-03-29.

[28] David Boike. *Learning NServiceBus*. Packt Publishing Ltd, 2013.

[29] Brad Abrams. The Pit of Success. http://blogs.msdn.com/b/brada/archive/2003/10/02/50420.aspx. Accessed: 2014-04-19.

[30] Vaughn Vernon. *Implementing Domain-Driven Design*. Pearson Education, 2013.

[31] Chris Patterson. MassTransit. http://masstransit-project.com/. Accessed: 2014-03-07.

[32] Alan Dickman and Peter Foreword By-Houston. *Designing applications with MSMQ: message queuing for developers*. Addison-Wesley Longman Publishing Co., Inc., 1998.

[33] Steve Vinoski. RPC under fire. *Internet Computing, IEEE*, 9(5):93–95, 2005.

[34] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F Ferguson. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[35] Jakob Nielsen. The usability engineering life cycle. *Computer*, 25(3):12–22, 1992.

[36] Samuel G McLellan, Alvin W Roesler, Joseph T Tempest, and Clay I Spinuzzi. Building more usable APIs. *IEEE software*, 15(3):78–86, 1998.

[37] Thomas RG Green. Cognitive dimensions of notations. *People and computers V*, pages 443–460, 1989.

[38] Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

[39] Elspeth Golden, Bonnie E John, and Len Bass. Quality vs. quantity: Comparing evaluation methods in a usability-focused software architecture modification task. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 141–150. IEEE, 2005.

[40] Linda Rochford. Generating and screening new products ideas. *Industrial Marketing Management*, 20(4):287–296, 1991.

[41] Yvonne Rogers, Helen Sharp, and Jenny Preece. *Interaction Design: beyond human-computer interaction*. John Wiley & Sons, 2011.

[42] Eric Evans. *Domain-Driven Design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[43] David A Caulton. Relaxing the homogeneity assumption in usability testing. *Behaviour & Information Technology*, 20(1):1–7, 2001.

[44] Robert A Virzi. Refining the test phase of usability evaluation: how many subjects is enough? *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 34(4):457–468, 1992.

[45] Jakob Nielsen. Heuristic evaluation. *Usability inspection methods*, 24:413, 1994.

[46] Morten Hertzum and Niels Ebbe Jacobsen. The evaluator effect: A chilling fact about usability evaluation methods. *International Journal of Human-Computer Interaction*, 13(4):421–443, 2001.

[47] David P Tegarden, Steven D Sheetz, and David E Monarchi. A software complexity model of object-oriented systems. *Decision Support Systems*, 13(3):241–262, 1995.

[48] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[49] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.

[50] Jeffrey Stylos and Brad A Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

[51] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, pages 529–539. IEEE Computer Society, 2007.

[52] Thomas Scheller and eva Kühn. Measurable concepts for the usability of software components. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 129–133. IEEE, 2011.

[53] Thomas Scheller and eva Kühn. Influencing factors on the usability of API classes and methods. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 232–241. IEEE, 2012.

[54] Andrew Troelsen. *Pro C# 5.0 and the. NET 4.5 Framework*. Apress, 2012.

[55] Jason King and Mark Easton. *Cross-platform. NET Development: Using Mono, Portable. NET, and Microsoft. NET*. Apress, 2004.

[56] Jeffrey Richter. *CLR via C#, 5th Edition*. Microsoft Press Redmond, 2012.

[57] TIOBE Software Quality Company. TIOBE Index for February 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, Feb 2014. Accessed: 2014-02-20.

[58] Xcoordination. Application Space. `http://xcoappspace.codeplex.com/`. Accessed: 2014-02-20.

[59] Shih-Chung Kang, Wei-Tze Chang, Kai-Yuan Gu, and Hung-Lin Chi. *Robot Development Using Microsoft Robotics Developer Studio*. CRC Press, 2011.

[60] Bill Gates. A robot in every home. *Scientific American*, 296(1):58–65, 2007.

[61] Maarten Balliauw and Xavier Decoster. *Pro NuGet*. Springer, 2012.

[62] Douglas Crockford. The application/json media type for javascript object notation (JSON), 2006.

[63] Douglas Crockford. *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[64] log4net. http://logging.apache.org/log4net/. Accessed: 2014-02-20.

[65] S Craß, T Dönz, G Joskowicz, E Kühn, and A Marek. Securing a space-based service architecture with coordination-driven access control. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 4(1):76–97, 2013.

[66] Stefan Craß, Jürgen Hirsch, eva Kühn, and Vesna Šešum-Čavić. An Adaptive and Flexible Replication Mechanism for Space Based Computing. In *8th International Conference of Software and Data Technology (ICSOFT-EA)*, Reykjavik, Iceland, July 29-31 2013. INSTICC Press.

[67] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[68] Michael Franklin and Stan Zdonik. Data in your Face: Push Technology in Perspective. In *ACM SIGMOD Record*, volume 27, pages 516–519. ACM, 1998.

[69] Łukasz Gąsior. *ReSharper Essentials*. Packt Publishing Ltd, 2014.

[70] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[71] Joshua Bloch. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507. ACM, 2006.

[72] Michi Henning. API design matters. *Queue*, 5(4):24–36, 2007.

[73] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008.

[74] Ken Arnold. Programmers are people, too. *Queue*, 3(5):54–59, 2005.

[75] Thomas Scheller and eva Kühn. Usability Evaluation of Configuration-Based API Design Concepts. In *Human Factors in Computing and Informatics*, pages 54–73. Springer, 2013.

[76] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[77] Juan J Perez and Sam Guckenheimer. *Software Engineering with Microsoft Visual Studio Team System*. Pearson Education, 2006.

[78] Kurt D Welker. The Software Maintainability Index Revisited. *CrossTalk*, pages 18–21, 2001.

[79] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[80] R.C. Martin. The Principles of OOD. `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`, 1995. Accessed: 2014-04-22.