DISSERTATION

Omnix: An Open Peer-to-Peer Middleware Framework

Engineering Topology- and Device-Independent Peer-to-Peer Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Leitung von

o.Univ.-Prof. Dipl.-Ing. Dr.techn. Mehdi Jazayeri
Institut für Informationssysteme
Abteilung für Verteilte Systeme (E184-1)

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Univ.-Ass. Dipl.-Ing. Roman Kurmanowytsch

`roman@infosys.tuwien.ac.at`

Matrikelnummer: 9327324
Brunnerstr. 28/12
A-1230 Wien, Österreich

Wien, im Februar 2004

[illegible]

Omnix: An Open Peer-to-Peer Middleware Framework

Engineering Topology- and Device-Independent Peer-to-Peer Systems

Ph.D. Thesis

at

Vienna University of Technology

submitted by

Dipl.-Ing. Roman Kurmanowytsch

Distributed Systems Group, Information Systems Institute,
Technical University of Vienna
Argentinierstr. 8/184-1
A-1040 Vienna, Austria

1st February 2004

Advisor:	o. Univ.-Prof. Dr. Mehdi Jazayeri
Second Advisor:	a.o. Univ.-Prof. Dr. Gabriele Kotsis

Abstract

In this thesis, we present *Omnix*, a Peer-to-Peer (P2P) middleware framework. P2P middleware provides an abstraction between the application and the underlying network by providing higher-level functionality such as distributed P2P searches and direct communication among peers.

P2P networks build an overlay network, a network on the network. The central idea of this dissertation is to apply the well-established ISO OSI layered model on P2P middleware systems. A layered P2P architecture helps to identify separate parts of a complex system, allows changes to the system without affecting other layers and makes maintenance easier due to modularization. The main advantages of having such a layered P2P middleware are: independence from the underlying network and the P2P topology.

This dissertation proposes a P2P middleware architecture consisting of three layers: the *Transport Layer*, the *Processing Layer* and the *P2P Network Layer*. The Transport Layer (which can be compared to the *Physical Layer* of the OSI model) is responsible for providing an abstraction to the network primitives provided by operating systems or libraries. This makes Omnix independent from the underlying network protocols and the devices and allows Omnix to run on heterogeneous devices. The Processing Layer (an analogy to the *Data Link* layer) provides an error free connection to remote peers and allows additional services to be plugged in. The P2P Network Layer comprises the routing algorithm, thus describes the topology (or structure) of the P2P network.

Omnix has a pluggable architecture that allows different components to be plugged in based on the requirements of an application, the capabilities of the device the application is running on, the number of peers involved, etc. Custom-tailored P2P middleware systems can be constructed, thus, increasing the flexibility of communication architectures that are built upon it. Applications that use Omnix are provided an API that is not affected whenever the underlying topology has to be changed or adapted.

Since testing P2P networks is hard to accomplish due to the large amount of peers required to get significant results, Omnix provides a way of testing P2P applications using this middleware framework. By replacing the communication layer by a virtual network, it is possible to simulate a network of virtual peers on a single computer without the necessity of changing the application on top of it.

Along with Omnix, several plugins are provided to demonstrate the versatility of the framework. They provide the functionality to run Omnix on normal PCs, on handheld computers (e.g., Compaq iPAQ) and on Java-enabled mobile phones.

Kurzfassung

Diese Dissertation stellt Omnix vor, eine Peer-to-Peer (P2P) Middleware. P2P Middleware Systeme bieten eine Schnittstelle zwischen der P2P Kommunikation-Infrastruktur und der darüber liegenden Applikation an. Dienste wie zum Beispiel das Suchen in einem P2P Netzwerk oder die direkte Kommunikation zwischen Peers werden dabei angeboten.

P2P Netzwerke bilden ein virtuelles Netzwerk auf einem existierenden Netzwerk. In dieser Dissertation wird das anerkannte OSI (Open Systems Interconnection) Netzwerk-Schichtenmodell der Internationalen Organisation für Standards (ISO) auf P2P Netzwerke angewendet. Das Schichtenmodell hat den Vorteil, dass komplexe Systeme in kleinere Teile aufgeteilt werden können, dass Schichten verändert werden können, ohne einen Einfluß auf die Schichten darunter zu haben und dass die Wartbarkeit und Erweiterbarkeit solcher System erhöht wird. In einem P2P Netzwerk bedeutet das hauptsächlich eine Unabhängigkeit von der zugrundeliegenden Netzwerk–Technologie und der Struktur des P2P Netzwerkes selbst.

In dieser Dissertation werden drei Schichte vorgeschlagen: die *Transport*-Schicht, die *Processing*-Schicht und die *P2P Netzwerk*-Schicht. Die Transport-Schicht kann mit der *physikalischen Schicht* des ISO Modells verglichen werden. Sie ist dafür zuständig, den Zugang zum Netzwerk transparent, und daher auch unabhängig vom eingesetzten Netzwerkprotokoll, zu machen. Die Processing-Schicht (welche an die OSI *Sicherungsschicht* angelehnt ist) erzeugt einen fehlerfreien Verbindungskanal und bietet darüberhinaus noch weitere, wichtige Dienste an. Die P2P-Netzwerk-Schicht beinhaltet die Routing-Algorithmen und beschreibt daher die Struktur des P2P Netzwerkes.

Omnix verwendet eine komponenten-orientierte Architektur die es erlaubt, das System den Erfordernissen anzupassen. Die dadurch gewonnene Flexibilität erlaubt es, Omnix für viele verschiedene Zwecke auf unterschiedlichsten Geräten einzusetzen, ohne jedoch dabei die Schnittstelle zur Applikation zu verändern.

Für das Testen von Applikation, die auf Omnix aufbauen, kann die Netzwerkkommunikation durch ein virtuelles Netzwerk ersetzt werden, dass es erlaubt, mehrere Tausend Peers auf einem einzelnen Computer zu simulieren. Dabei ist es nicht notwenig, die Applikation selbst zu ändern.

Um die breite Einsetzbarkeit von Omnix zu demonstrieren, werden verschiedene Module angeboten, die es erlauben, Omnix auf einem PC, einem Handheld (z.B. Compaq iPAQ) oder einem Mobiltelefon mit Java.

Acknowledgments

Many people directly or indirectly contributed to this dissertation.

I am greatly indebted to my wife Hemma for her patience and mental support during the stressful time of writing this dissertation. She was always willing to let me work until late in the evening while she was taking care of our one year old daughter. Her enduring support and understanding made this dissertation possible. I am also grateful to my parents for their continuous support over the long years of my education.

I would also like to thank my advisor Prof. Dr. Mehdi Jazayeri who introduced me to research and provided an environment where I was able to look into many research areas and work with interesting people. His feedback has been invaluable in structuring and writing this dissertation. I further offer my sincerest gratitude to Prof. Dr. Gabriele Kotsis, who displayed a great deal of flexibility and kindness.

Doing research at the Distributed Systems Group of the Information Systems Institute at the Vienna University of Technology was an incredible experience. I would like to acknowledge the work and dedication of the people working there.

Special thanks go to my colleague and friend Clemens Kerer, who was always willing to listen to my ideas and point me to new directions. I would like to express my appreciation for all the efforts he made in numerous joint projects which has provided me with a great deal of enjoyment over many years. Thanks must also be given to Engin Kirda, who made working at the Distributed Systems Group at any time a lot of fun. He always provided encouraging words when they were most needed.

I thank Michael Loibl, Johannes Schmidt and Reinhard Steiner for their input into this thesis by working on various sub-projects in the course of their master theses.

This dissertation is dedicated to my children Lea and Noah.

Roman Kurmanowytsch
Vienna, Austria, February 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One might believe P2P systems are mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion.
Hari Balakrishnan et al. [1]

With the ongoing advancements in the computing area, the number of different computing devices dramatically increased. While there were, back in the old days, only a couple of big mainframe machines around the world[1], nowadays millions of PCs and laptops are sold every year. Processing power has also found its way into many more devices, among them handhelds, mobile phones, embedded devices and sensors, thus creating a vast, heterogeneous environment of different devices connected together.

With the increasing computing power in every device, the traditional client-server architecture, where clients with less capabilities connect to a central server, no longer suffices. The sheer number of these devices, their heterogeneity and the increasing connectivity among them calls for another communication paradigm where the communication is brought to the edges of the network, away from the servers (i.e., so-called *hot spots*) that can no longer cope with the increasing requirements. The Peer-to-Peer (P2P) approach is taking this direction. The main advantages of P2P are, among others, increased robustness, fault-tolerance, and scalability.

In many application domains, P2P can be used as the underlying communication platform. Section 2.2.2 lists some of these use cases, shows how P2P is used to enhance them and gives examples of existing systems. There already exist some P2P middleware systems that support developers using P2P communication in their applications. Examples of such P2P middleware systems are JXTA [2], Pastry [3], Groove [4], or BASE [5].

Most of these systems fail to provide full versatility in respect to the application domain where they can be used. This is due to the fact that they are not able to employ arbitrary P2P network topologies. The topology of a P2P network influences many aspects of a P2P network. It (among other things) 1) defines how peers are connected, 2) how messages are routed in the network, 3) influences the scalability and stability of a system, and 4) possibly sets limits to the search functionality. Hence, for a P2P middleware, it is crucial to be able to adapt the P2P topology to the needs of the application on top of it [6].

[1]"The number of Unix installations has grown to 10, with more expected.", The Unix Programmer's Manual, 2nd Edition, June 1972

What is also missing in most existing P2P middleware systems is the support for heterogeneous devices. The reason for this shortcoming is that they do not provide a clear separation between those parts that are responsible for accessing the network primitives provided by the operating system or the virtual machine and the remainder of the middleware system. Therefore, it is impossible to use the systems on other networks or operating systems without adapting the middleware system as a whole.

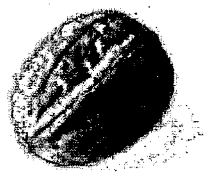
What we see is a lack of standardization in the P2P area. This is analog to the problems experienced in distributed systems in the mid-1970's. At that time, various protocol suites like the system network architecture (SNA) by IBM or the protocols designed by the ARPANET existed, with no interoperability between them. To mitigate this problem, in 1977 the British Standards Institute put forth the idea for a common standard for communication infrastructures. As a consequence, the International Organization for Standardization (ISO) created the 7-layer OSI model[2] and made it standard in 1979. This standard, which helped to provide interoperability between different protocols and made the protocols independent from the underlying devices, is clearly missing in the P2P area.

The main advantages of using a layered model such as the ISO OSI model are:

- Layering helps to identify and understand separate pieces of a complex system.
- The communication between different systems do not require changes to the underlying hardware or software.
- Maintenance and updating is easier due to modularization.

The disadvantage of layering is the complex data exchange between the adjacent layers.

1.1 Problem description in a nutshell



Peer-to-Peer middleware systems provide services by creating an overlay network on existing networks. Existing systems do not employ a layered structure (e.g., as the ISO OSI model) for this network or do not fully cover the lower layers. Hence, they do not allow to replace transparently 1) the underlying network or device, 2) the protocols used, or 3) the topology (i.e., routing) of the overlay network.

[2]The American National Standards Institute (ANSI) proposed this model, which has been originally designed by a group at Honeywell Information Systems in 1977 under the name Distributed Systems Architecture (DSA).

1.2 Omnix

To provide a common ground for existing and new P2P applications, we propose an open P2P middleware system design that, like the OSI model, helps to cover a wide range of protocols and devices and does not impose any restriction on the application on top of it. This is, by all means, not an easy task as it may sound because the right balance between technological beauty and usability is crucial for a middleware system (as for any other system). What does it help to have a technologically perfect design solution, when it is not feasible that it will ever be implemented?[3] For example, it must not be the case that the system gets so complex that it can no longer run on less capable devices such as a handheld computer. For these reasons, we do not intend to apply the ISO OSI model as a whole on P2P but to identify a system design that provides best flexibility while still being useful.

In this dissertation, we take the first step and propose a P2P middleware system design–*Omnix*–that provides a separation of and clear interfaces between the networking primitives provided by the device (or operating system, virtual machine, communication libraries, etc.), the routing algorithm and the application.

Omnix provides a consistent API to the application layer that has been crafted to cover most of the functionality of existing and conceived P2P systems. Chapter 8 discusses to which P2P topologies the API can be applied and where shortcomings exist. In addition, various existing components (e.g., delivery guarantee mechanisms or security) can be easily extended and support the creation of P2P services without the exigence of building the P2P system from scratch.

1.3 The Program Committee Scenario

This section shows an example usage scenario for the Omnix P2P middleware platform. It shows how Omnix can be used by applications that require a communication platform. Furthermore, it exemplifies how Omnix helps to reduce the time an application programmer needs to change from one P2P topology to another (and why this would be necessary) and how to enable P2P applications to run on different devices, always using the same code.

1.3.1 The Setting

A program committee of an international conference is meeting at the Vienna University of Technology. The purpose of the meeting is to discuss submitted papers and to agree on which of these papers to accept. Before this meeting, the submitted papers have been distributed among the committee members for reviewing. Each paper has been assigned to 2–3 people, taking into account their area of interest and research. Depending on the number of submissions, each reviewer got 10–20 papers for reviewing.

The meeting starts in the morning. Every committee members brings her assigned papers (as PDF files) and the corresponding reviews to the meeting. After they have told the

[3]As an example: in the IP network, most protocols (e.g., SMTP, HTTP) do not have an interface between the Session layer and the Presentation layer, simply because it would increase complexity.

secretary how they like their coffee, the committee members start the meeting by looking at the first submitted paper scheduled for this morning session.

One of the assigned reviewers of the paper gives a short overview of the paper. She talks about the problem definition, the proposed solution and the soundness of the paper. The other reviewers of this paper add some comments where appropriate. After this short introduction, the program committee decides whether the paper should be accepted or not. They follow the suggestion of the three reviewers and accept the paper.

This process is repeated with the other papers scheduled for this morning.

1.3.2 Computer Support for the Review Process

There are a lot of ways how computers and a network of computers can help to streamline the review process and to support the committee members in their decision to accept or to reject a paper.

In this scenario, I want to concentrate on the papers assigned to the reviewers, the corresponding reviews and the decision making process of whether a paper gets accepted or not.

Since each paper is only assigned to 2 or 3 reviewers, the other committee members see the paper for the first time at this meeting. Before they decide in favor or against the paper, they might want to have a look at it or want to have a glance at the reviews of the three reviewers. Without computers, each paper must be copied as many times as committee members are involved. This is also true for the 2–3 reviews available for each of the submitted papers. This would not only create tremendous work for the person responsible for making the copies but would also be a waste of paper.

It would be easier to collect all papers and their reviews electronically on a central server. Hence, before the meeting starts, the reviewers upload their reviews to the central server. The participants of the meeting then can easily access the paper currently discussed. Furthermore, the person could also access the reviews for the submission in question and have a look at it.

A computer program could also help the process of deciding in favor of or against a submission. If all committee members have a program where they can vote for or against the paper, the results could be collected and interpreted at a server.

1.3.3 Using Omnix in the Meeting

One the program committee members is Alice[4] When she arrives at the meeting in the morning, she downloads the software onto her laptop. The software is implemented in Java and uses the Omnix middleware (not that she is quite interested in this fact).

When she first runs the software, she is asked for her name and the password (which has been issued when she arrived at the meeting). The software checks whether the username and password are correct by connecting to the server, which has been installed for this meeting only. After that, she has to specify the location of the PDF files she was supposed to review and also to provide the reviews. The software automatically uploads the PDF files

[4]She seems to be very popular as she appears in many example scenarios.

and the reviews to the central server. From this point on, the other committee members can access the papers and the reviews.

What was the role of Omnix so far in this scenario?

The software that Alice started in the morning is an application that uses the Omnix P2P middleware. Omnix is configured to use a topology module that requires a central server. At the time when Alice provides a username and a password, the application uses Omnix to request the server to authenticate the given credentials. After that, the application asks Alice to upload the assigned PDF files and the appendant reviews. Again, the application uses the topology module of Omnix to send the files to the central server.

The process of deciding whether a submitted paper gets accepted or not is also supported by the application. Alice presses a button to indicate a "aye" or "nay". The software sends this decision to the server which holds this information for the applications of the other participants. The chairperson can collect this information and present the outcome of the voting.

1.3.4 A Second Meeting

Unfortunately, the meeting took longer than anticipated. Only two thirds of all papers have been processed. To make things worse, a social event (visiting a "Heurigen") is scheduled in the afternoon. The committee members decide to proceed with the meeting at the restaurant. For this purpose, they put all files and information on their (network-enabled) iPAQs and Palm computers.

It is clear, that they will not have access to the server installed at the University. Hence, they have to cope without a central server and switch to a another form of group collaboration. It is not feasible to change the software in this short amount of time. Therefore, they decide to stick to the same software.

Omnix allows this because it does not require a central server. By using another topology module, the do no longer need a central server but communicate directly with each other. The local technician provides a new topology module for download at the web server.

While, at the University, the reviewers upload the PDFs and the corresponding reviews to the server, they keep these files on their local machines and share them with the other participants at the restaurant. Searching for a PDF file or a review is done the same way. The difference is that the topology module does not send the search request to the server but sends it to the other devices in the vicinity. Instead of downloading the files from a server, they download the PDFs and reviews directly from the reviewers' devices.

The voting is done the same way. Instead of sending the verdict to a single address, the decision whether a paper gets accepted or not is sent to all participants, including the chairperson who gets the results from all participants and is then able to make a conclusion.

But there is an problem. Alice uses a Palm handheld computer which has a Java version that is not fully compatible with the Java 2 Standard Edition (i.e., it uses network access libraries that differ from the Standard Edition). Hence, before the group leaves the University, she quickly downloads a plugin for the Omnix middleware that allows to use the Java network primitives provided by the Palm handheld device. The decoupling from the application and the topology in use makes this simple replacement of the underlying transport layer feasible.

As a result, the committee meeting can proceed at the Heurigen without any problems.

1.3.5 Alternatives

It would be, of course, possible to use another P2P middleware. The most widely used middleware is JXTA. It also provides P2P primitives like sending messages and to search for meta-information in the P2P network.

The application described above would be different in the way that the two scenarios described above would have to be implemented separately by the application programmer. Since the application is only supposed to be used at the University where a central server is available, it is possible (or maybe even likely) that the application programmer did not consider to cover the aspect that the central server is not available. It is more likely that the programmer provides a backup server in case the server crashes.

Furthermore, JXTA does not run on small devices (with some limited exceptions; more on JXTA in Section 8.3.3). This may not be a major restriction, but it means that the software using JXTA cannot run on anything like a normal PDA or a mobile phone.

1.4 Contributions

The aim of this thesis is to create an open P2P middleware framework that can be used in conjunction with various devices, network protocols and P2P topologies. Existing P2P middleware systems are either very heavy-weight and complex or not open for other networks or P2P topologies. Our objective is to create a layered P2P middleware system that represents a compromise between flexibility and complexity.

Reference for application developers

To show the need for an open P2P middleware, we perform a study of classes of P2P topologies. With the help of several evaluation criteria, we explore how the topology influences several aspects of a P2P system. The conclusion from this evaluation is that a P2P middleware, if it should be usable for arbitrary applications and use cases, devices and networks, must not use a fixed P2P topology. This study is also a work of reference for application programmers who have to choose the best fitting P2P topology for their purpose.

Identifying the requirements and a design for an open P2P system

Based on this evaluation, we identify five fundamental requirements for an open P2P middleware system: platform independence, small footprint, topology-independence, openness, and support for testing the application by simulating a P2P network. To the best of our knowledge, there exists no P2P middleware system that fully covers all these requirements. For this reason, we devise an architecture of a P2P middleware system (*Omnix*) that meets all of the aforementioned requirements by using layers. The layered structure of Omnix allows to replace single layers without affecting the rest of the system. The Omnix API allows applications using this middleware to access the services of the lower layers independently of the topology in use.

Designing an adequate open P2P protocol

Furthermore, we identify important requirements for an open P2P protocol (among them: low network usage, topology independence, reliability, transport protocol independence, and support for multiple hops) and present a protocol that fulfills these requirements. The Omnix reference protocol uses a light structure that requires very low costs for parsing, processing and generating of messages.

Covering a wide range of devices

This thesis further contributes a Java implementation of the P2P middleware architecture, which is able to run on heterogeneous devices (e.g., mobile phones, handhelds, or PCs), thus enabling many pervasive computing scenarios that are not possible with existing P2P middleware systems. Omnix is the first initial step in allowing the creation of flexible P2P applications that are platform and topology-independent.

1.5 Stakeholders

There are three roles that might have a strong interest in the Omnix P2P middleware system.

- *Application programmers.* The Omnix P2P middleware system is targeted at being used by application programmers who need a P2P infrastructure for the communication between program instances. Its versatility makes it possible to use Omnix in various contexts (e.g., groupware applications, file sharing, meta-data search, gaming, etc.).
- *System contributors.* Since Omnix provides an abstraction between the middleware framework and the topology of the P2P network, it is possible for third parties to provide so-called topology plugins. If, for example, a P2P topology can be applied to more than one use case, it may given (or sold) to other application programmers.
- *System administrators.* Once a P2P system using Omnix is deployed, a system administrator is responsible for maintaining the network. Various tasks like the replacement of topology plugins (if necessary) or the setup of proxy peers (see Section 6.3) are then part of the system administrator's task.

1.6 Structure of this Thesis

The remainder of this dissertation is structured as follows.

Chapter 2 presents the concepts and technologies used in this dissertation. It includes an introduction to P2P networks, defines the terminology and points out where P2P systems may be used as a communication platform.

Chapter 3 covers the related work in respect to existing P2P networks. It discusses the various topologies of P2P networks currently available and conceived. In this chapter, evaluation criteria are defined and applied on these topologies.

Chapter 4 describes the architecture of Omnix. It first details the advantages of this architecture over other approaches considered in the design phase of Omnix. Furthermore, it delves into advanced issues like firewalls and NAT, proxying, changing topologies, bridging between topologies, etc.

Chapter 5 defines the requirements for a protocol and introduces the reference protocol for the Omnix middleware framework.

Chapter 6 focuses on a reference implementation of Omnix. It emphasizes the versatility of the Omnix architecture by showing implementations for PCs, handheld computers and mobile phones. It details how a certain level of security is achieved and presents ways of testing and running applications using the Omnix P2P framework.

Chapter 7 gives an overview of applications using Omnix as the underlying P2P communication infrastructure. These applications include a file sharing application, a generic service engine and an experimental study providing some performance measures.

Chapter 8 evaluates the Omnix architecture in respect to other P2P middleware systems and the lessons learned from developing P2P applications on top of Omnix.

Chapter 9 concludes this dissertations by summarizing the essential contributions of this thesis. In addition, potential directions for further research are discussed.

Chapter 2

Concepts & Technologies

I can see peer-to-peer computing being made the flavor of the day by the wireless community and working its way back into mainstream data processing.
Simon Phipps, Chief Technology Evangelist at Sun Microsystems, Inc. [7]

2.1 Various definitions of "Peer-to-Peer"

Especially when it comes to P2P different opinions about what this term comprises exist. The most common definition of this "buzzword" reads, "A network of nodes that can act as server and client at the same time". Well, most people working in this field would agree that this is statement is not wrong. The problem is, that this definition is not complete. It leads to questions like, "Is a network relying on a central server like Napster P2P?".

Clay Shirky [8] defines P2P as a way of accessing decentralized resources at the edges of the Internet with unreliable connectivity and dynamic IP addresses. Further, P2P must be independent of the DNS (Domain Name System [9]) and must have no significant dependence on central servers. The first part of this definition resembles the initial definition above. The reason why a P2P network should be ready for unreliable connectivity and changing IP addresses is obvious: many P2P users are using modem connections what leads to very slow and brief connections. Furthermore, whenever a user makes a dial-in connection he/she gets probably another IP address. This means that the P2P system should not expect a node (or peer) to be available at the same address at all time. Dial-in connections also sometimes do not have a DNS name and can therefore only be addressed by their IP address. This is the reason why Peer-to-Peer systems should not rely on the DNS. The part of the definition above that sometimes gets overlooked is the independence of a central server. Napster, the best example for a system that got the P2P name tag, relies on a central server.

One might argue whether P2P must not make use of a central service. It would help reliability and scalability (among other crucial requirements for distributed systems) a lot but the reason for Napster's success was not its technological beauty, but the fact that it worked. Users did not complain about Napster using a central server. On the contrary, it is very hard, still not impossible, to deliver the same service Napster offered by using a serverless solution. A more detailed comparison of such P2P systems is given in section 3.1.

The peer-to-peer working group [10] defines P2P as follows: P2P is the sharing of computer resources and services by direct exchange between systems. Peers can act as servers and clients, assuming whatever role is most efficient for the network. This part of the definition makes it almost impossible to match it with the definition of Shirky [8]. That some peers act as server and some as client is nothing special and is reality in today's P2P systems. The problem is the requirement that the peer assumes the role that allows the network to be most efficient. It is questionable whether this is possible without a central server coordinating the peers and the communication flow between these peers. Anyway, the definition does not prohibit the usage of central servers. Nevertheless is this definition particularly interesting because it does not only include the direct exchange between systems (as any other P2P definition) but also includes the goal to make the use of network bandwidth most efficient. This aspect might become important when P2P systems are used in a closed domain like within a company network (referred to as Enterprise P2P).

By the definition of webopedia.com [11], P2P is a type of network in which each workstation has equivalent capabilities and responsibilities. The first part of the definition is not always true. P2P systems have to take into account that some peers have less capabilities than others (e.g., bandwidth, processing power, storage space, etc.). If the first part only refers to the software, it is also not fully correct. Peers do not necessarily have to be equal. There are P2P systems imaginable where different types of devices have different P2P software running, tailored to the capabilities of the device (e.g., a mobile phone maybe does not provide services, but only accesses services of the network). The second part is also misleading. While in simple P2P networks like Gnutella all peers have the same responsibility (i.e., forwarding and processing search requests), in newer P2P systems, depending on the capabilities, selected peers take more responsibilities than others. They provide, for example, information about other peers to improve search performance or scalability. Examples for such systems are some distributed hashtables and the FastTrack network.

Another definition, which is used in this document, reads as follows, "Peer-to-Peer is the exchange of information between systems that can act as clients and servers. These peers should be able to work without a central server but should also be ready to make use of such a server if it is necessary to fulfill its tasks, adds to the efficiency of the system or is more convenient for the user. A P2P system should not rely on static IP addresses and reliable network connectivity." It is also possible to reverse the definition and state that a server-based peer-to-peer network should also be able to function even when the central server is not reachable. The aim of this definition is to stress that a server-less P2P system have some disadvantages and should therefore not refuse acceptance of a server just for the sake of it.

2.2 Peer-to-Peer: An old new idea

2.2.1 Evolution of computing

During the history of computers several evolutionary steps occurred. Back in the (not so) old days, computers were once thought of as big, monolithic machines. Limited to small numbers, these mainframe computers were housed in big rooms and operated by skilled technicians.

The growing number of computers, their reduction in size, and the increasing connectivity among them made place for the client-server paradigm. Servers provided information and processing power to multiple client computers. Different approaches of how much work is shifted from the server to the client exist. The simplicity of this model made it very popular and is one of the reasons why the client-server paradigm is still widely used on the Internet.

The next trend was the so-called three-tier architecture. In this model, the work is partitioned into three parts: 1) the presentation, which is responsible for displaying information 2) the processing of the data, and 3) back-end services such as databases. Each of these parts could run on different machines, providing increased flexibility and scalability.

The increasing processing power of modern computers, the huge number of computers connected to the Internet, and the fact that computing capability has moved into almost every device imaginable gave rise to the next big evolution in networked computing, the Peer-to-Peer (P2P) paradigm.

In P2P computing, the client/server paradigm is replaced by a model where every computer is both a server and a client. This has the effect that information is no longer concentrated on central servers (so-called "hot spots") but provided by each computer (i.e. "peer"), utilizing the full bandwidth of the Internet. The idea behind P2P is to bring the communication to the edges of the network because using centralized hot spots are no longer scalable due to the increasing size of information provided these days. Furthermore, the usage of the currently underutilized processing power of the computers connected to the Internet is also made possible with P2P model (a remarkable example is Seti@HOME [12]). Main advantages of P2P are the the robustness of the network, the fault tolerance and the combined performance.

The exchanging of files over the Internet has shown the vast potential of P2P systems. But there is more to P2P than just sharing music and video files over the Internet.

2.2.2 Some application domains of P2P applications

More and more companies and scientific researchers are now searching for new ways of applying P2P technology on known problems. For most of these problems, traditional client-server solutions exist and are in use.

There are two reasons for what seems to be like reinventing the wheel again. The first one is the fact that some problems can be solved better by using P2P. Compared to the client-server paradigm, P2P offers more robustness, fault-tolerance, scalability, processing power, bandwidth, etc. Hence, in certain use cases, P2P is superior to the client-server model.

The second reason is the hype around P2P file sharing systems. In this "gold rush", people are now trying to apply the same technology on everything else, whether it makes sense or not [1].

There are many application domains where P2P systems are in use:

Instant messaging A convenient way of communicating with a small group of selected people (e.g., friends, family members, etc.). Usually, a central server is used to store user profiles and to have a list of registered users. While communication takes place between the peers, searching for other people is done using the server. One of the reasons

[1] Or, worse, some systems are named P2P although they are not

why a server is needed is the ability to send messages to other persons (i.e., peers). If the target peer is not online, the system has to store the message until the target peers becomes online again. This would be, of course, also possible with a server-less P2P system, but the price would be an increased complexity and a certain probability of messages getting lost. Examples for such systems are Napster [13], ICQ [14], threedegrees [15], and Jabber [16].

File exchange There is little dispute about the usefulness of P2P file sharing applications. While downloading files is always done directly between peers (or via a proxy peer to enable anonymity), the way of searching for these files differs in many P2P applications. Some use central servers (e.g., Napster) while others send search requests directly to other peers (e.g., Gnutella [17], Freenet [18], and FastTrack [19]).

Collaboration Collaboration is not a typical example for the usefulness of P2P technology. It is about having people having the same view or different views on shared information. This would typically call for a server storing this information. This way, the information is available to all members without the necessity of having the information provider or contributor online or the data distributed to all other participants. But there are use cases where P2P technology comes in handy. One example could be ad hoc collaboration of devices in an environment where no connection to a server exist (e.g., people are meeting in a place where no connection to the Internet is available). In this scenario, people would communicate (and collaborate) in a server-less P2P manner. One perfect example for this use case is Groove [20]. It uses a server to store shared information but is also able to provide collaboration services without the existence of such a server.

MIPS sharing One of the major assets of the Internet is its combined processing power, which is currently vastly under-utilized. To utilize these resources, user are asked to download and install programs that are able to do a small part of a complex computation while the computer is not used (e.g., while the screen saver is running). Examples for MIPS sharing systems are Seti@HOME [12] and Genome@HOME [21]. In this category of P2P applications, the social aspect is very important. Were it not for the search for extraterrestrial life or cancer research, not many people would be willing to share their processing power (hence, there must an incentive for users to share computer resources, be it money, public well-fare or the like). Furthermore, this type of P2P application can only function with a central server that is coordinating the distribution of computation tasks and the validation of the results.

Lookup services Most of the scientific P2P research is done in the area of lookup services. This is not very surprising because searching is one of the major challenges in P2P networks. Most of the P2P systems that are optimized for lookup services are using distributed hashtables (DHT), which are capable of searching with logarithmic complexity. The drawback of most of these systems is the fact that they are only able to search for numbers (in case they are searching for strings, they are searching for numerical representations of these strings). Examples for such systems are PAST [22], Chord [23], and P-Grid [24].

Mobile ad hoc communication Ad hoc communication, especially when it is done among mobile devices (i.e., the devices are connected directly via a wireless communication link), is the best example for the usefulness of the P2P paradigm. Devices connect to each other in an ad hoc manner. Due to the limited communication capabilities of mobile devices (such as mobile phones or handheld devices), frequent disconnections may occur. When mobile devices are connected together, there is no guarantee that a central server may be available. Hence, ad hoc mobile communication must not rely on the existence of such a server. All these characteristics also apply to the P2P paradigm. There exists only a small number of P2P systems that can be used in conjunction with small devices, among them GnuNet [25] and JXME [26] (JXTA for J2ME - the Java 2 Mobile Environment).

Content Distribution P2P can also be used for the distribution of information or files (sometimes called ESD - electronic software distribution). Instead of having a central source that emits files to the destination computers directly, a P2P network may disseminate files while avoiding hot spots in the network. The load (i.e., the bandwidth, CPU power, throughput, etc.) is distributed over the whole network. This concept is successfully used, for example, by Intel where software is distributed to international branches in a P2P style. This system can be compared to a push system. The advantage is that there is no need for a fixed environment of push server and proxies.

Middleware The most demanding use case for a P2P system is its use as a middleware platform. P2P middleware systems provide services such as distributed search or peer discovery to higher-level applications. As will be discussed in Chapter 3, different kind of P2P systems have a limited set of application domains. Depending on the structure (or topology) of the P2P network, various use cases may become feasible or impossible. If a P2P system is needed as a middleware, the use case turns the balance which P2P system best fits the requirements. Only a few P2P systems may be used as a P2P middleware platform. Among those are JXTA [27] and Omnix.

For a comprehensive list of P2P applications, see [28].

Schoder et al. [29] identifies the following challenges in the P2P area (among others): 1) Network control (the size and structure of a P2P network can not be predicted exactly), 2) Security, especially if the whole Internet community is allowed to access the information and services a peer is providing, 3) Interoperability (how to connect two or more proprietary P2P networks) and 4) Metadata (how to describe the information and services shared in the network and how to search these descriptions). In addition, several other challenges exist such as the bypassing of firewalls and NAT configurations.

Although the P2P paradigm will have a great potential in the future, in many cases it is more advisable to apply the client-server paradigm for the sake of performance, security, or simplicity.

2.2.3 Historical P2P systems

The idea of having independent computers connected with each other without the use of a central organizational unit is not new. One of the best known examples is the Usenet, which dates back to 1979. Two students from the Duke University of North Carolina, Tom Truscott and Jim Ellis, had the idea of connecting two computers to exchange information with the Unix community. The result is the today's Usenet network. It is a collection of independent computers exchanging chat messages without any central authority.

In 1982, the Corvus Omninet was developed. It had been used to share the expensive harddisks with other computers. There was no central authority to accomplish this.

Another example is the FidoNet (created by Tom Jennings in 1984), which – like Usenet – is used for exchanging messages in a decentralized way. The reduction of modem (or telephone) time was the idea behind this system. In 1992, 30,000 nodes in the Fidonet were approximated.

One of the biggest example of P2P networks is the IP network itself. In 1962, the U.S. Air Force asked the RAND Corporation to analyze how it could keep control over its forces after a nuclear attack. The result was a completely decentralized, fault-tolerant and robust network of nodes, the so-called ARPANET, the predecessor of the Internet.

All these systems experienced the same problems as the P2P systems developed in the recent years (i.e., scalability, security, fault-tolerance, etc.). Hence, it makes sense to look at these "old" systems and learn from them.

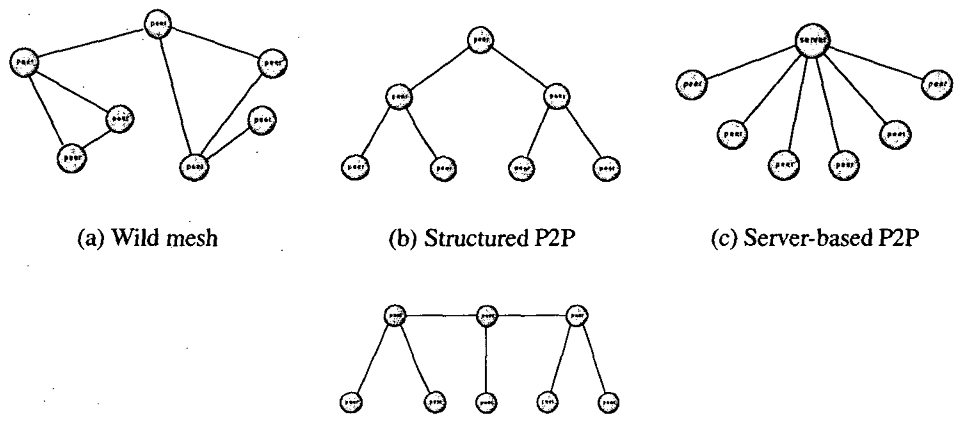
2.2.4 The context has changed

Experienced computer scientists smile when they look at the hype around P2P. For them, the idea is not all that new. Having computers connected directly without a central administrative authority is not new and has been implemented in various computer networks, among them the IP network.

But there is one major difference that changes a lot: *connectivity*.

To illustrate this, a few examples are in order. The Usenet, for example, is often labeled as a P2P network, but it is a network of servers, set up and maintained by a (relatively) small number of system administrators. The millions of people using the Usenet are clients that connect to one of these servers. Without these servers, nobody would be able to exchange messages in the Usenet (because it would not scale). The Internet's mail system is also a kind of a P2P network. Mail transfer agents (MTA) can connect directly to each other for exchanging mails. Again, in this case the peers are professionally maintained servers for many people who connect to these mail servers. The communication pattern between the people and the MTAs is client/server. The IP network itself is also considered as a classical P2P network. But even in this case, some differences exist. The peers (i.e., the endpoints of the IP network) are using the services of fixed, central servers (such as routers, gateways, etc.) for the communication. They do not support the communication of other peers nor do they have any effect on the structure of the network. Furthermore, if a new endpoint is added to the network, this is usually done manually (e.g., by changing the routing tables; although this does not apply to dial-in clients of ISPs, of course).

All these examples do not fully apply to the idea of the "modern" server-less P2P paradigm, but there are a lot of similarities. In the server-less P2P paradigm, only peers exist, without any supportive infrastructure (with one exception: the network, of course) nor central services (e.g., DNS). There is no other authority that decides how peers are connected



(a) Wild mesh (b) Structured P2P (c) Server-based P2P

(d) Hybrid P2P

Figure 2.1: P2P topology classes

or how communication should flow between peers. The network is defined and maintained by the peers themselves. Millions (maybe soon billions) of peers are connected directly. The problems like naming, discovery, security, etc., that have been solved before for the client/server model now have to be considered again and existing solutions re-evaluated.

2.3 Topologies

P2P systems can be differentiated by their use of a central server, but there exists no clear separating line. Different hybrid models exist that make it impossible to categorize them only into server-less vs. server-oriented P2P networks. This sections provides an overview over these topologies and shows some general advantages and drawbacks that all P2P systems have in common. The term topology refers only to the structure of the overlaying P2P network, not the IP network beneath. It comprises peers and the connections between these peers, which may be directed and have different weights: it can be compared to a graph with nodes and vertices connecting these nodes. Defining how these nodes are connected affects many properties of P2P networks, as the following sections will show. A more detailed discussion of the properties of P2P networks is given in section 3.1. Note that the terms "Pure P2P" and "Hybrid P2P" are not used in the same way as in other literature (e.g. Yang et al. [30]). A more fine-grained definition is used to show the differences between the respective P2P systems more clearly.

2.3.1 Pure P2P

A pure P2P system is a network of nodes that do not depend on a central server. Peers are directly interconnected and send search messages to their neighbors. In some P2P networks each peer receiving a search request searches its local database, returns the result and in parallel forwards the search request to one, some or all of its neighbors (depending on the protocol). Since there is no central organizational unit, the peers have to maintain the topology and connectivity by themselves.

The advantage of pure P2P systems is the fact that there is no single point of failure (i.e. central server) which results in an enhanced survivability of the P2P network. One disadvantage of a fully distributed P2P network with no central authority is the disability of updating the system thoroughly. It is possible to build the protocol and the software in a way that allows central-server-driven updates but there is currently no P2P system in use that exhibits this ability. The most important problem of pure P2P systems is the bootstrapping process. When the P2P software is first started on a machine, at least one peer of the overall system must be known in order to be able to connect to the network. In most cases, this is usually done by providing a fixed, well-known server that serves a list of some of the P2P peers. Another possibility would be to employ connections to random IP addresses in certain address ranges that provide acceptable probabilities. But this solution is regarded as very unfriendly and therefore only a last resort. A P2P system that contacts a central server only once for obtaining a list of other peers can still be regarded as a pure P2P system because further interaction with the server is not necessary.

In pure P2P networks, two schools of thoughts exist. The first one (of which Gnutella is an example) is very minimalistic but provides high flexibility. The other optimizes performance while, in turn, the search functionality is limited. Here, both variants will be taken for evaluation.

"Wild Mesh"

The wild mesh-style P2P model (Figure 2.1a) is very simple. Each node holds an arbitrary amount of connections to other nodes. Incoming search requests are processed locally and, in parallel, sent to the connected nodes until the TTL (time-to-live) is reached. The connections are chosen on a random basis. A node does not have a specific place in the network where it belongs to.

The topology is flat, there exists no hierarchy (if connection preferencing is not considered). Evaluating such P2P models is very hard due to its indeterministic connection states. Still, there exist some evaluations of the Gnutella (a perfect representative for the wild mesh P2P) network (among others [31], [32], [33], [34]). One output is that the network tends to build clusters. Clusters have a high degree of connectivity within, but only a few connections to other clusters.

Some of the information presented here about wild mesh P2P networks is based on literature about Gnutella, since Gnutella is a wild mesh P2P network par excellence and only little information on wild mesh P2P networks in general is available.

Structured P2P

At the other end of the spectrum of pure P2P networks are those systems that do routing based on the content that is provided (e.g. [22], [23], [35], [36], [37], [38]). Gunther [39] gives an overview on how this topology could be related to Gnutella. Typically, peers work together like a big distributed hashtable (Figure 2.1b). A function is used to compute a number based on the content (or the description of the content). This hash number is then used as an identification number in a distributed data structure. In some systems, searches for a specific number begin at the root of the structure and narrow down the number with every step down the hierarchy. At the end, either the peer that holds the information has been found or at least the peer that holds the information on where to find the actual content. The topology is not necessarily restricted to a tree. Hypercubes or other constructs (like multi-hypercubes [40]) might also be used to organize the search space by numbers. However, in the following only a tree will be used for simplicity reasons.

Whenever a peer joins the network, it may become part of the tree. In some P2P networks, the content itself is shifted to the peer whose ID matches best the ID of the content. In other systems, only meta-data about the shared information is transferred. Expanding the structure is usually triggered by exceeding the size of the routing table, the workload on peers, or other criteria.

2.3.2 Server-based P2P

As the name indicates, server-based P2P systems (Figure 2.1c) rely completely on a central server or on a set of well-known servers (Figure 2.1c). There are different ways how a central server might be used. The server can provide search and indexing facilities (e.g. Napster [13]), authenticate users and encryption key distribution (e.g. Grokster [41]) or coordinate the work of the peers (e.g. Seti@HOME [12]).

There are several advantages when using a central server. The protocol can be easily changed and improved without having the problem of convincing all clients to support this protocol (although this is not always regarded as an advantage because a protocol should always be designed in a way that it is flexible enough). Further, the usage of a central server helps security. Authentication and authorization mechanisms can be employed because all significant traffic (e.g. search requests, login, etc.) is going through a single organizational unit, the server. This may be not important when it comes to distributed file sharing but security becomes definitely an issue when the P2P system is to be deployed in a corporate environment. Along with security billing is also much more easier to manage in a P2P system that can rely on a central server.

On the other side, there are drawbacks with central-server-based P2P networks as well. Obviously, the central server represents a single point of failure. This is not only a technical problem but also a legal one. As Napster has shown, using a central server is not a problem when it comes to efficiency and stability (as stated in Milojicic et al. [42]). It is more a legal issue that brought the fall of Napster. But this is not a general problem of P2P systems - as long as they are used to share legal content. Still, denial-of-service attacks are a potentially big threat to central servers (as for any other central server like www.cnn.com).

According to the definition given in Section 2.1, P2P systems may take advantage of a central server but should also be prepared to function without such a service. An example for this functionality is Groove [20]. Groove instances may connect to a central server for storing information permanently. In the absence of a central server, computers running Groove can still interact and provide most of the functionality.

2.3.3 Hybrid P2P

A hybrid P2P system (Figure 2.1d) uses also servers for indexing and searching. The difference to the server-based approach described above is, that these servers are chosen from the pool of all peers currently online. The concept of using a hybrid P2P was first widely used in the FastTrack [19] network. Certain criteria like bandwidth, latency and CPU power are used to elect the appropriate "server" (in FastTrack they are referred to as "SuperNodes"). A normal peer connects to a SuperNode and sends a list of all shared files. Search requests are sent to the SuperNode only, which, in turn, forwards the message to other SuperNodes. Every SuperNode receiving a query searches the local database of shared files (those made publicly available by the "client" peers).

The advantage of this system is that it combines the advantages of Napster (fast searches) with those of Gnutella (no single point of failure, survivability). This makes it a strong P2P model, a fact that has also been proven by the wide acceptance of P2P file sharing tools using the hybrid approach (e.g. Kazaa [43] and Grokster [41]). There is also an effort using so-called search hubs in the Gnutella network (see [44] and [45] for more details).

The term "hybrid P2P" is not clearly defined. Yang et al. [30] regard server-based P2P as a hybrid P2P system as opposed to a pure P2P system.

In a hybrid P2P system, peers complying to some criteria become special peers (in FastTrack: SuperNodes). Hence, the definition of P2P that every peer acts as a server and a client no longer holds. To accommodate this fact, this model is called a "hybrid P2P" system in contrast to a "server-based" model. In this document, this definition for "hybrid P2P" will be used.

Note that a set of centralized P2P servers could also be interconnected. This is not regarded as a hybrid P2P network since the servers themselves are still fixed and must be known to the peers.

2.4 Myths about P2P

P2P did have a difficult start because people were unsure what P2P exactly is. When Napster was introduced, it was called the first P2P application. But later, when Gnutella came out, Napster was no longer considered P2P because, after all, it used a server. As a consequence, people came up with ideas on how to differentiate between P2P systems and those that are not. Today, there exist so many definitions of what P2P is that it is even harder to tell whether a distributed system is P2P or not.

This led to the creation of several myths about P2P. The following list deals with the most common misconceptions about P2P and tries to solve some of these usual misunderstandings.

- *P2P does not allow any central coordination.*

  The advantage of P2P is that it does not have a central authority that is a bottleneck for system performance and a single point of failure. The idea behind P2P is to distribute the workload, the bandwidth usage, the content, etc. over the network instead of putting this altogether on a single machine (or on a very expensive server farm).

But there are two things to be considered before banning any kind of server from the P2P world. First, there are P2P topologies (and actual systems) that do take advantage of a server. In Napster, for example, the server was used to collect the meta-data of all shared files and to provide a search facility. Napster, however, was a P2P system because downloads were performed directly between peers. Only the searching had been provided by the server (which, obviously, reduced the burden of the server significantly).

The second, and even more important reason for a central coordination in a P2P network is the *bootstrapping*, a problem that every P2P system is confronted with. If you are starting a peer for the first time, it does not know a single address of another peer in the network. Hence, it first has to connect to a central server (or, at least, to a set of well-known peers, which is not so much a difference) to get a list of available peers. Once the peer is connected, it does not require the server anymore (when it is restarted, a peer may use a list of peers recorded in previous sessions). Hence, the server is vital to interconnect all peers.

- *A distributed system is only a P2P system, if it 1) supports temporary network addresses and 2) allows the peers significant autonomy.*

  As a general rule, this statement is too restrictive. If a P2P system requires permanent network addresses, it may be used in closed areas where this prerequisite can be fulfilled (e.g., in companies, universities, etc.). This "rule of thumb" (or "litmus test") has been created with public file sharing P2P systems in mind. For this use case, the above stated rule is perfectly true, but it does not apply to other scenarios where it is not a necessary assumption that network addresses are not permanent.

  The degree of significant autonomy (however *significant* can be interpreted) is also not a good indication whether a distributed system is a P2P system or not. There exist so many different nuances of peer autonomy that it cannot be taken as a sign for a P2P system.

- *All peers in a P2P system are of the same type.*

  In most cases, this is true. A peer typically is implemented in the same way as any other peer. This is true for, as an example, Gnutella where all peers have the same responsibilities. It is also true for hybrid P2P systems where peers may have different responsibilities, such as FastTrack, because every peer should be ready to take over any role in the network.

  But the point is that it is not a general requirement for P2P systems to have all peers equal in a P2P system. Depending on the topology, it might be the case that peers are different (e.g., if two different topologies are connected by a bridge). It would be too restrictive to define a P2P system only as a system of equals only because popular file sharing systems do it this way. With the development of new topologies that provide even more scalability and reliability, it is definitely feasible that peers are no longer equal but depend on the device they are running on, the bandwidth of the Internet connection, the processing power, and much more.

- *P2P does not scale.*

In the recent years, some theoretical studies about the scalability of P2P have been published, either in favor of the P2P approach (e.g., Schollmeier et al. [31]), or against it (e.g., Ritter [32]). It is very hard to assess the scalability of a given P2P system. But it is completely impossible to tell whether P2P itself is scalable. Because there are so many topologies with different ways of increasing scalability, it cannot be foreseen how scalable P2P really will be. In the past 5 years, new P2P systems with new topologies have been introduced, each increasing scalability significantly (e.g., a distributed hashtable has only logarithmic – depending on the number of hosts in the network – complexity for searching).

What definitely can be said is that P2P is more scalable than the client/server architecture. Due to the fact that the load of distributed searches, downloads, maintenance messages, etc. is taken off from a single server and distributed over all peers in the network, no bottleneck that reduces the scalability of the network exists.[2]

The next chapter delves into the evaluation of P2P topologies. It discusses the various classes of P2P topologies and evaluates them by choosing relevant criteria.

[2]As an example: FastTrack, the most popular P2P file sharing application, has constantly more than 2.5 million users online.

Chapter 3

Peer-to-Peer Topologies

Architects of P2P systems claim thousands, millions, even billions will interact in a seething, transient pattern of communication. Can we really achieve guarantees in the chaotic P2P environment?
John Kubiatowicz [46]

A myriad of Peer-to-Peer (P2P) systems exist, all having some advantages and drawbacks. P2P is also no longer just used for sharing music files over the Internet. An increasing number of P2P systems are used in corporate networks or for public welfare (e.g. providing processing power to fight cancer). System developers are more and more using the P2P paradigm for the underlying communication architecture of their applications (e.g., in the pervasive computing area).

Every P2P system has its own set of characteristics which make them more or less useful for the intended purpose. These characteristics are mainly influenced by the underlying topology of the P2P system. Before a system developer chooses a particular P2P system for communication, she has to evaluate which P2P system best meets the requirements of the application. If an ill-fit P2P system is taken, various problems (e.g., scalability, functionality, security, etc.) may be the result.

Due to the ongoing advancements in the P2P area, this chapter compares P2P topologies, but not actual P2P system implementations (as, for example, in [47] or [42]). For this purpose, a classification for P2P topologies is needed and introduced in this chapter. This survey gives an overview of the advantages and challenges different P2P topology classes have. The aim of this evaluation is to provide a work of reference to application programmers to choose the right P2P network topology. Depending on the requirements the application has to meet, the system designer could use this evaluation to find the best fitting type of P2P system for her application.

Defining a model for a P2P network

For the purpose of this evaluation, we use a simplified model of a P2P network: it is a collection of nodes connected over a network. There cannot be any assumptions about the capabilities of the peers (i.e., processing power, the type of connection, the available bandwidth, etc.) be made. Peers may connect to and leave the network anytime. Each node provides some content or services that other peers can access. For each file or service shared, the peer has to provide a description (or *meta-data*) that can be searched by other nodes. Due to the absence of a global index of this data, the main responsibility of the P2P network (i.e., all peers combined) is to provide this search functionality. Depending on how the nodes are connected, different ways of searching are feasible. In general, a peer searching for information sends a so-called *search request* to one or several other nodes. These nodes then process this search request by either running a search in their own set of meta-data descriptions or by forwarding the message to other nodes in the network. Message can take multiple hops (i.e., they are forwarded multiple times) until they reach their goal. Once the content is located at a remote peer, it is typically accessed directly.

3.1 Evaluation

Comparing topologies of P2P systems is very complex because the topology of a P2P network influences nearly all aspects of the system. Surveys in the P2P area (like [42] or [48]) use various characteristics like *Robustness, Efficiency, Scalability,* etc. to evaluate existing P2P applications. [6] uses these evaluation criteria and some more to provide a broad evaluation of P2P systems.

In the following, those criteria that have the highest impact on the usability of a P2P system have been chosen to evaluate classes of P2P systems.

Ad hoc mobile communication

The topology of a P2P network has an high impact on its ability to handle mobile ad hoc communication. Mobile ad hoc communication means that P2P devices connected through a wireless communication facility (e.g. WLAN or Bluetooth) communicate directly without the need for a link to the Internet. Ad hoc mobile communication will gain more importance in the future. It is expected that small, network-enabled devices (e.g. sensors) will create a P2P network enabling pervasive and ubiquitous computing [49].

Wild mesh: The wild mesh topology has many advantages when it comes to ad hoc mobile communication. Its decentralization makes it a good candidate for this type of communication. There is no need for a central server and the communication may start immediately without setting up the topology. On the other side, due to the lack of routing tables, a lot of unnecessary traffic is created which might incur high battery and bandwidth consumption, which is a problem for mobile devices.

Structured P2P: In this context, devices are typically connected in a micro or pico cell. Content-based P2P networks are more designed to work efficiently with huge amounts of data in a global setting. The overhead of at first building the tree structure before the actual communication can start could be too costly, considering that small devices with only a limited timeframe for communication are involved. The frequency of disconnections and reconnections of peers is another problem when using a structured P2P network in a MANET (mobile ad hoc network). Mobile peers would be too volatile to be relied on in a tree indexing meta-data or holding actual content. Taking into account that mobile peers do not necessarily have high bandwidth capabilities and CPU power, the usage of these peers in a tree might not be a good idea.

Server-based P2P: In the case that mobile devices are directly connected via a wireless network, no central server might be available. Hence, it is not possible to use this topology for ad hoc mobile computing, unless one the involved devices decides (or gets elected) to run the server. In this case, communication will be concentrated on this single device. Hence, the overall communication load would be taken away from the less capable devices.

Hybrid P2P: Hybrid P2P networks have some advantages over other topologies. One SuperNode must be elected before information exchange could start. Although it introduces more complexity (compared to wild mesh P2P networks) to the initiation of the communication, the advantage of having servers may be a good trade-off. The fittest devices involved in the communication could become SuperPeers helping less capable devices to minimize the workload.

Connectivity overhead

The fact that peers are very volatile (i.e. they may suddenly disconnect and reconnect with another IP address) forces most P2P systems to be able to handle such characteristics. Since it is likely that peers are only online for a very short period (see [50]), the P2P system has to strive to reduce costs (in terms of bandwidth and CPU power necessary) for closing the hole opened by the missing peer and for reintegrating the peer when it is back online. A general rule is that the more structured the P2P network is, the more effort is necessary to integrate new peers and compensate disconnected peers.

Wild mesh: In a wild mesh network, there is virtually no connectivity overhead. Assuming that a peer has a list of other peers, it connects to another peer in the list only if an existing connection breaks down. This is possible because the network does not have to maintain a specific topology. There is no need for finding special neighbors or building up routing tables. When a new peer enters the network, it may connect to any peer (if there are no limitations like connection preferencing).

Structured P2P: Compared to the other classes of P2P networks, content-based routed P2P networks pay the highest penalty for leaving or joining nodes. When a peer joins the network, it has to find a place in the tree (or whatever structure is used), all routing tables in the proximity have to be updated, and data has to be transferred to the new node. In networks where not only meta-data but the content itself is stored, a lot of information may be transferred when a new node arrives. When a node leaves the network, again the routing tables of all predecessors, children or siblings have to be updated. Content that was stored by the peer has to be replicated again.

Server-based P2P: The overhead of connecting to a server depends on the actual implementation. It is necessary that the server receives information about all the files and services a peer is sharing. If a peer reconnects to a server, it may be the case that the server still has this information in its database. Then, only changes have to be sent to the server. If the server does not cache this information, the complete meta-data has to be transferred to the server. Depending on the frequency of connections or disconnection, this traffic might impose a problem.

Hybrid P2P: As in a server-based topology, clients have to upload information to a "server." The difference is that in this topology, servers (or SuperNodes) cannot be relied upon. When a SuperNode ceases to exist, peers connected to such a SuperNode have to connect to another SuperNode and transmit all meta-data information again. This could be a rather costly process, depending on the size of the meta-information available. The result is that the connectivity overhead in hybrid P2P network is comparatively high. For normal peers it is very cheap. They only have to check a single connection, the one to the SuperPeer.

Fault resilience

This criteria deals with the question: What happens, if a peer disconnects without any warning? Does this affect the other peers? Is there a loss of information? In general, P2P networks have to take into account that peers are not reliable at all. The structure of a P2P network (i.e. its topology) has a high impact on its fault resilience.

Wild mesh: In wild mesh P2P system, the loss of connection to a peer is not so critical since the system is designed not to rely on specific peers or central servers. Pure P2P networks show a very high resilience compared to other topologies. According to Cohen et al. [51], power-law networks with an exponent of at most 3 are very rugged when it comes to random peer breakdowns. Gummadi et al. [52] claim that with a fraction of up to

$$1 + (1 - m^{\tau-2} N^{3-\tau} \frac{\tau - 2}{3 - \tau}) \tag{3.1}$$

lost nodes the network still survives, which means that the network does not split into two disjunctive parts. Given that the minimum node degree m is 1 and the maximum node degree N is 20, up to 59.8% of all nodes may breakdown before the network is divided (if τ is 2.3, as stated by Gummadi et al.). In [53], the power-law exponent is 2.07 while Jovanovic et al. [33] found -1.4 as the value for τ.

Structured P2P: There exists almost no literature on fault resilience (in respect to quantitative evaluations) of structured P2P systems. Wang et al. [54] describe how to enforce resilience to failure in structured P2P networks. However, it is still hard to compare the resilience of one structured P2P network with that of another structured P2P network due to the lack of a common evaluation process. Usually, when a peer disconnects, other peers are trying to fill the gap. This way, the communication overhead is considerable but this way, information has a higher availability. A typical value for the number of messages needed to replace a disconnected peer is $\mathcal{O}(\log^2 n)$.

Server-based P2P: It is not hard to imagine that server-based P2P networks are very fault resilient (as long as the server is available, of course). If a client disconnects from the network, it does not have any impact on all the other clients. It only affects those peers that are using services provided by this peer. It might be even the case, that information that has been uploaded to the server is still available when the owner is no longer available (it does not have always be the case that only meta-data instead of real data is uploaded to a server). This has the advantage that meta information about files or services could be acquired for future reference when the owner is back online. The only risk in a server-based P2P network is the server. If the server (or the cluster of servers) is no longer available, the complete system is–obviously–rendered useless.

Hybrid P2P: The fault resilience of hybrid P2P networks depends on certain implementation issues. Generally, the loss of a normal peer has only little effect on the network. As in server-based P2P network, only the server (or SuperNode) has to handle this situation. More severe are the implications when a SuperNode ceases to exist. If this is the case, the combined meta-data information of all connected peers is instantly not available. This information is not lost, but all connected peers have to connect to another SuperNode and upload this information again.

Message count

One important metric when comparing P2P systems is the average amount of messages a search request implies and how many hops these messages take. This metric is more important for pure and hybrid P2P systems because the amount of messages and their hop count is highly predictable in a server-based P2P system: it is always 1 (as long as the central server is not a cluster of multiple servers). The other type of message that has to be considered are those messages that are needed to maintain the topology of the P2P network. This includes simple Is-Alive messages (like *PING* messages in Gnutella) or messages needed to build and maintain the routing tables. This type of message is not so important to the user as is the case with search requests but the complexity of the P2P network has an high impact on the bandwidth consumption of *structural* (or *maintenance*) messages. In the literature, the number of messages used to search for information is the primary key for comparing P2P networks. Sometimes, Landau's notation (or also known as the $\mathcal{O}(.)$ notation) is used to describe the costs for searching in a P2P network. This might not always be useful. It is true that this notation describes the upper bound but, as an example, the possibility to omit constants (e.g. $\mathcal{O}(1E10 * \log n)$ equals to $\mathcal{O}(\log n)$) makes it very imprecise for P2P networks where not millions of peers are involved.

Wild mesh: Due to its very low complexity, wild mesh systems might use search messages to keep up the structure (hence, they combine these two tasks). Gnutella gives an example on how to reduce the cost for maintaining the system. Peers are found by evaluating forwarded result messages.
The basic idea of searching in wild mesh topologies is to send search requests to a set of neighbors, which process the search request and forward it to their neighbors. This process stops when a certain hop count (the so-called TTL - time to live) has been reached. Typically, search results return on the same route as the search request. Hence, all peers that forward a request see the response and are able to update their caches and list of known peers according to the data in the returned message. From the searching peer's point of view, responses are received sporadically. In most cases, the client shows the results that have been received so

far while still waiting for further response messages. This method of searching has the effect that a single search request creates an avalanche of subsequently forwarded messages. The total number of messages sent over the network in one direction is (theoretically)

$$2n\sum_{i=0}^{t-1}(n-1)^i \tag{3.2}$$

where n is the average number of live connections a peer has and t the typical TTL (time-to-live). Obviously, not a very efficient search strategy, regarding both the amount of messages sent and the accuracy of the routing algorithm. Jordan Ritter presents in his paper [32] some interesting computations on the amount of traffic generated only by search requests and the appendant result messages in the Gnutella network. Schollmeier et al. [31] show in their work that this is only theory that does not apply to the real-world Gnutella network. One of the reasons they give is that the network is finite. The formulas used in [32] only apply to a infinite network size. Furthermore, Gnutella's topology is not a tree, which would be a prerequisite for the worst-case scenario. The Gnutella network shows a lot of loops, which prevents messages from multiplying by the number of connected neighbors every hop. Another reason why the number of search messages does not explode as depicted in [32] is the small-world property of Gnutella (see Jovanovic et al. [33]), which means that there is a high clustering coefficient in the network. The conclusion of Schollmeier et al. is that the probability of having a tree as Gnutella's topology is near to zero. Anyway, it is clear that the flooding algorithm of Gnutella is still far from perfect.
Improvements like depth-first iterative deepening [55] (which is used in artificial intelligence), random walks [56] or Directed BFS [57] have been proposed recently. They provide some significant changes to the efficiency of the search mechanism but exhibit longer search times or less search results. The conclusion is that wild mesh P2P systems are, despite all efforts to minimize the traffic, still the most inefficient and most bandwidth-consuming type of P2P networks.

Structured P2P: The average path length of a lookup is approximately equal for all widely known structured P2P networks. For CAN, the mean path length is $\mathcal{O}(\frac{d}{4}n^{\frac{1}{d}})$ (which can be transformed to $\mathcal{O}(n^{\frac{1}{d}})$) in a d-dimensional space. Chord has a typical path length of $\mathcal{O}(\log n)$, in Pastry it is $\mathcal{O}(\log_{2^b} n)$ (which equals to $\mathcal{O}(\log n)$). Tapestry has also, as Chord or Pastry, $\mathcal{O}(\log n)$ as the cost for searching in the network [54]. In all formulas, n is the number of peers connected to the network. The number of messages needed to search for information in a structured P2P network can be reduced by increasing the size of the routing table. The more entries in the table are available, the higher the chance that the message finds its way faster to the destination. However, this comes at the price of higher bandwidth consumption for maintaining the routing table.
No matter how useful these assessments are: structured P2P networks provide a very efficient search functionality. Compared to any other server-less P2P system, this kind of topology provides the most efficient (which is not necessarily the best) search method.
The other type of message that is sometimes not even mentioned in the literature about structured P2P systems is the maintenance message. There exists little information on how many messages are necessary to keep up the structure. Periodic messages are needed to ensure that peers are still online. If a peer does not send a message in a specific interval or does

not respond to a recurrent message, actions have to be taken to ensure the robustness of the structure. It is safe to assume that structured P2P networks have the highest costs for maintaining the structure as it completely relies on a fully functional topology that does not allow to be separated.

Server-based P2P: For peers, the number of messages used to search for information is exactly one (maybe more in the event that the server is not reachable and another server has to be contacted). So this is pretty efficient from the peer's point of view. However, the complexity for the server may be a little bit higher. As Yang et al. [30] point out, there are different ways of how servers in a cluster can be connected. They could be working together in a chain. The first server processes the request and forwards the search request to the next server in the chain if it was not able to come up with enough search results. The message is forwarded as long as the minimal number of search results is not reached. The search can start anywhere in the chain as long as the chain is a loop. The chain is not fixed. Any server could be used as the next hop to get better load balancing. The next possibility would be to replicate all the information that is available on the servers. If every server has knowledge about all meta-information stored in the cluster, no additional messages are necessary to process the query. The third way of organizing the information on the server's side is to distribute the content according to some algorithm (as in a hashtable). This has the advantage of enhanced load balancing but has also the problem that meta-information with multiple keywords may be found on multiple servers. Finally, all servers in a server cluster could be completely loosely coupled. No information is exchanged between these servers. The search horizon of a peer is limited to the peers that are connected to the same server. The complexity for searching is $\mathcal{O}(1)$ (regarding the number of messages needed). According to [30], the chained architecture is the best choice for large scale systems like file sharing. The worst scenario is the loosely coupled architecture because the number of search result is limited more than scalability is achieved.

Hybrid P2P: The assessment of the number of messages used to process a query has two facets in a hybrid P2P network. From the viewpoint of the normal peer the number of messages is one. That can hardly be beaten by any other P2P topology that does not use a central server. For the SuperNodes, the situation is a little bit different. When a SuperNode receives a search request, it processes it locally (i.e. it searches the local database) and, in the case that not enough search results have been found, forwards the search request to one of the other SuperNodes. This forwarding is done until enough search results have been accumulated.
The other type of message necessary in an hybrid P2P network is the maintenance message. It is needed to check whether the SuperNode is still available and whether the connected peers are still online. Since a lot of peers could be connected to a SuperNode (up to several thousands), the rate at which connected peers should report in must be chosen carefully. Furthermore, SuperNodes themselves should verify that the other known SuperNodes are still alive.

Scalability

The most commonly used criteria for defining the scalability of a P2P system is the amount of peers it is able to handle without exceeding the bandwidth available. Other measures as CPU power, memory consumption, etc are mostly ignored, which is understandable because the primary bottleneck of a peer is its bandwidth.[1] Estimating the scalability of a P2P model is a very difficult tasks due to the dynamic nature of the peers and the connections between them.

Wild mesh: The dynamics of wild-mesh P2P networks make it almost impossible to estimate how well the network scales. Many papers (e.g. [32], [33], or [59]) deal with scalability issues of Gnutella. It is undisputed that a wild-mesh P2P network without any optimizations like connection preferring, search hubs, caching, etc. is not scalable. [60] reports that the Gnutella network is limited to approximately 10 queries per second, which is created by 10.000 to 30.000 peers. One reason for the lack of scalability of pure P2P networks in general (and especially in wild mesh P2P networks) are peers with low bandwidth connections. Since peers do not only send search requests but also relay search requests for other users, the low bandwidth capability of these users is very soon exhausted.

Structured P2P: In the world of pure P2P networks, structured topologies that act as a distributed hashtables have the highest chances of being scalable. The problem is that not a single structured P2P system has ever been used on an Internet-scale. There exists no real-world data on the scalability of these systems. There do exist simulations but it is very hard to simulate all the side-effects that could happen in the real world.
Scalability also depends on the usage of the P2P system. When the majority of messages transferred is used to search for data, these systems are scalable up to millions (or more?) nodes. This is also fortified by the fact that the average search path has only $\log n$ hops. This means that in a network of 1,000,000 nodes it takes only six hops to reach the node that holds the requested information. However, if there are considerably more updates (i.e. joining or leaving nodes, changed content, etc.) than searches, scalability of these system declines. The typical cost for inserting or removing a node is $\mathcal{O}(\log^2 n)$ messages. Still, this is not a big disadvantage.
In systems where content (or meta-data) is moved to those places where it is expected to reside, there is another issue to be considered. What if there exist a lot of information items that are to be located in the same area, so-called hotspots? Since peer capacity is finite, there might be soon no free resources (in terms of memory and CPU power) left. Hence, content may not be stored where it is expected. If this happens, there are two ways of resolving the situation. The first step is to split the search space into fragments, making place for other peers to take over parts of the information. The second possibility would be to use replication techniques to build clusters of peers that establish a virtual node in the structure.

Server-based P2P: It is very difficult to tell whether a server-based P2P network is scalable. Many factors have to be considered when evaluating the scalability. It is not sufficient to simply say that this kind of topology is not scalable because there is a single bottleneck. It is also a question of what should be achieved with this P2P network. The application domain influences whether server-based P2P systems are scalable enough. The average message

[1]According to Backx et al. [58] most P2P system only consume about 1% of CPU power and at most 3% of the available local file transfer rate.

size, the frequency of search requests, the rate of joining and leaving peers and the internal implementation of the server mainly affect the scalability of the system. The latter point is especially interesting because different schemes of arranging servers in a cluster may provide different search horizons and, therefore, scalabilities. Additionally, the type of users connected to the network are important. If most of the users have a slow modem connection, server-based P2P networks may scale better than pure P2P networks. The same applies to small devices. If peers are not able to be part of a pure P2P network, server-based P2P networks may be better. The bottom line is that the scalability of a server-based P2P network has obviously a limit (the limit of the server) but this limit could be high enough.

Hybrid P2P: Whether a hybrid P2P system is scalable depends, as in the case of a server-based P2P system, on the requirements it should fulfill. If it is necessary that a search request visits all SuperNodes to guarantee complete search results, it is hard to argue that this system is scalable. If, on the other side, only a limited number of matching files is necessary to make a successful search result, the hybrid P2P system is scalable (there is no upper limit).

Search functionality

Different P2P networks provide different ways of searching for information. Ideally, the topology of a peer-to-peer network should not have an impact on the search functionality (or *expressiveness*). Unfortunately, this is not true. It is the case that any P2P system can adopt a meta-data model like that provided by Kazaa. But there are exceptions.

Wild mesh: Since there exists no routing that is strictly based on content, the way in which content is described has no influence on the behavior (i.e. the routing) of the network. A problem that most likely occurs in wild mesh P2P networks is the heterogeneity of the peers. If the network is only defined by the protocol (as is the case with Gnutella), it is not clear how searches are actually performed at the peer. This could lead to unexpected search results. The reason why especially wild mesh P2P might have this problem is the fact that the peers are very loosely coupled in this topology. There exists no central authority that might enforce standard compliance.

Structured P2P: A disadvantage of typical content-based routed P2P networks (distributed hashtables) is their search functionality. Usually, a peer can only search for a number. This search functionality is useful where a single ID is sufficient to identify content. There are plenty of use cases where this approach is sufficient. Still, in some use cases it might be not satisfactory to search only for a specific key (or key-phrase). In the most widely applied use case of P2P networks, the sharing of files, it is necessary to search for more information than simply the filename (or parts of it). A solution would be to register all elements of the meta-data available for a file. But this could mean that a single file is registered several (ten to hundred) times in the network.
The résumé is that searches in distributed hashtables is, while very efficient, afflicted with major constraints.

Server-based P2P: The search functionality of a server-based P2P network is not limited by the topology at all. It fully depends on the implementation of the search functionality in the server and the server architecture.

Hybrid P2P: As in wild mesh and server-based topology, the search functionality of the P2P network only depends on the implementation. There exists no restriction by the topology itself. Hybrid P2P system have the same problem as wild mesh P2P systems: the search functionality is typically not defined by the protocol or the topology. In structured or server-based P2P network, the way in which searches are performed is known. In hybrid P2P networks, different implementations with the same content could produce, although adherent to the protocol, different search results.

Search results

There are several ways of evaluating search results. One measure would be to analyze the hit rate (or the rate of false positives). This identifies how many search results really match the requested information. But the hit rate mostly depends on the granularity of the meta-data model used. Hence, the hit rate will not be taken for a comparison of topologies. It is more interesting to look at how many messages are necessary to get a satisfying result. The number of messages vs. the result-satisfaction ratio depends highly on the routing algorithm used. Another aspect when looking at the quality of search results is their *completeness*. Does the P2P system provide complete searches? *Complete searches* means that searching for a file that is shared by another peer is guaranteed to succeed.

Wild mesh: In wild mesh P2P networks, the unstructured routing has very bad effects on searching. A search message gets relayed until the TTL has been reached. In the worst case, when a search request is sent out, a total of

$$n\sum_{i=0}^{t-1}(n-1)^i \tag{3.3}$$

hosts are reached (where n is the average number of live connections a peer has and t the typical TTL). So, when the initiating peer has found what it is searching for, it cannot stop the message from propagating and wasting bandwidth.
When searching in a wild mesh P2P network, messages containing search results are coming in arbitrarily. There is no point in time where the client can be sure that no more search results may be expected. Furthermore, due to the very frequent change of peers currently online, two identical search requests do not necessarily provide the same results. Peers are coming and going frequently which has the effect that messages are forwarded to different peers any time a search request is sent. Another problem with searching in wild mesh P2P networks is the fact that only peers that are currently online will process a search request. Since the average time a peer is online is very low (e.g., from just a few minutes to an hour), most of the peers that participate in the network cannot be searched. Another drawback of this kind of P2P network is that searches are not complete. This means that it is not guaranteed that a peer can find the information even if it is available in the P2P network. Only a small subset of the overall network may be covered by a single search request.

Structured P2P: Structured P2P networks have the highest probability of finding content if it is available. The technique of placing the content exactly where it can be found again (as in normal hashtables) has the effect that search messages will find the requested information in any case. Search requests are always forwarded to peers that have a higher probability of storing the information (or, in other words, whose key is nearer to the requested key of the content).The only reason why an information could not be found is when the peer holding the information in question is no longer available. Replication techniques mitigate this problem.

Server-based P2P: Whether server-based P2P networks are able to provide guaranteed search results and how many messages are necessary depends solely on the server cluster architecture (if it is a cluster at all). If the cluster is arranged in a way that there is no information exchange between the servers (be it replication of stored meta-data information or the forwarding of search requests), the search horizon is limited and therefore complete search results impossible. If, on the other side, a chained architecture is used or information is replicated among all servers in the cluster, search results can be guaranteed (as long as they are available). Yang et al.[30] provide a more exhaustive comparison of these cluster architectures.

Hybrid P2P: Once a hybrid P2P network has reached a certain size, search results cannot be guaranteed to be complete due to scalability reasons. Although SuperNodes may forward search requests to other SuperNodes, it is infeasible that all SuperNodes may be reached while maintaining scalability.

Security

P2P networks offer a lot of security threats and challenges that may be a reason why companies are still skeptical about using P2P software in their environment. The most important problem is authentication. How does a peer know that the remote peer that it is talking to is really the peer it claims to be. Shared secrets are no solution because then every pair of peers must have a singular secret key, which is impossible regarding the potentially huge amount of peers. An answer to this problem would be to use public/private key pairs. But then, a central certification authority would be necessary to verify the public key of the remote peer. This is not feasible in pure P2P systems and in mobile ad hoc networks, where no connection to a central server is available. Another way of solving this problem is to use a so-called "Web of Trust" [61]. This means that public keys are not accredited by a central authority but by other peers that are trustworthy (Sundsted [62] gives an overview on how to use this technique in P2P networks). But this system is not as strong as a central authority because as the chain of trust gets longer, the probability of errors increase (which is more a social than a technical problem). Instead of using a certification authority, centrally issued smart cards are a promising attempt to mitigate this problem. The code running on the smart card can provide the necessary authentication mechanisms. Since tampering with smart cards is very hard, this appears to be a very practical solution. Still, P2P networks that want to support devices with no smart card reader attached will not be able to support this solution.

The next problem is the question of authorization. Now that we have established the identity of the remote peer, how do we know what access right it might have. This information might be kept central (e.g. this group of peers/people do not have access to all account files), but there is no virtue in doing that. Another solution would be to issue certificates to the peers describing what they are allowed to do. In this case, a central authority to maintain certification revocation lists might be desirable for flexibility.

Encryption is the third big cornerstone of security. But in contrast to AA (Authentication & Authorization), encryption of communication between peers not knowing each others has been solved (see, for example, SSL [63]). Peers might also use the existing infrastructure used for authentication to use public/private key encryption. P2P systems that use encryption are, among others, Freenet [18], FastTrack [19] and Locutus [64].

So far, obvious security issues have been discussed. P2P networks, however, bear other security challenges as well. Among them is the verification of content provided by the remote peer. Is the other peer really providing the file it claims to share? One solution would be download a portion of the file and verify this part (if it is possible), for example by verifying the digital signature [62]. Another form of content verification (as used in SETI@Home [12]) is to give multiple peers the same task and compare the results returned by the peers. This way, a peer is not forced to trust the results of a single peer.

One problem that all P2P systems have in common is the antagonism between authentication and anonymity. Most P2P systems use some sort of pseudonym that does not necessarily have a relationship to the real identity of the user. As a result, security must be considered already at the design phase of the development of the topology. "We add security later."™is an argument for many experimental P2P systems, which should make users very suspicious as it is very hard to make an existing system secure if it has not be designed to be so.

Wild Mesh: There exist many scientific and non-scientific reports and papers about the security problems of P2P networks in general and of Gnutella especially (e.g. [65], [66], [67], [68], [69]). But most of the problems described in these articles also apply to other P2P topologies. Bandwidth consumption in corporate networks, security concerns about faulty software that shares more information than the user is aware of, unencrypted communications and much more are not problems of wild mesh P2P networks alone.

There are only a few security problems that apply only (or mostly) to wild mesh P2P networks. Especially in this kind of network authentication is a very delicate task. Without a central certification authority, there is no way a peer can be authenticated. Alternatives like the web of trust may be a solution but are not as strong as a centralized trusted third party (as, for example, the server in the server-based P2P model could be).

Wild mesh P2P networks are the only topology where only the content provider does also provide the search functionality. In other topologies, the owner of the file or services sends meta-data (or the file itself) to those peers (or servers) that are responsible for it. Hence, in wild mesh networks, there is no third party that may filter inappropriate content like spam or "guarantee" a working search service. A peer may return anything it wants, regardless whether it is related to the actual search query or not. This actually happened in the Gnutella network [70] where malicious peers (from *Flatplanet.net*) returned spam content when they received a search request.

The completely distributed nature of this kind of P2P network poses also the problem that detection of malicious hosts creating tons of useless search requests is harder than in any other P2P network. A peer constantly sending search messages and thus flooding the network is not restricted by the protocol to send these messages always to the same set of neighbors. This is also a problem for server-based or hybrid P2P system but there, the list of potential receivers of these messages is considerably smaller than in pure P2P networks. With a smaller set of peers receiving search messages, the detection of peers flooding the system is easier (although it is still not child's play).

Structured P2P: Structured P2P networks have roughly the same problems as wild mesh P2P networks when it comes to security. There are, of course, differences but they have in common that there is no central authority (or trusted third party) that may be used to achieve authentication and, as usual, peers should not trust other peers that they are doing what they are supposed to do.

What is different in structured P2P networks is that peers that provide content do not necessarily provide the search functionality for that content. Usually, the content or meta-data describing the content is transferred to those peers that are responsible for it. Hence, there is a (small) possibility that spam, fake files or other unsolicited content could be removed from the structure by other peers than the owner.

What could be a advantage could also be a drawback. Now that another peer is responsible for the information provided, how can we be sure that the information is not altered by malicious peers? A solution to this problem would be to encrypt the data which makes only sense in P2P networks that are used as a personal storage facility (as in Pastry).

Server-based P2P: Having a single server in a P2P network has some advantages over other topologies. The server could be used to authenticate users (or peers). The server could be used, for example, as a trusted third party for public key certification. The server could also be used for electronic commerce, as it is the case with [71].

Since all peers are connected to the same server (or server cluster), changes to the protocol (or the peer software) could be easily spread by forcing peers to update the protocol. This would be much harder in a wild mesh P2P network where there is no incentive to change the peer software if it still works with (some) other peers. This may be considered as a drawback as well because peers might be forced to update the software against their will.

Hybrid P2P: Hybrid P2P networks have roughly the same security risks as pure P2P networks. Authentication is almost impossible to achieve. It is also questionable whether other peers are trustworthy enough to become SuperNodes (in FastTrack, there is supposedly a second level of SuperNodes manages the SuperNodes and has also a Blacklist of peers that should not become SuperNodes).

Furthermore, it is very easy to sabotage the system by becoming a SuperNode and provide no or only a useless search functionality. But this only affects a local part of the network. All other peers and SuperNodes should not be affected by this malicious peer. Hence, it should not be allowed to a peer to make itself a SuperNode.

Small devices

Small devices are supposed to play an important role in P2P networks in the future [49]. The use of many small devices like sensors creates a lot of information. The question is how this huge amount of information can be processed. P2P networks may be a solution for this problem due to their ability to combine the computing power of many machines. But it does not suffice to use just any P2P network for linking heterogeneous devices together. So, the question is whether a P2P system supports small devices; e.g. by not forwarding traffic through these devices. When working with small devices, special care has to be taken that the available resources are not wasted. Low CPU power, limited operating system possibilities (i.e. no threads), low bandwidth capabilities, small displays and many other problems arise when designing systems for small devices. Devices could be, among others, PDAs, mobile phones or sensors.

Wild mesh: Whether a P2P system can be used on small devices, depends on the application domain of the P2P network. When the P2P network is only needed to exchange email addresses and synchronize calendars in an ad hoc network, wild mesh P2P networks would be a good choice. In cases where a lot of traffic is produced, other topologies might be better. The routing algorithm of pure P2P systems is the biggest problem for small devices. The idea of this topology is that every peer contributes to the system in the same way. This approach is very inefficient when devices with very limited CPU power and bandwidth are involved. In this case, a separation between powerful peers and peers with limited capabilities (as in server-based or hybrid P2P systems) is necessary. However, there are also advantages compared to other P2P topologies. Small devices may also be mobile devices, which typically have a limited transmission range. So, a resilient P2P network that allows for frequent disconnections is favorable. Furthermore, in a wild mesh P2P network, no additional communication to set up the network (e.g. finding or electing a central server) is needed. By issuing a broadcast, every peer in the area may be found and included in the P2P network. For ad hoc networks with adjusted bandwidth requirements, wild mesh P2P networks might be the best solution.
Structured P2P: The low number of messages needed to search for content in structured P2P networks may lead to the conclusion that they are a good choice for small devices. However, there are a few things to be considered. Small devices have typically only a limited memory capacity. Hence, it might be not possible for such devices to store a routing table. If the device is only a small sensor with very few capabilities, it might be impossible to insert it in a structured P2P network. On the other side, routing tables are not very big (usually $\log n$ entries). Another problem that might occur is that in some distributed hashtables, all peers are requested to be part of the structure. Hence, no differentiation between powerful and small peers is made. Therefore, these devices might become very soon a bottleneck in the P2P network. At least, care should be taken that small devices are not placed in hotspots or along the routes to hotspots.

Server-based P2P: Server-based P2P networks are particularly interesting for small devices. Devices would be able to send their data to a central server for further processing. There is no interaction with the other devices involved. A limiting factor is the scalability of the system. If there are too many devices connected to a single server, other topologies might fit better. It is also assumed that the costs for sending data to the server is not higher than sending data to other peers (i.e. the power used for transmitting data over a WLAN network).

Hybrid P2P: The beauty of the hybrid P2P topology is its flexibility. Depending on the configuration and the environment at runtime, it could be used as a wild mesh or as a server-based P2P network. If there exists at least one powerful peer in the nearby, this peer could become a SuperNode and provide bandwidth and CPU power to all small devices that are connected to this SuperNode. Hence, small devices have to maintain only a single connection to the SuperNode and do not have to provide any search functionality. If the design of the network allows the replication of data, it would even be possible to replicate all information to the SuperPeer and the small device could disconnect to save power and bandwidth. If there is no powerful peer available, all peers could start as SuperNodes. This would have the effect that all peers are equal, which would create a wild mesh P2P network. Hence, no complex restructuring is necessary although this also means that searches are less efficiently processed.

Vulnerabilities

In addition to general security problems like encryption, authentication etc. there is also the threat of having vulnerabilities in a P2P systems. This means that not a single peer might be attacked by malicious peers (or lawyers) but the whole network. An important aspect, for example, is the existence of a single point of failure. Another question is whether the system is safe from attacks by individuals or organizations (as proposed in the US, where the RIAA wants to actively attack peers sharing content protected by intellectual property laws [72]). There are several other questions, like: Is there a single point of failure? Can peers take the identity of other peers (if it is useful in attacking the system)? Can the P2P network be corrupted by peers that reply with false positives for every request? Can a peer push itself into a position (e.g. a SuperNode) where it can do more harm than "normal" peers? These questions depend not solely but to a considerable degree on the topology of the network.

Wild Mesh: A way of disrupt or even destroy (at least for a short time) a wild mesh P2P network is to create a lot of peers that may even provide useful data. This situation could be used in two ways to mutilate the system. First, when creating a lot of peers that do not provide any content, these peers could be used to fill up the "neighborhood set" of other normal peers. Every peer has live connections to a specific number of other peers. The more of these bogus peers are within this set of peer, the better for the attacker. This way, the malicious peers can infiltrate the system and hinder normal peers from finding content. A lot of highly connected hosts may also be used to fragment the network in many small pieces. If all bogus peers shutdown at the same time (the so-called "meltdown"), many normal peers loose communication and the network gets split up in many small fractions. It would take some time until the network is connected again. However, this technique is very costly and needs a lot of processing power and bandwidth to be accomplished. The RIAA tried to use this technique but had to give up due to costs and the inefficiency of this attack.

Structured P2P: Most widely known structured P2P networks (or distributed hashtables) have been designed to perform efficient searches and to remain stable, even in the face of tremendous peer losses. What has been treated like a stepchild is the resilience to malicious peers.

The best way to sabotage a structured P2P system is to attack content hotspots and routing hotspots (or, in other words, those peers where a lot of traffic is coming through, if they are identifiable). One possibility would be to overcrowd hotspots with similar keys. This does not have the effect that the normal data cannot be found anymore. What it does is that those peers responsible for these keys get overloaded with information. There is slight possibility, depending on the implementation of the P2P network, that the correct content is removed to make room for this new (fake) data. More probable is that other peers are used to split the information stored on the hotspot. Hence more peers are needed to hold the inserted information. This could lead to unbalanced structures where a lot of peers are bound to a single place. Sophisticated accounting mechanisms as, for example, in PAST mitigate this problem.

Another way of doing a lot of damage to structured P2P systems is to insert malicious peers into the network. These peers could be used to route messages to other destinations than they are supposed to. This way, loops could be generated. Since loops are not possible if all peers work correctly, there is maybe no technique designated for detecting loops.

Compared to wild mesh P2P networks, there are some advantages when having a distributed hashtable. The most obvious one is that flooding would take much more bandwidth on the attacker's side to flood the network. Messages do not grow exponentially as in wild mesh P2P networks.

Server-based P2P: The good news is that in this kind of technology there is only a single point that is vulnerable. The bad news is that it the most important part of the system. If the central server is down, the P2P network does not work.

There are several ways how the server could be rendered useless. A DDOS (distributed denial of service) attack could be used to flood the server with search requests or fake messages making the server believe new peers join the network. This is especially dangerous because the attacker has only to concentrate on a single machine (or cluster) instead of multiple peers, as is the case in the other topologies presented here. Furthermore, the law could be a reason to stop the server (a possibility not too far-fetched). Generally, the central server could be unreachable for trivial reasons like a disconnection from the Internet by the ISP.

Hybrid P2P: As in server-based P2P networks, if a SuperNode ceases to exist, the data stored by this SuperNode is no longer available. The connected peers have to find another SuperPeer to connect to. Hence, the data is not lost but unavailable for a certain period of time. But what happens if many SuperNodes disconnect at the same time (because all of them belong to the same evil company)? Since peers store addresses of potentially many other SuperNodes, this kind of attack is not very promising.

In file sharing networks, jurisdiction might concentrate on SuperNodes because they could be accused of supporting copyright infringement. It is questionable whether this is still possible if the user does not know whether his/her peer is a SuperNode or not and has no possibility of influencing this decision.

3.2 Summary

Four classes of P2P topologies are discussed in this chapter: wild mesh, structured, server-based and hybrid P2P systems. There exists no clear dividing line between these four flavors of P2P models. All of them could be somehow related to the others with some minor distinctions. However, even these small differences influence the applicability of P2P systems for certain use cases.

For the choice of the right P2P topology, it is important to know what exactly it is needed for. Requirements like scalability, survivability, support for ad hoc networking are as important as, for example, information about the ratio of search requests per second vs. the frequency of joining and leaving nodes or the size of the memory a peer might have.

	Wild Mesh	**Structured**	**Server**	**Hybrid**
Ad hoc mobile communication	good	average	poor	good
Connectivity overhead	poor	poor	good	average
Fault resilience	good	average	good	average
Message count	poor	good	good	average
Scalability	poor	good	average	average
Search functionality	good	poor	good	good
Search results	poor	good	good	average
Security	poor	average	good	average
Small devices	good	poor	good	average
Vulnerabilities	good	average	good	average

Table 3.1: Comparison of P2P topologies.

Table 3.1 summarizes the strengths and weaknesses of the topologies regarding the evaluation criteria used in this chapter.

In this work, some aspects of P2P networks have been evaluated with different topology classes. It was not the goal of this chapter to evaluate existing P2P system but rather the underlying topologies these systems have.

However, this is not the end of the story. The next generation of P2P systems will have to employ a flexible topology. It will not be sufficient to just choose a topology and do everything on top of that. It is more likely that the topology must adapt itself to the requirements and capabilities of the underlying network and hardware. In some cases, a wild mesh topology fits better than the others while in other cases, another topology might be more appropriate.

What is needed is a P2P system that allows to change its topology according to the requirements of the application. The design of such a system is presented in the next chapter.

Chapter 4

Design of Omnix

Peer-to-peer is [...] not an application, it's an infrastructure.
Chris Shipley [73]

This chapter deals with Omnix, a topology- and platform-independent P2P system carried out as part of this thesis. This chapter presents the challenges dealt with in the design of Omnix. It includes a detailed discussion of the architecture and the design of the major building blocks of Omnix, and discusses the topology module interface.

4.1 Requirements

The goal of designing a layered P2P middleware architecture can be broken down to the following main requirements. In addition, some requirements have been identified that cannot be directly deduced from the need for a layered architecture. We nevertheless consider them as being important to provide a useful P2P middleware system.

Platform-independence

To achieve true platform- (or device-) independence, the connecting points between the P2P middleware and the underlying operating system services must be replaceable. That means an interface must be designed to create an abstraction of the underlying OS. Furthermore, the middleware must be stateless: whenever the device running the middleware is restarted, it is not guaranteed that the information of prior sessions is still available. Finally, the middleware must not use any language-specific functionality (e.g., Java RMI, DCOM, etc.).

Small footprint

In the context of platform-independence, it is also important for a middleware system to have a small footprint. To be able to run the middleware on a wide range of devices (with varying capabilities; e.g. mobile phones, PDAs, or embedded systems), the middleware must be as small as possible. To achieve this goal, only a thin core should be used in combination with additional components and libraries. This approach can be compared with microkernels as opposed to monolithic kernels in operating systems.

In addition to the ability to run on various devices, it is also important to produce as little overhead as possible. On resource-constrained devices such as mobile phones, only little

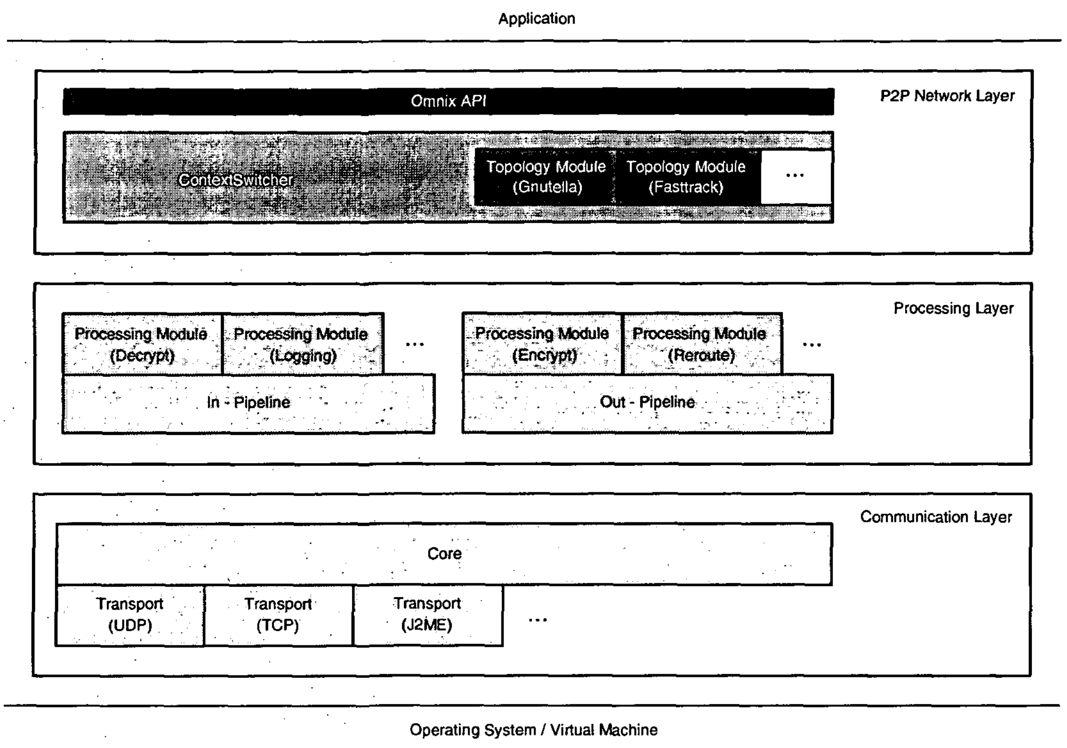

Application
Omnix API
P2P Network Layer
ContextSwitcher
Topology Module (Gnutella)
Topology Module (Fasttrack)
...
Processing Layer
Processing Module (Decrypt)
Processing Module (Logging)
...
Processing Module (Encrypt)
Processing Module (Reroute)
...
In - Pipeline
Out - Pipeline
Communication Layer
Core
Transport (UDP)
Transport (TCP)
Transport (J2ME)
...
Operating System / Virtual Machine

Figure 4.1: The layered, modular architecture of Omnix.

memory and CPU capacity should be consumed by the middleware. At the time of writing, a typical J2ME enabled mobile phone, for example, allows an application with a maximum size of 64 kB.

Topology-independence

As shown in Section 2.2.2 (and in greater detail in [6]), being independent of the P2P network structure is an important requirement for a general purpose P2P middleware. Although it is not entirely impossible to meet the requirements of all possible use cases with one specific P2P topology, it may be a tremendous challenge (if not impossible) to achieve this in a efficient way. As an example, distributed hashtables could be used (somehow) for searching files based on meta-data but it is not the best solution.

Since we cannot tell for what purposes a P2P middleware will be used (the use cases described in Section 2.2.2 are only a subset of all possible scenarios), the P2P middleware must be able to adapt its topology. This flexibility not only comprises the independence of the topology at compile time but should also enable the application to change the topology according to its requirements at runtime.

Openness

The P2P middleware must be designed in a flexible and extensible way. This includes the network protocol used, the structure of the message, the ontology of the content of such messages as well as the way how messages are processed.

The main reason for this requirement is: adaptability. Omnix must Application programmers must be able to change every aspect of the middleware platform that affects its functionality. An example where this might be needed is the heterogeneity of existing P2P systems. It might be desirable to modify the middleware in a way that it uses standard Gnutella messages to communicate with other peers. This would enable communication with other, proprietary P2P networks. The challenge in this case is to provide an interface that covers the commonly used aspects of P2P networks.

Testing

One of the most difficult tasks in the creation of P2P systems is the evaluation of the network. There are many properties of an P2P network that needs evaluation before it can be safely used on an Internet-scale: 1) overall performance of the network (i.e., the average throughput of the network connections) 2) the average time until a satisfying search result is obtained 3) the localization and the removal of hot spots and hot routes in the network 4) the scalability of the system and 5) much more.

There are many ways how a P2P system can be tuned to change its properties. In the Gnutella network, for example, the number of average connections a host maintains, the typical hop count of search requests, or the introduction of so-called search hubs have great effects on the performance and scalability of the system.

It is obvious that these changes cannot be made easily once the system is running on thousand of nodes. Hence, it is very important to get experience on how the system reacts to configuration changes and how to find the optimal settings beforehand. This can either be done by employing difficult mathematical methods (of which the results may not be trustworthy due to the sheer complexity of the problem) or by facilitating simulators. A more detailed discussion about the assets and drawbacks of various simulators is given in Section 6.4.

Hence, Omnix must provide means to simulate thousands (or millions) of nodes in a virtual P2P network.

4.2 Architecture

This section presents the architecture of Omnix. For each component of the architecture, it will be shown how it contributes to fulfill the requirements described in Section 4.1 and, where appropriate, alternative designs are discussed and evaluated.

4.2.1 Overview

The Omnix middleware is located (like any middleware) on top of the operating system and provides an interface to an application on top of it. This interface provides a set of services that allows the application to access the functionality of the P2P middleware.

Omnix itself is divided into three layers, the *Communication Layer*, the *Processing Layer*, and the *P2P Network Layer*. Figure 4.1 shows a high-level view of these layers with their main components.

The Communication Layer (which can be compared to the *physical layer* of the OSI model) is responsible for the actual sending and receiving of point-to-point messages to and

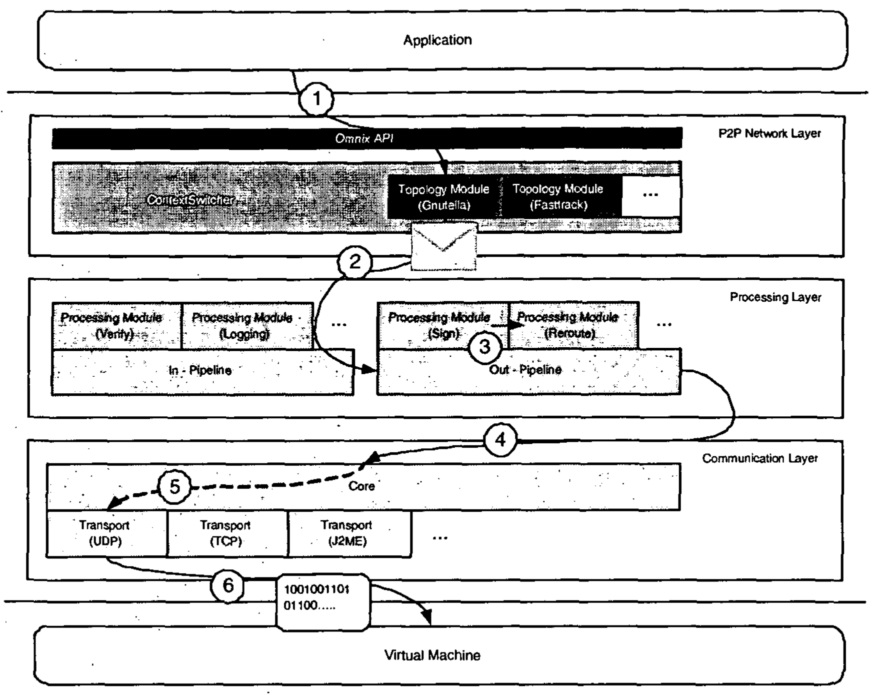


Figure 4.2: Walk-through of sending a message in Omnix.

from peers across the available "physical" network (e.g., the Internet). It consists of transport modules and a core module that is needed for the coordination of all transport modules in this layer.

The Processing Layer (an extension of OSI's *data link layer*) deals with the processing of incoming and outgoing messages. It consists of two message pipelines, one for the peer's outgoing and one for its incoming messages. A message pipeline comprises a list of processing modules, each acting upon a received or to-be-sent message. Processing modules provide basic services like, for instance, encryption/decryption of messages.

The P2P Network Layer (a combination of the *network layer* and the *transport layer*) consists mainly of *Topology Modules*, which are coordinated by the *Context Switcher*. It is the topology module that contains the most important part of the P2P middleware: *routing*. The Omnix API allows the application to access the services provided by the middleware.

Typically, middleware operates only in the upper layers [74] (e.g., provide services on top of TCP/IP or UDP/IP). But this is no longer true in the P2P area. The reason for this is that P2P networks create an overlay network, a *network on the network*. P2P middleware has to start again at the physical layer (the name is kind of misleading because we do not actually access the physical layer but those primitives provided by the operating system). Omnix only covers the layers 1 to 4. The reason for this design decision was that, to enhance flexibility, the layers 5 (*Session*) and 6 (*Presentation*) should be covered by the application (as an analogy: in the Internet, TCP/IP covers only the layers 1 to 4, too. All protocols in the layers 5 to 6, like SMTP, HTTP, FTP, etc., have their own structure tailored to the specific use case).

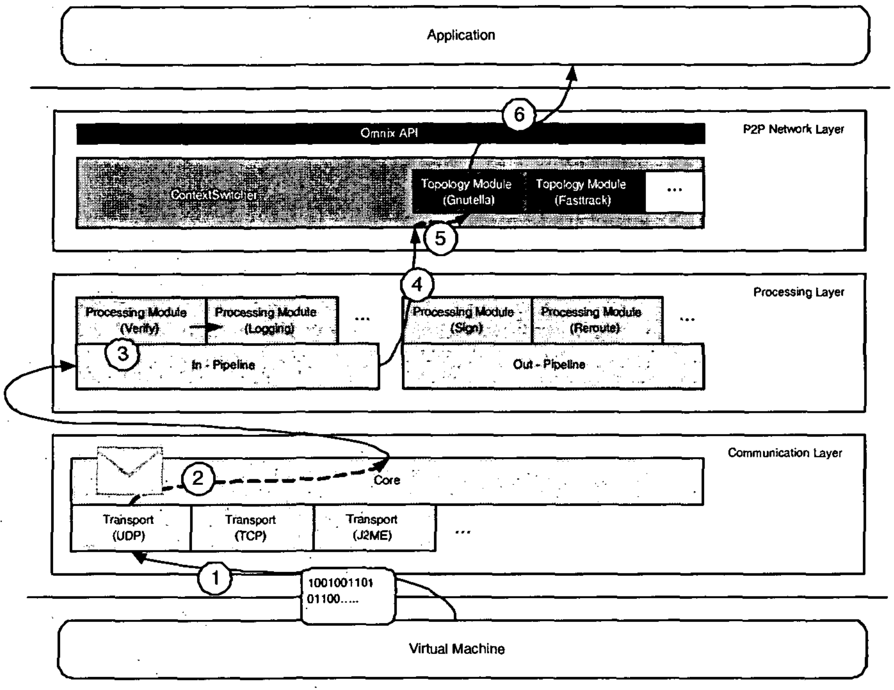


Figure 4.3: Walk-through of sending a message in Omnix.

Figures 4.2 and 4.3 exemplify how the different parts of the system work together: when an application wants to send a message (Figure 4.2) to another application residing on a peer across the network, it calls the *invoke()* method specified in the Omnix API ①. This method requires information such as the contact information of the target peer (e.g., an IP address) and the content of the message. The topology module then creates a self-describing message object out of this information and puts it into the outgoing message queue (i.e. the out-pipeline) of the processing layer ②, where it gets processed by the installed processing modules. A module, for example, adds security-related features like a digital signature ③. Once the message has passed the outgoing processing pipeline, it is given, via the Core ④, to the appropriate transport module which conveys the message over the network ⑤. The Core chooses the correct transport module according to the system's configuration.

On the target peer's side (Figure 4.3), the message is received by a transport module ①, parsed and put into the incoming message pipeline by the Core ②. In the incoming pipeline, the message gets processed in the same way as in the outgoing pipeline on the sender's side. The digital signature is verified by one of the installed processing modules ③. After the pipeline the context switcher gets hold of the message ④. According to the meta information stored in the message, the message is dispatched to one of the topology modules installed ⑤. Finally, the message is forwarded to the application above ⑥.

Note that it is not compulsory to use the components of the Processing Layer or the P2P Network Layer. It is also possible for an application to directly access the Communication Layer. This way, applications enjoy an even higher flexibility, which might be necessary if the Omnix API does not provide the required means for a specific functionality.

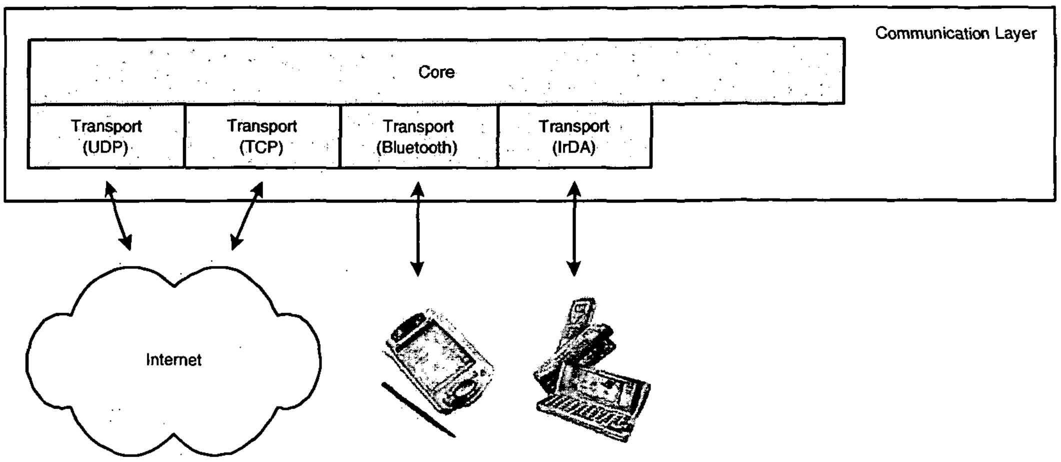


Figure 4.4: Support for heterogeneous devices and protocols.

In the following sections, the components of the Omnix architecture and mechanisms are described in more detail.

4.2.2 The "Physical Layer"

4.2.2.1 Transport

Supporting heterogeneous platforms and protocols

Almost every P2P system currently available has only been built for working in the Internet's IP network (i.e., they are using plain TCP/IP or higher-level protocols such as HTTP or TLS, which are all based on the IP network). Omnix takes the next step and embraces other network types as well (e.g., Bluetooth).

The problem that we see is that there exists no common network access interface for all these types of networks. Bluetooth, for example, for which third party libraries like BlueDrekar [75] or OpenBT [76] are needed, cannot be accessed in the same way as the IP network.

The solution to this problem is an abstraction layer between the communicating software and the underlying networking services provided by the operating system, the virtual machine, by additional libraries. Having such an abstraction makes it possible to provide a common interface to all types of networks and protocols without the necessity of changing a single line of code in the remainder of the system or the application. To achieve this, Omnix uses so-called *transport modules*. A transport module is responsible for sending and receiving messages between two peers. How the network is actually accessed to perform these operations is transparent to the upper layers. For the Core, which sits on top of these transport module, it makes no difference whether a transport module accesses the IP network, uses Bluetooth for communication, sends a message to another peer via infrared link

```
public interface Transport extends Runnable {

    public void addInputMessageListener(MessageListener ml);
    public boolean removeInputMessageListener(MessageListener ml);

    public boolean sendMessage(Message msg);
}
```

Figure 4.5: The Transport module interface.

or uses some other network technology (Figure 4.4). Figure 4.5 shows the Java interface of the transport module.

Allowing multiple protocol structures

Normally, a message would be simply processed by a parser. But this is no longer sufficient if the middleware is supposed to support multiple protocols at the same time (e.g., SOAP requests, Gnutella messages, CORBA method invocations, etc.). This is necessary, for example, if a peer wants to use two or more protocols, to build a bridge between different P2P networks. Furthermore, the middleware can then be used for more than just one type of communication (e.g., using one protocol for instant messaging while another protocol is used for sharing files).

To solve this problem, a *Proxy* must be used. For each message received, a so-called *MessageParserProxy* iterates through the list of available MessageParsers and picks the first one that claims to be able to parse the given message (the MessageParser can make this decision by parsing the message headers, for example). The complete workflow of this task is depicted in Figure 4.6. This way, the structure of the message is open to changes.

Once the message is parsed, it is passed to a list of subscribers (*MessageListener*) that have registered to get informed in case a new message has been received. One of these components is the Core.

Making networking transparent

A key requirement when using a layered architecture is not to rely on any network-specific libraries **above** the transport module level. Hence, it is not possible to use, for example, a Java `Socket` or `InetAddress` object in the Core or above, as they are only valid in an IP network. It is crucial that the networking part are completely encapsulated in the transport modules. Above that, only generic components must be used. Therefore, Omnix does not use any `java.net` classes outside the transport modules.

The Communication Layer is designed so that transport modules are completely independent from each other (to be more precise, they are not allowed to depend on the existence of another component, as it would create a dependency that might not be fulfilled by the system's configuration). This has the advantage that multiple transport modules can be used in parallel, which allows an application to use various communication protocols at the same

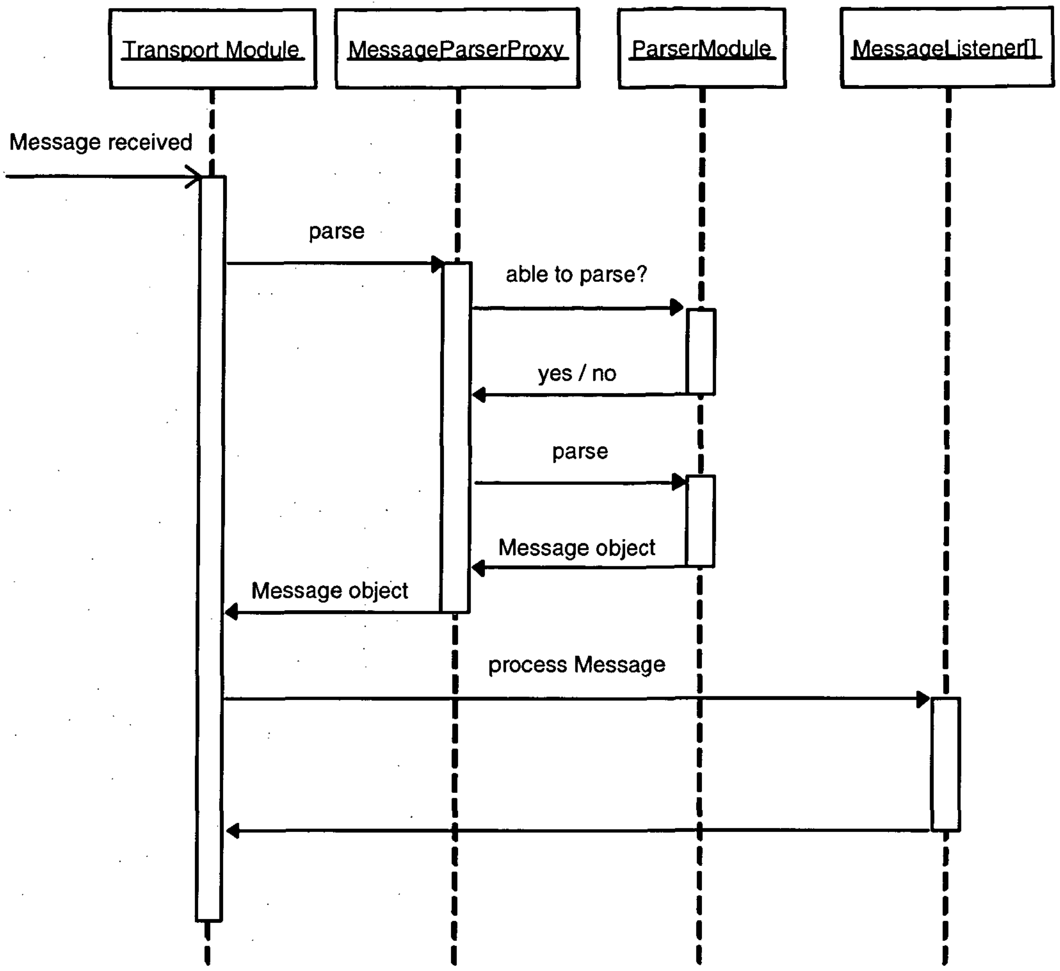


Figure 4.6: Sequence of actions in the Communication Layer.

time. A peer, for example, could connect to peers in the nearby via Bluetooth while other peers are contacted using an Internet connection.

Providing a flexible network address abstraction

To identify a peer in the IP network, an *address:port* pair is sufficient. In other types of networks, however, this is no longer the case (as, for example, in Bluetooth). Hence, it is necessary to use a network independent addressing scheme above the transport modules. JXTA achieves this by assigning a globally unique ID to every peer in the network. It uses only this ID to address remote peers. Whenever a message is sent to a remote peer, the component responsible for sending the message has first to determine the actual address of the destination peer by resolving the ID.

Omnix provides the abstraction of a *Contact*, which represents a remote peer. The content of a Contact is not fixed and depends on the type of the network, but would be in most cases

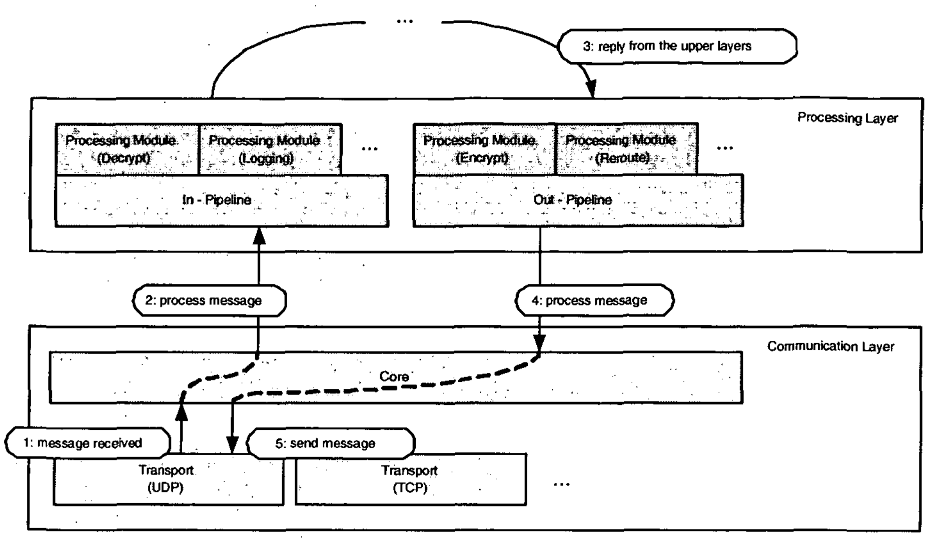


Figure 4.7: Omnix-Core dispatching between Processing Layer and transport modules.

a URI-like identifier (e.g., an IP address). However, a Contact can also be a unique ID or a set of descriptions (e.g., the Message Bus [77] uses an addressing scheme that uses multiple tuples describing the destination).

Upper layers use Contact objects without knowing the internal structure or the actual addressing scheme hidden behind the interface. Thus, it is not necessary for any lookup mechanisms when a message is sent to a remote peer.

4.2.2.2 Core

The *Core* is responsible for organizing the transport modules and for dispatching messages between the Processing Layer and the transport modules. If a message has been received by a transport module, it is passed to the Core, which delivers it to the *In-Pipeline* (explained in Section 4.2.3.1) of the Processing Layer. If the Core gets a message from the *Out-Pipeline*, it must determine which transport modules to use for the sending of the message (this information is stored in the message or in the system's configuration). If no transport module is specified, a default transport module is used. Figure 4.7 shows these steps schematically.

4.2.3 Common Services

Allowing pluggable components to change the behavior of the system

In the OSI model, the second layer (*Data Link*) is mainly responsible for providing a channel free of detected errors. Omnix provides this functionality and more in this layer. The main task of this layer is to provide common services for all types of P2P systems and to keep the Core as thin as possible. We decided to introduce pluggable components that can process incoming and outgoing messages.

The important aspect in this context is that these pluggable components are in a layer between the transport layer and the routing layer (i.e., the P2P Network Layer). Hence, plugins in this layer operate on point-to-point messages between two peers. This makes it possible to introduce services that are necessary for all P2P topologies (e.g., encryption, error correction, logging, etc.). In JXTA, there exists no such pluggable layer between the *Messenger* and the routing layer.

4.2.3.1 Pipelines

Organizing the components

How can these plugins can be managed by the Omnix middleware? If a message is received or to be sent, it has to be processed by the processing modules. In the design of Omnix, we has do decide, whether plugins should be autonomous or work in concert. Furthermore, we had to decide whether plugins should process messages sequentially or whether loops should be allowed.

We decided to design processing modules to work autonomous (i.e., they are not allowed to depend on the output of other processing modules). The reason for this decision was that modules should have a specific objective that they can reach without the help of other modules. This increases robustness as a module works in any system configuration.

The next design decision we had to make was whether Omnix should use a structure of plugins that allows loops and conditional branches. The problem with this approach is be that the complexity of the system (i.e., of the processing modules) would demand special logic in the processing layer (e.g., to detect and avoid infinite loops). As the processing layer can only provide simple, point-to-point services we do not consider such a complex organization of plugins as necessary. Hence, we decided to arrange processing modules in linear pipelines.

The Processing Layer consists mainly of two *pipelines* (or *channels*). One pipeline is for outgoing messages and the other for incoming messages. If a message has been received by the Core, it is put into the *In-Pipeline* while outgoing messages from the upper layers are passed through the *Out-Pipeline.*

A pipeline is a list of *Processing Modules* (Section 4.2.3.2). If a message is sent through a pipeline, it gets processed by each of the processing modules sequentially. Specified by the system's configuration, arbitrary processing modules may be inserted into the pipelines. This way, Omnix can be easily extended with additional functionality. Figure 4.8 shows an example of how messages are passed through the processing pipelines.

If a message has passed the In-Pipeline (i.e., no processing module has consumed or discarded the message), it is forwarded to the *P2P Network Layer.* In the other direction, if a message has gone through the complete Out-Pipeline, it is passed to the Core which takes care of sending the message via a transport module.

Each module may stop the message from being processed further. Either the module re-

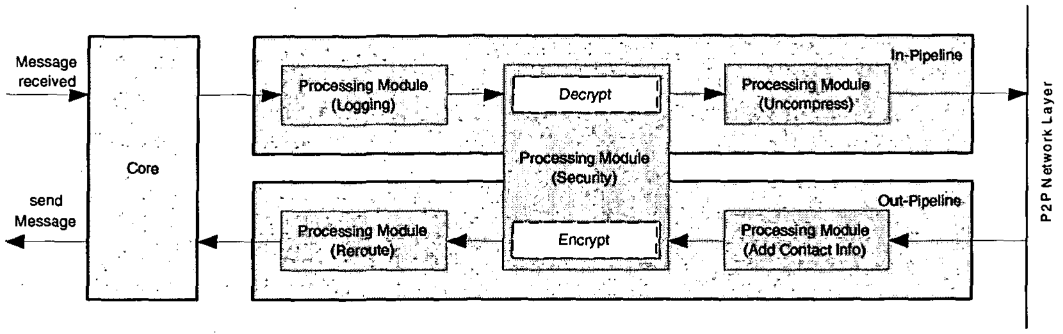

Message received
send Message
Core
In-Pipeline
Processing Module (Logging)
Decrypt
Processing Module (Uncompress)
Processing Module (Security)
Out-Pipeline
Processing Module (Reroute)
Encrypt
Processing Module (Add Contact Info)
P2P Network Layer

Figure 4.8: Example of processing modules in the message pipelines.

turns an error that tells the pipeline not to proceed with processing the message (e.g., because the message content is corrupt) or indicates that the module has consumed the message (i.e., the message has reached the final destination). Examples of what processing modules could be used for are given in Section 4.2.3.2.

It is also desirable to modify pipelines in the Omnix framework at runtime. It is possible to add new processing modules to the pipeline and to remove existing ones. For instance, if a peer needs an authentication module, it may download it from a trusted website and install it in the two pipelines at runtime (note that it is not necessarily the case that all peers in a P2P network must have the same processing modules installed).

4.2.3.2 Processing modules

```
public abstract class Module {

    public int receiveRequest(Request req) { return MOD_OK; }
    public int receiveReply(Reply rep) { return MOD_OK; }

    public int sendRequest(Request req) { return MOD_OK; }
    public int sendReply(Reply rep) { return MOD_OK; }
}
```

Figure 4.9: The Processing module interface.

As described in Section 4.2.3.1, *Processing Modules* can be used as a flexible and extensible way of changing the behavior of the Omnix framework.

Processing message objects

So far, it is not clear what a processing module can actually do with an incoming or outgoing message. Since modules are arranged sequentially in the pipeline, the interface between the pipeline and the processing module plugin can be very simple (e.g., processing modules do not have to interact with other processing modules).

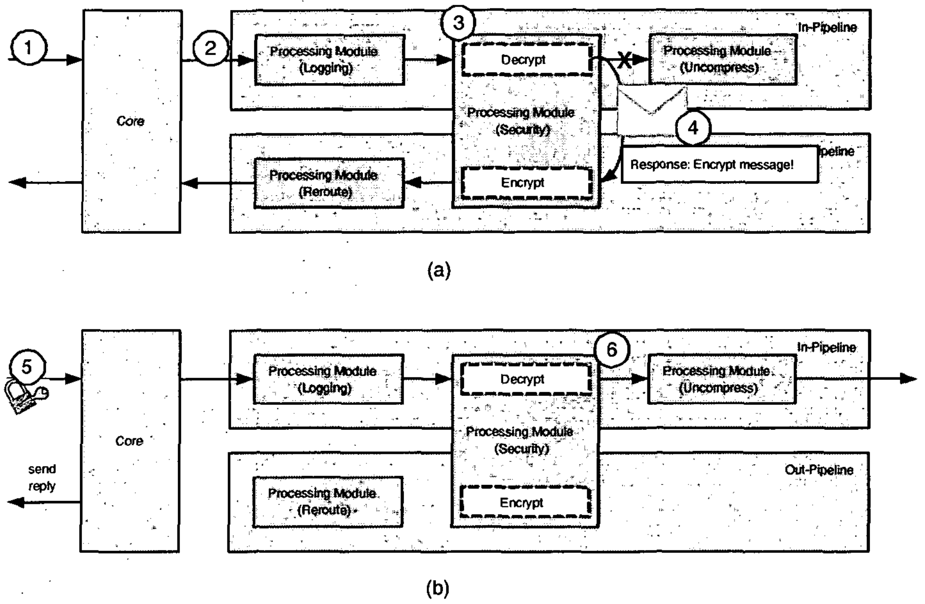

Core
Processing Module (Logging)
Decrypt
Processing Module (Security)
Encrypt
Processing Module (Uncompress)
In-Pipeline
Processing Module (Reroute)
Response: Encrypt message!
(a)
send reply
Out-Pipeline
(b)

Figure 4.10: Processing Messages in the message pipelines.

Figure 4.9 shows the interface between a processing module and the pipeline that it is attached to. A module is prepared to process incoming and outgoing messages. A message could either be a request or a response. In each of these cases, the modules get the opportunity to process the corresponding message.

A module can process a message in the following five ways:

- *Ignore the message (default):* In this case the module takes no action on receiving the message. The message is passed on to the next module in the pipeline without any modifications.
- *Change the content:* The processing modules might change the content of the message before it gets further processed by the other modules in the pipeline. An example could be a module that restores compressed data to its original format.
- *Silent processing:* A module can also process the message without changing the message. An example could be a processing module that logs all incoming and outgoing messages.
- *Send a message* The processing module could also send a message in response to a request received. This message could either be a response back to the originator of the message or a new request to anther peer.
- *Discard the message* It is also possible for a message to discard the message completely. This would make sense, for example, if a sending peer is on a blacklist or if the incoming request is not allowed.

Figure 4.10 shows another example of how a processing module can define the behavior of the system. (a) A message received by the Communication Layer ① is put into the In-Pipeline ②. The processing module that is responsible for encrypting and decrypting messages gets the message and checks whether the message is encrypted ③. In this scenario, the message is not encrypted. Hence, the module creates a response indicating that the sending peer must encrypt the message with the encryption key attached to the response ④. (b) Eventually, the sending peer sends the message again, this time encrypted with the public key of the receiver ⑤. Again, the message gets to the processing module in the In-Pipeline but this time passes the test ⑥.

Processing modules could be used, among many other things, for creating a reliable channel between two peers by verifying incoming messages and resending messages if no feedback is provided. The processing layer thus meets (and exceeds) the requirements of the OSI *data link* layer.

4.2.4 The P2P Network Layer

This layer combines OSI's *Network* and *Transport* layers. The first one is responsible for providing the routing in the network (the most complex part of networking) while the latter is mainly used for keeping track of multiple connections and–potentially–providing reliable connectivity. The challenge that we face here is how to separate the routing algorithm from the P2P services an application typically would need. To solve this problem, we propose an interface that allows an application to use common P2P middleware services without imposing restrictions on the topology of the underlying P2P network.

4.2.4.1 Context Switcher

Supporting multiple topologies and P2P networks – at the same time

When an application is using a conventional P2P middleware, it would be connected to other peers in a network using a fixed topology. We aim to provide a P2P middleware that allows an application to connect to multiple P2P networks at the same time. As pointed out in Section 4.1, one of the requirements for a P2P middleware would be to support multiple topologies.

Hence, in addition to provide access to multiple P2P networks at the same time, Omnix must also allow that these P2P networks use different network topologies. Each P2P network may have its own topology that is completely independent from the other P2P networks that the application is connected to.

This functionality is achieved by *Topology Modules*. A topology module implements the routing within the P2P network. Depending on the topology of the network, different routing algorithms are required. In a wild-mesh network, a peer would send a search request to its neighbors while in a server-based network, the search request would be sent to a central server. All the properties and functionalities that make out a P2P topology are encapsulated in a topology module.

The problem that we face here is, that, if multiple topology modules are installed, it is no longer clear which topology module should receive incoming messages (which also means that incoming messages cannot be assigned to a network). To solve this problem, each message should provide a *network identifier*. All peers in a P2P network must use the same network identifier (which could be a simple string describing the purpose of the P2P network). If a message does not provide such an identifier, it is assigned to the default P2P network.

The Context Switcher is responsible for this coordination of various topology components. Incoming messages are forwarded to the proper topology modules (thus providing multiple connections as in the *Transport* layer) and applications can use the ContextSwitcher to access the topology modules.

4.2.4.2 Topology modules / Omnix API

Providing an abstraction of the routing algorithm

As indicated in Section 4.2.4.1, a *Topology Module* is the place where the topology (i.e., the routing) of a P2P network is located in Omnix. A topology module contains the logic that decides how the topology of the P2P network should be constructed. The logic affects how the peer responds to incoming search requests, where outgoing search requests should be sent to, how peers in the neighborhood are detected, and much more.

For instance, a topology module representing a wild-mesh network such as Gnutella, would receive incoming search requests, process and forward them to other connected peers (with a decreased time-to-live). Note that a topology module does not have to cover all aspects of a P2P network. In a server-based P2P network, there exist two roles of peers. The first one is the normal peer that connects to the server and the other one is the server itself. Although these roles belong to the same topology, they do not have necessarily to be implemented in the same topology module.

For the application, it should not make a difference which topology module is used to communicate with other peers. If an application is searching for a file, it is not important for the application to know which topology is employed to perform the search. To meet the requirement of topology-independence, Omnix provides a generic abstraction of these topology modules. By using a general interface that Omnix provides, the application may access all topology modules in a uniform way. Hence, the application is not bound to a specific topology but might use any topology available without the necessity of changing a single line of code in the application itself.

Specifying an topology-independent service interface

The challenge here is to identify the functionality that is 1) useful within a P2P network and 2) that can be provided by most, if not all, P2P systems currently available. By analyzing the network primitives required by different classes of P2P systems, we identified the set of method depicted in Figure 4.11 to be sufficient to cover a wide range of P2P systems and application domains.[1] An implementation of a topology module does not have to implement all functions of this API. If a method is not implemented, an exception is raised and, hopefully,

[1]Dabek et al.[78] propose a similar approach for structured P2P networks only, which allows them to introduce a more detailed API.

handled gracefully by the application.

```java
public interface Network {

    public String search(SearchDescription sd, SearchResultListener srl);

    public String inject(ServiceDescription[] sd);
    public String remove(ServiceDescription[] sd);

    public ServiceDescription[] injectedLocally();

    public String subscribe(EventSetDescription esd);
    public String unsubscribe(EventSetDescription esd);
    public void notify(EventInfo ei);

    public byte[] invoke(Contact c, String method, byte[] data);
}
```

Figure 4.11: The Omnix API.

The following list gives an overview of the methods defined by the Omnix topology module interface.

Search
This is the most vital method in a P2P network. It lets the application search for meta-data describing the information or services shared in the network. It is not defined how the search should be performed: it is completely up to the topology module itself. There can be no guarantee about the success of the search. Search results may not exist or may be received after a long delay and not as a single block of information. Every time a search result is received, it is immediately forwarded to the given *SearchResultListener* (i.e., the application). The search method must not be blocking (e.g., until a search result has been received).
The ontology and structure of search requests are not defined by the Omnix framework. How search messages and search result messages look like is defined by the application. Omnix provides *SearchDescription* as the interface between the application and the topology module to define the search terms.
Although it might be possible to find a general structure and ontology that covers all possible aspects of searching in arbitrary P2P networks, this is not really desirable. The advantages we would gain by such a system (e.g., the Resource Description Framework - RDF [79]) do not justify the complexity that it creates at the same time. The reasons for this decision are the goal to keep the Omnix framework easy to use and also to have a small footprint, which would not be possible with a very complex search ontology.
The Omnix framework takes a different approach. It lets the application handle the structure of the search messages and their results (because they know best about the structure of these messages). The only requirement is that the corresponding components implement a generic interface provided by Omnix. This is necessary so that the lower layers of Omnix can access the search request and search result specifications in a way that can be transferred over the network.

Inject

Content that can be searched for in a P2P network has first to be published, advertised, or *injected* into the system (in the form of meta-data describing the shared content). The middleware must not impose any restriction on the type of the data injected. It is within the scope of duties of the topology module to define what should happen if an application injects information (e.g., sending it to a central server of storing it in a local database).
As with search requests, the Omnix framework just requires that whatever information is injected into the network, adheres to a given interface (*ServiceDescription*). This interface is needed so that the data structure describing the information to be injected can be accessed by the lower layers of Omnix. There is no implication on the structure, data type or ontology of the service description.

Remove

Removing injected meta-data from a P2P network might be a very complex task. Omnix cannot provide any security mechanisms to prevent malicious peers from removing data from the network without permission (e.g., by allowing peers only to remove the content that has been inserted by the same peer before). Since Omnix has no information about the information shared over the network nor how this information is distributed over the network, it is impossible to provide any security mechanisms. This functionality must be provided by the topology module. Furthermore, the removal of an information cannot be guaranteed in general. If, for example, an artifact is distributed over the network automatically after the injection, it might well be impossible to remove all copies of this artifact.

List local descriptions This method provides a list of all service descriptions stored locally on a peer. While this might sound unusual in many P2P networks (because the service descriptions stored on a local peer are usually the ones that have been inserted by the application), this method makes sense in distributed hashtables and some special forms of wild mesh systems (e.g., Freenet [18]). In both systems, injected content gets either automatically distributed to remote peers or cached at routes between a provider and the consumer. This method might prove useful if an application wants to screen for what (potentially illegal) content it shares system resources.

Subscribe / Unsubscribe / Notify

P2P networks using Omnix can also be used as an event-based system (see [80] for an introduction to event-based systems; PeerWare [81] and DERMI [82] are example P2P event-based systems). Omnix only provides generic interfaces for the definition of subscriptions and notifications. This provides the highest degree of flexibility possible. For Omnix, it is only important that the information regarding subscriptions and events can be transported over the network. When an application subscribes for an event, it uses the *EventSetDescription* interface, which is provided by the topology module. Omnix does not specify the functionality, the content, or the structure of this EventSetDescription. This structure can be very simple (e.g., by identifying a channel by its name) or very complex (e.g., the specification of a set of correlated events).
To produce an event in an event-based P2P system, the application can use the *notify* method. The content of the event and the description of where this event belongs to (i.e., which chan-

nel) is not restricted by Omnix. It just needs a way of getting this information in a form that can be transferred over the network.

Send (Invoke)
For direct communication between peers, peers can also invoke "remote methods". These methods cannot be compared to higher-level communication facilities like CORBA (Common Object Request Broker Architecture), RMI (Java's Remote Method Invocation) or RPC (Remote Procedure Call). Those systems are unfortunately infeasible in Omnix because they 1) need (potentially) scarce resources (i.e., bandwidth and memory space) and 2) they are not supported on all devices that Omnix should be able to run on (e.g., mobile phones). Hence, method invocation in Omnix uses the standard sendreceive pattern, which can be compared to sending a normal message and waiting for a response.
A remote method invocation in Omnix consists of the *contact* information of the remote peer, a textual representation of the method name and the arguments used for the invocation (as a binary array). The return value is also binary encoded so that it can be transferred by the Omnix system. There is no restriction on the content of the binary data. For instance, it could be a serialized Java object.

Separating the routing from the content

The main disadvantage of this design is that there is not a strict separation between the P2P networking layer and the application. It is obvious that there must be some binding between them. If, for instance, an application wants to inject some information about a service offered, a distributed hashtable topology module must have access to the description (because it needs a key based on the content that can be used for the hashtable).

Hence, there is a possible dependency between the topology module and the application on top of it. To face this problem, Omnix provides default structures that may be used for most use cases. These structures can be used in a very flexible way.

Designing an interface that would meet the aggregate requirements of all possible P2P applications could be possible, but the increased complexity of the result would render the complete middleware less usable (and less attractive). We decided to provide an open interface and, to keep the layered structure to some extent, a default implementation that suffices for most applications.

Section 8.2.3 shows that the methods introduced in the Omnix API are sufficient for all P2P topology classes (see Section 2.3) and use cases (Section 2.2.2).

4.3 Streaming

Identifying communication classes in a P2P network

Typically, in P2P networks three classes of message exchanges exist:

- *Maintenance:* To keep up the structure of the P2P network, it might be necessary to communicate with other peers in the network. The most common example is the sending of periodic message to other peers in the (network) neighborhood to verify that they are still reachable. If a remote peer is no longer available, it has to search for another peer that it can connect to. The complexity of this task depends on the topology chosen. Another example is that peers have to continuously search for other peers to complete (or improve) their routing tables.

- *Search related:* This kind of message is used for searching content or services in the P2P network (no surprise here). There are typically two message types: requests and responses. The request contains a (maybe incomplete) description of the desired information (e.g., some pattern that can be matched with meta-data). The response contains a list of descriptions of artifacts (e.g., files, services, names, etc.) that match the search specification. Note that in most cases this kind of message does not contain the actual information itself but meta-information about where to find it. This is important for the searching peer. So, it can browse through the description data instead of downloading all information directly (which could be painstakingly slow) before selecting the exact match.

- *Streaming:* Once the desired information is located, it has to be downloaded or accessed (in the case of services). While the first two classes of communication typically consist of small packets (i.e., they are *message-oriented*), downloading files (or in general: any kind of bi-directional communication with large amounts of data) calls for streaming (small amounts of data can, of course, be transmitted in a message-oriented way). When streaming, not single, small packets are sent to the remote peer but a channel is opened to transmit continuous data with increased throughput and reliability. An example for a stream is the TCP/IP protocol which creates a connection-oriented connection to another host.

The JXTA core implementation is completely message-oriented (i.e., it does not provide streams). There exist additional projects like *p2psockets* [83] that build this feature into JXTA.

Designing an open streaming architecture

The most simplistic solution would be to let the application on top of Omnix deal with the creation and use of streams (e.g. by using a TCP stream). This way, Omnix would only be responsible for locating the data without any means to acquire the data. This has the advantage that the application would have full control over the streams which would reduce the complexity of the interface between the application and Omnix.

Unfortunately, this is not possible. The first reason for this is the possibility of firewalls between the server and client peer. A firewall might not allow incoming connections to the server peer. Another problem in this context are NAT (Network Access Translator) configurations. The solution would be to contact the server peer and ask it to create a stream to the client peer. But it does not make sense to let the application programmer create this functionality while Omnix could do this as well. The second reason why it is not desirable that an application on top of Omnix opens a channel itself is: *platform independence*. While it is possible to open a channel (e.g., a TCP/IP connection) on a normal PC or a handheld device, it is impossible on a Bluetooth device or a mobile phone without changing the code of the application.

Hence, there is a need for an abstraction between connection-oriented streams and the underlying network protocols. This is achieved by the *StreamManager* (Figure 4.12).

```
public interface StreamManager {

    public String requestStream(Contact remotePeer, String context,
                                StreamReceptor receiver);

    public void addStreamReceptor(StreamReceptor sr);
    public void removeStreamReceptor(StreamReceptor sr);
}
```

Figure 4.12: The StreamManager interface.

Establishing a stream

In the following, we describe the interactions of the various components of the open streaming architecture for opening a stream to a remote peer.

If the StreamManager of a peer detects that a remote peer wants to create a stream, it iterates through the list of so-called *StreamReceptor*s (Figure 4.13; typically provided by the application), asking each to analyze whether it is willing to accept the incoming stream request. Information about the stream, provided by the connecting peer, helps the Stream-Receptors to evaluate the relevance of the requested stream (e.g., by providing the name of the file that should be uploaded).

```
public interface StreamReceptor {

    public boolean analyzeStream(Contact c, String context);
    public void takeoverStream(Contact c, String context,
                               StreamControl sc);
}
```

Figure 4.13: The StreamReceptor interface.

Once a StreamReceptor has accepted to take over the stream, the StreamManager opens the stream and passes it to the StreamReceptor in form of a *StreamControl* (Figure 4.14). Note that it is necessary to use a generic StreamControl component that encapsulates the networking aspects of streaming to ensure true platform independence (e.g., it would not be possible to pass a `Socket` object to the StreamReceptor because a J2SE–`Socket` does not exist on a Java-enabled mobile phone). A StreamControl contains an identifier and two streams, one for each of the two directions the communication might take place. If no StreamReceptor is willing to take over the stream, the StreamManager returns an error message to the requesting peer.

```
public interface StreamControl {

    public InputStream in();
    public OutputStream out();

    public String streamID();
    public void close();
}
```

Figure 4.14: The StreamControl interface.

If an application wants to open a stream to a remote peer, it instructs the StreamManager to do so and registers as a *StreamRequestor* (Figure 4.15). The StreamRequestor is similar to the StreamReceptor. The difference is that the StreamRequestor cannot, once it has requested the StreamManager to open a stream, refuse to accept the established communication link.

```
public interface StreamRequestor {
    public void takeoverStream(StreamControl sc);
}
```

Figure 4.15: The StreamRequestor interface.

Providing a network- and protocol-independent abstraction of streams

To provide complete platform-independence, it is not possible for the StreamManager to directly use the network primitives provided by the operating system or virtual machine. As in the case of transport modules, it is necessary to have an abstraction between the Stream-Manager and the underlying network. This abstraction is achieved by the use of so-called *StreamProvider* components (Figure 4.16). A StreamProvider provides all functions necessary to create a connection-oriented communication link with another peer. For each type of network, a different StreamProvider might be used. For example, StreamProvider components could be used for the TCP/IP protocol, some proprietary Bluetooth protocol, or for anything else that can provide streaming. For the StreamManager, it does not make a difference how the communication is achieved in the StreamProvider (i.e., the interface is always the same) as long as the communication takes place.

Providing tailored streams

A StreamProvider, in conjunction with StreamControls, may not only be seen as an abstraction for using sockets (and the like). It is rather a possibility to tailor streams to the requirements of the application. If streaming, for example, is used for Voice over IP (VoIP), the StreamControl object may be an implementation of the Real-Time Transport Protocol

```
public interface StreamProvider {
    public Contact createServerSocket(String streamID,
                                      StreamRequestor sr);
    public StreamControl connectToServer(String streamID, Contact c);
}
```

Figure 4.16: The StreamProvider interface.

(RTP, [84]), which in fact runs on top of UDP. Hence, the stream is not necessarily a normal stream (in the sense of TCP, for example) but can be anything that transmits data. Depending on the use case, different StreamProvider might be used. For downloading a file, a StreamProvider with a reliable connection is needed. When streaming real-time audio or video, it is more important that the StreamProvider can guarantee that the stream is continuous. Supporting streaming while on the move or QoS (Quality-of-Service) are additional features that are also possible.

The central part of streaming is the *StreamManager* itself. It is the central controlling component that manages the registered StreamProvider and StreamReceptors. It is also responsible for establishing a stream-connection across firewalls and NAT configurations.

Establishing a connection to a peer behind a firewall

When a peer wants to create a stream (e.g., a TCP connection) to a remote peer, it might be the case that the firewall installed at the remote peer does not allow incoming TCP connections. In this case, the stream has to be opened in the other direction. To support this feature, the StreamManager must allow two ways of opening a stream:

- *As a server:* (Figure 4.17) The StreamManager instructs a StreamProvider to accept incoming connections and sends a message containing a connection request to the remote peer's StreamManager. The remote StreamManager iterates through the list of StreamReceptors the application has installed and picks the first one that accepts the stream. The resulting StreamControl (i.e., a representation of the actual established stream), which is returned by the StreamProvider, is handed to the StreamReceptor (i.e., the application). The initiating StreamManager gets a corresponding StreamControl object from the StreamProvider, which is passed to the StreamRequestor (i.e., the application). Now, the applications on both sides have full control over the stream (as StreamRequestor and StreamReceptor).

- *As a client:* (Figure 4.18) It might be the case that a peer is not able to accept incoming connections (e.g., because a firewall blocks incoming TCP connections). In this case, the StreamManager can also be configured to try to connect the other way round. When the application wants to create a connection to another peer, the StreamManager sends a message (again, via the Core and a transport module) to the remote peer. This message requests the remote peer to accept a connection. The remote peer searches for a StreamReceptor willing to take over and, if successful, brings the StreamProvider to wait for an incoming connection (e.g., the remote StreamProvider opens a server

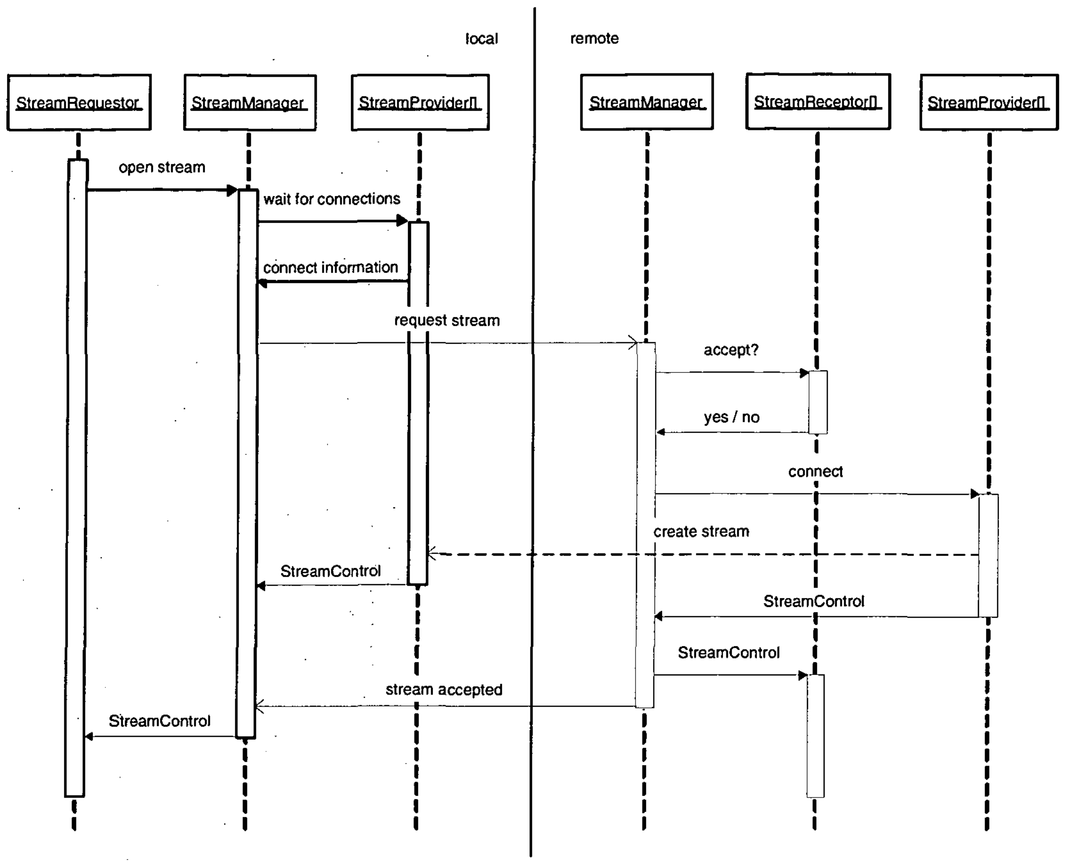


Figure 4.17: Opening a stream as a server.

socket and waits for incoming connections). Having done this, the remote peer sends back a response to the local peer. This response must include information about how and where to access the remote peer's StreamProvider (e.g., the port number). After receiving the response, the StreamManager instructs its StreamProvider to open a streamed connection to the remote StreamProvider and returns the corresponding StreamControl object. The StreamManager forwards this StreamControl to the initial StreamRequestor.

By default, the StreamManager first tries to create a streaming connection as a server. If this does not succeed (e.g., because the remote peer accepted the connection but no connection has been established – possible due to a firewall), the StreamManager automatically makes a second attempt in the other direction. Hence, the StreamManager provides a simple service for dealing with firewalls and NAT configurations.

However, this is only a best effort service. If both peers are not able to accept incoming streamed connections, it is impossible with this implementation of the StreamManager to create a working communication link. However, the flexibility of Omnix allows the StreamManager to be replaced by another implementation. Another StreamManager, for instance, could use a central server to connect two peers behind firewalls. Both peers connect to the

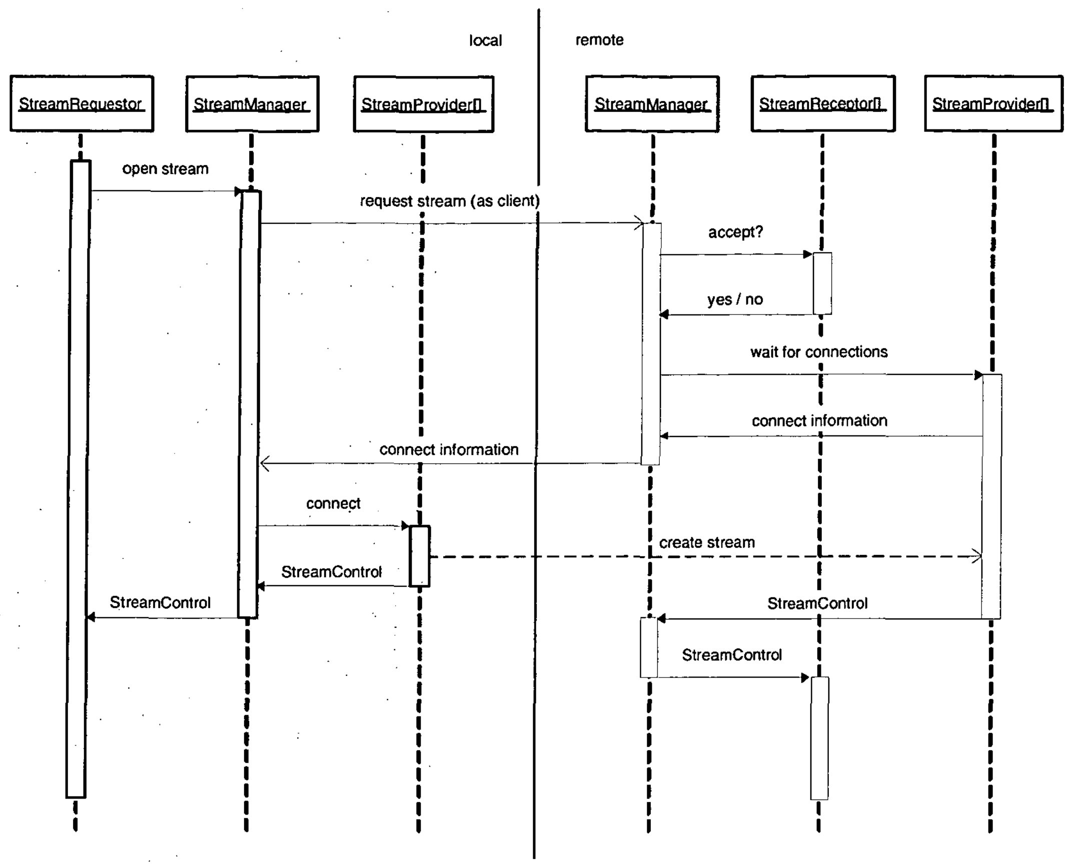


Figure 4.18: Opening a stream as a client.

same server, which would then be used as a proxy, forwarding the data streamed to the server.

Load balancing

Furthermore, the StreamManager does not necessarily have to be on the same machine as the StreamProvider. Since StreamManager and StreamProvider are replaceable, they could themselves use the network for communication. This would have the advantage that the StreamManager could do, for example, load-balancing among the available StreamProviders. This could be useful in P2P networks with central servers.

In this chapter, we introduced Omnix, a middleware framework for P2P networks. By using a layered architecture, Omnix is able to adapt itself to the requirements of the application. By using plugins it can be customized to run on various devices, use arbitrary network protocols or employ any network topology that seems fit. What has not been covered by this chapter is the connecting link between two peers, the protocol. The design for an open P2P protocol is presented in the next chapter.

Chapter 5

Omnix Protocol

If everyone on campus turned off the outbound KaZaA traffic, approximately 50% more bandwidth could be freed for other Internet traffic.
Jintae Lee [85]

The protocol plays an important role in a P2P network. It specifies how information is exchanged between two end-points. Note that it does not (necessarily) define the topology of the network. Depending on the purpose of the network, the protocol must support distributed searches (e.g., for peers, artifacts, users, etc.), direct communication between two peers, accessing files and services, etc.

5.1 Requirements for a P2P protocol

In the course of designing Omnix we identified the following requirements as being vital for an open P2P protocol:

- *Topology independent:* Since the purpose of the Omnix P2P framework is not determinable, the protocol must be independent of the use case and the topology. Hence, the protocol must only provide means for connecting two peers. It must not rely on the existence of a network service infrastructure (such as DNS). This way, restrictions on the overall topology of the network are impossible.
- *Arbitrary content:* The protocol must be able to transport any type of content. It should not impose any restrictions on the payload of the protocol (e.g., no binary data).
- *Reliable:* It cannot be guaranteed that the underlying transport protocol is reliable. While TCP is fairly reliable, for example, this assumption cannot be made when using UDP, infrared communication, Bluetooth, etc. Reliability can be achieved by sending acknowledgment-message on incoming messages. This way, a sending party can resend a message if no corresponding acknowledgment has been received. The protocol must support acknowledgments to messages and ways to match messages and their corresponding acknowledgments.

- *Transport protocol independent:* The P2P network protocol must be independent of the underlying transport protocol used (e.g., TCP, UDP, Bluetooth, etc.). This means that the protocol must not make any assumptions on the type of transport network (e.g., connection-oriented vs. connectionless). Furthermore, the addressing scheme must be independent of the underlying transport protocol. For instance, it would not make sense to use IP address/port pairs for connecting two peers over an infrared communication link.

- *Small messages:* It is necessary that the protocol is not too "verbose". It is a requirement that messages sent over an Omnix P2P network are as small as possible. This is necessary because there cannot be any assumptions made about the capabilities of a peer. If a device has only a limited memory capacity, it may not be able to process large messages. Furthermore, the protocol must not rely on the existence of streams. If streams are needed (e.g., for the exchange of large chunks of data), a *StreamManager* should be used.

- *Secure:* It is easy to say that the protocol must be secure. But this does not specify what security means in this context. There are a lot of security aspects to be covered: *secrecy* – or *confidentiality* – (has to do with keeping information out of hands of unauthorized users), *authentication* (deals with determining whom you are talking to before revealing sensitive information), *non-repudiation* (how can you prove that somebody has "really" sent a message although she denies it), and *integrity control* (makes sure that a message has not been altered between sender and receiver). The protocol must support all four security aspects. This sounds more sophisticated than it is. The major work is done by the peers. The protocol just has to support these aspects (e.g., by allowing encryption of parts of the message).

- *Firewall/NAT savvy:* Firewalls and NATs are major problems in Internet-scale P2P networks. The protocol must support the traversing of firewalls and NAT boxes. There exist three ways how this can be achieved. 1) by using ports that are typically open for incoming requests (e.g., the HTTP port 80), 2) by using connection-oriented, persistent connections to the outside (which are used by remote hosts to connect to the local host behind the firewall), and 3) by using a proxy that is used to traverse a firewall or NAT box. The first two ways are not very "system-administrator friendly". They force an administrator to parse HTTP connections and to disallow outgoing connections. Furthermore, there is no guarantee that the first two ways always work (e.g., what if two communication partners are behind a firewall?). The third solution has the advantage that it would be in accordance with the system administrator (she is the one who has to install the proxy peer) and reliable. The bottom line is, that the protocol must support the use of proxy peers that forward messages to the original destination peer.

- *Simple structure:* The structure of a message must not have a high complexity so that devices with limited capacity are also able to process it. This has the effect that no high-level message structures like XML may be used because they typically require special libraries for parsing and creating such structures. On a mobile phone, for instance, it is quite cumbersome to parse XML messages. Hence, simplicity is one of the requirements for a protocol in Omnix.

- *Multi-hops:* In some topologies (e.g., distributed hashtables), a message needs multiple hops (i.e., it is forwarded by peers until it reaches its final destination). If this is the case, the response may be sent back either directly to the initiating host or by using the same route in the reverse direction. The protocol must support both ways. The latter could be provided by recording a list of visited hosts on the way from the sender to the receiver.

Omnix can be used in conjunction with any protocol, such as (among others) SOAP, CORBA, or RMI. Nevertheless, to meet the aforementioned requirements, we designed a flexible protocol that can be easily extended. It has similarities with the HTTP [86] and SIP [87] protocols. The following sections will show how the protocol works in general, explain the structure of the messages, and discuss how messages are processed.

5.2 Overview

The Omnix protocol mainly deals with the communication between two peers. The sending peer sends a request to the remote peer, which in turn sends back a response, providing information about the processing of the request. There are no semantics associated with a request. It may be a search request, an instant message, the current time, etc. The response may be a status report, a requested artifact, etc. For every request received, a peer must send back a response.

A peer is required to send back a response after processing the request. If, within a certain time interval, no response has been received, the sending peer sends the request again. If the processing of a request takes a longer period of time, the processing peer should send back a *provisional response*, indicating that the request has been received successfully and a final response will be sent later.

Messages can also be forwarded to other peers (a scenario not very unlikely in a P2P network). There may be several reasons for a peer to do this. Search requests are typically forwarded to other peers that may have the desired information. It is also possible that a proxy peer is used to forward messages to peers behind a firewall. If a peer forwards a message, it adds its address to the top of the list of visited hops (i.e., peers) contained in the forwarded message.

Responses take the same route as the request, but in reversed order. This has many advantages: if a proxy has been used to bypass a firewall, the same proxy can be used to bypass it in the other direction. Furthermore, peers on the route may cache the information contained in the response for future requests. If a peer receives a response, it checks whether its address is on top of the list. If this is the case, it removes its address from the message and forwards the response to the next hop (i.e., previous hop for the corresponding request), until the response has reached the peer that has initiated the request (i.e., when the list of visited hops contains the sender's address only). If this address is not correct, the peer may discard the message.

Figure 5.1 shows examples of how messages could be sent through a P2P network.

Peers may also send a "Redirect" response (Figure 5.1–b), telling the initiating peer to instead contact the peer specified in the response. This could be useful, for instance, in

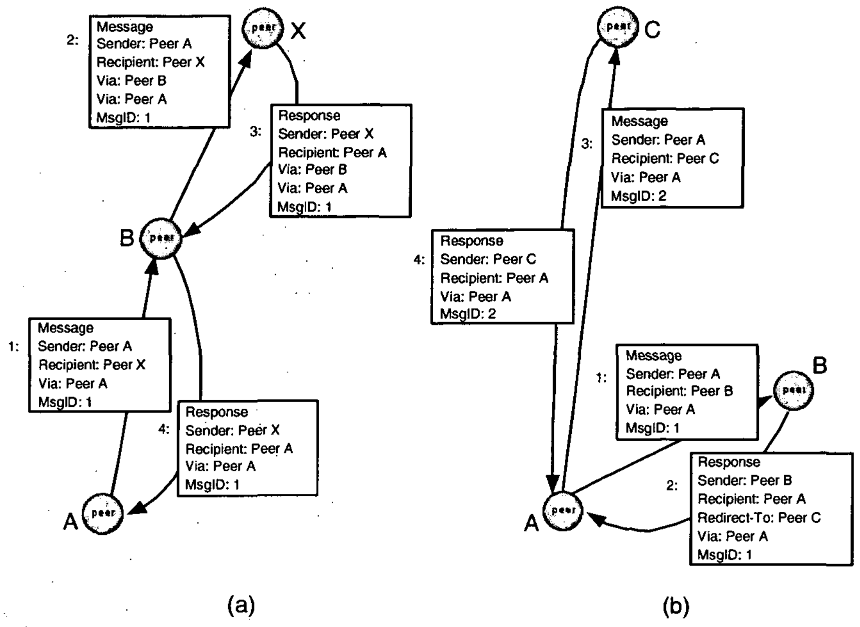


Figure 5.1: Examples of message flows in Omnix.

distributed hashtables where a peer does not want to forward messages but rather redirect the client to the next peer.

A request may contain content (i.e., the payload of the message). This content is application-specific and has no relevance for the Omnix protocol. It may be any binary data, including normal text.

5.3 Message types and structure

This section deals with the messages used in the Omnix protocol. Omnix has only two message types: *Requests* and *Responses*. Unlike other protocols that specify message types for every class of message exchanged between peers, Omnix uses a single pair of messages, which can be used arbitrarily. They are the building blocks for higher-level communication flows.

The Omnix protocol is text-based. This is necessary because simple peers may not be able to parse complex structures such as XML. The format of the messages can be compared with the Session Initiation Protocol (SIP, [87]) and the Internet Message Format (see [88]). Both message types, *Requests* and *Responses* have a single header line, multiple header fields, an empty line signaling the end of the header fields, and a content (Figure 5.2 shows a simplified version of the message structure). In this dissertation, message structures are described using the Augmented Backus-Naur Form (ABNF, [89]). Appendix B gives an overview of the reference message structure in Omnix.

```
message    =    requestline / responseline
                *messageheader
                CRLF
                [ messagebody ]
```

Figure 5.2: The general structure of Omnix messages.

5.3.1 Request / Response

The semantics of a request (e.g., whether it is a search request, an instant message, the invitation for a game of chess, etc.) is specified by the *requestline* (Figure 5.2). It is defined by the sending component of a peer, which could be a processing module, a topology module, or the application itself.

The requestline consists of a method, an URI, the protocol used, and the version of the protocol (Figure 5.3).

```
requestline    =   method SP requestURI SP
                   protocol "/" version CRLF
```

Figure 5.3: The request line structure.

The *method* of a request indicates the purpose of the request. Omnix does not have any restrictions on the method. It may be any string that contains no white-spaces. What methods are used is only defined by those modules that want to exchange information with other peers. The *requestURI* in the request is also application (or module) specific. Omnix only provides the space for such an URI, but does not define how this space should be used by the application. Typically, the address of the destination peer (or the name of a service at a remote peer) is specified in this field. The *protocol*, which is also a single text, is also arbitrary. It may be the case that a single application wants to use more than one protocol for communicating with remote peers. The *version* field is used for determining the version of the protocol used. If a module receives a request, it must make sure that it understands both the protocol and the version. If this is not the case, the module must not process the message. Figure 5.4 shows an example for a request line. In this example, the message is a search request for a file. The protocol's name is "Omnix", the version is 1.0.

```
SEARCH file OMNIX/1.0
```

Figure 5.4: An example for a request line.

A *responseline* has the structure shown in Figure 5.5. It contains the protocol and version identifier used in the corresponding request. Furthermore, a *statuscode* indicates the outcome of the processing of the request. As in the HTTP protocol, the statuscode is a three-digit number. There exist classes of statuscodes (identified by the first digit of the code). Appendix A provides a short overview of the different classes of statuscodes.

```
responseline  =  protocol "/" version SP
                 statuscode SP reasonphrase CRLF
```

Figure 5.5: The response line structure.

In addition to the status code, a response also contains a so-called *reason phrase*, which is a plain-text, human-readable translation of the status-code. This text could be used either for making debugging easier or to display it to the user of the application. Figure 5.6 shows an example for a *responseline*.

```
OMNIX/1.0 400 Not found
```

Figure 5.6: An example for a response line.

Appendix A shows all status codes and accompanying reason phrases currently defined in the Omnix protocol.

5.3.2 Header fields

A message can contain multiple header fields. Header fields hold information about the sending peer, the target peer, the content of the message (e.g., encoding, MIME-type, etc.), the length of the message body, and much more. Header fields are similar to the header fields defined in HTTP or SIP, although some small differences exist (see below).

While Omnix defines only a small number of headers that have to be present in a valid Omnix message, applications (and modules) may freely introduce new header fields. For instance, a topology module may add information like importance, duration of validity, etc. to a message.

A header field is a key/value pair, separated by a colon. Header field values may contain ISO 10646, UTF-8 [90] encoded characters. It is possible to use binary data in header fields (which may prove useful if a header field contains, for example, a binary encryption key).

```
messageheader  =  headername COLON *WSP headervalue CRLF
```

Figure 5.7: Structure of a header line.

As shown in Figure 5.7, Omnix has some limitations compared to SIP or the Internet Message Format. In Omnix, all information in a header field value must be encoded in a single line. It is not allowed to use a CRLF element within a header field value. Furthermore, the *headername* must immediately be followed by a colon, instead of allowing an arbitrary amount of white spaces in front of the colon. After the colon, white spaces may be used.

Typically, the ordering of headers is not important. There is one notable exception: the *Via* header. The Via header is used to record the route from the emitting peer to the target peer. This route is necessary so that the response can take the way back that the request came along before. At every hop (starting with the sending peer), a Via header is added on top of the list of header fields. This information is then copied into the response and used for finding the way back.

Figure 5.8 shows an example for typical header fields in a message. The message shows that the peer at the address `195.34.150.72` has sent a message to `128.131.172.113`. This message has been routed through a proxy (at `128.131.172.1`).

```
To: 128.131.172.113:10000
From: 195.34.150.72:3602
Via: 128.131.172.1:3602
Via: 195.34.150.72:3602
Content-Length: 2323
MsgID: 3985
```

Figure 5.8: Example of header lines in a Omnix message.

It is also possible to encrypt parts of the message headers (so that only the final receiver is able to decipher the message). It is obvious that some header fields must remain plain text so that forwarding peers in the middle of the transfer are able to forward the message correctly. These header fields are: *To*, *From*, *Content-Length*, all *Via* headers, and the *MsgID*. All other header fields may be encrypted by the sending peer.

There is a distinction between required and optional header fields. Required header fields are those that are necessary to successfully deliver a message at the target peer. If a message does not provide the required header field, a peer may drop the message. Optional header fields are those that may be freely added to the message. Required header fields are *To*, *From*, *Via*, and *MsgID*.

5.3.3 Body

An Omnix message may contain arbitrary binary content, which is defined either by the application, a topology module, or a processing module. The content is placed below the header fields, separated by an empty line (which must be in place even if no content is contained in the message).

If a message contains content, it must specify at least the size of the content, by using the *Content-Length* header field. This field is necessary because the transport module that receives the message does not know how large the content may be (especially if the message is transferred over a connection-oriented communication link such as TCP). In addition to the length, optional header fields like *Content-Type*, *Content-Encoding*, or *Content-Language* may be used.

Unlike in HTTP, in Omnix both request and responses may contain content. There is, however, a limitation of the size of the content. Since it may be the case that messages are transported with small-sized packets (e.g., UDP), the size of the content must be as small as possible. The maximum size of the message is defined by the transport modules.

5.4 Security

This section discusses how the four security aspects defined on page 68 can be achieved with the Omnix protocol.

- *Secrecy:* Secrecy, or confidentiality, can be achieved by encrypting parts of or the whole message, so that it can only be read by the target peer. Encrypting the complete message is problematic because it makes it impossible for peers in the middle (e.g., proxies) to forward the message to the destination peer. Hence, Omnix allows the encryption of parts of a message by specifying the header fields that must remain plain-text. The other header fields may be encrypted by any means chosen by the application, topology module, or processing module. There is no restriction on the encryption method employed.

  Note that it is also possible to encrypt messages between two hops along the route of the message. Hence, if a message is sent in plain-text through a proxy, the proxy may encrypt the message before it is forwarded to the next hop, where it may be decrypted and forwarded as plain-text again.

  As stated in Section 5.3.2, *Via* fields must not be encrypted. Hence, the route of a message is recorded in plain-text in the message. To hide the route of a message from snoopy peers, a hop in the middle (e.g., a proxy) may remove the *Via* headers (a process called Via-hiding). This is only possible if this peer saves the removed list of *Via* headers. When a response is received (which takes the same route as the request, but in the reverse direction), the peer must restore the saved list in the response, so that the message finds its way to the originator of the request. This method implies that the proxying peer must be stateful (instead of a state-less peer that only forwards messages regardless of previous messages).

  If a peer wants to encrypt the complete channel to another peer (i.e., everything – including header-lines and required header fields – sent over the wire is encrypted, such as through a SSH-tunnel), this can be implemented by the transport modules. There is no need for Omnix to provide support for this type of encryption.

- *Authentication:* There are many ways how authentication can be achieved in Omnix. Either by using a shared secret or with the use of public-key encryption. It is not the scope of this document to discuss the various methods of achieving authentication. Authentication schemes like those used in the HTTP protocol (i.e., the *basic* or the *digest* authentication scheme) may be used in the same way in Omnix. However, it is important to note that in Omnix both parties may require authentication (as opposed to the HTTP protocol, where only the client must authenticate itself).

If a peer receives a message and requires authentication, it sends a message with status code 401 (*Unauthorized*). Upon receiving this message, the sender must resend the initial request with authentication credentials.

- *Non-repudiation:* It is hard, if not impossible, to assure non-repudiation. One attempt to achieve non-repudiation is using digital signatures with a public-key infrastructure. This way, only the holder of the private key can produce the digital signature. Is this enough? Unfortunately, not. It may be the case that only the owner of the private key is able to sign a message. But it cannot be proven that the owner of the private key was the only one who had the private key at the time when the message has been signed. Hence, a peer may sign a message and afterward claim that the private key was already compromised at that time. Hence, non-repudiation cannot be achieved in Omnix. Still, it is possible to use digital signatures to achieve a certain level of non-repudiation.
- *Integrity control:* A typical way of providing integrity control is to use digital signatures. In most cases, they are a hashsum, encrypted with the private key of the creator of the message. The receiver is then able to create the same hashsum (if the message has not been modified) and verify it by decrypting the hashsum enclosed in the message. Signatures can be embedded in an Omnix message by using multipart MIME contents in the message body (more on this can be found in [91]).

Omnix itself does not provide a public-key infrastructure (e.g., a Certification Authority). If an application needs such a feature, it may implement processing modules that provide that functionality.

Except for non-repudiation, every aspect described in the requirements are possible in the Omnix protocol. If there is a way of providing non-repudiation, it may be easily added as a processing module to the Omnix P2P framework.

5.5 Protocol Alternatives

It is legitimate to ask why Omnix uses a new protocol instead of applying one of the existing protocols. There are so many P2P systems out there. Is there not a single one that can be used for Omnix? The answer is: not really. This section discusses this topic and shows why existing protocols (a few examples are given) are not suitable for the requirements listed on page 61.

The Gnutella protocol (version 0.4) [92], for example, has a fixed structure. Depending of the type [1] of the message, different message structures are used, each having a fixed set of fields. For most fields, the length is also exactly specified. It is not designed to support the addition of new information to a Gnutella message. It would be possible to add information to certain messages (e.g., the *Query* message), but this would mean that this information has to be encoded in the search string. Hence, although possible, the Gnutella protocol does not fit the requirements of Omnix.

The Napster protocol [93] has the same restrictions as the Gnutella protocol. There exists only a limited number of message types and those messages have a fixed structure and

[1]Gnutella 0.4 only knows five message types: Ping, Pong, Query, QueryHit, and Push.

partially a fixed length. It is further based on a client/server architecture and does not support concepts like forwarding messages to other peers (including loop detection and the like). For obvious reasons, it is completely unsuitable for Omnix.

Also not usable is the Freenet protocol [18], which provides strong security mechanisms. These mechanisms are needed to provide user anonymity. This has a very high price. Searches can only be performed on IDs, not on arbitrary meta-data defined by topology modules or applications.

P2P networks that act as distributed hashtables may have no restrictions on the structure or the content. The main problem of such protocols is that they are designed for a fixed topology (i.e., that of a distributed hashtable, however it may be organized). They use special routing algorithms with routing tables and have proprietary ways for maintaining these routing tables and to discover other peers. Furthermore, routing is based on the usage of numbers for searching. Hence, full meta-data search is not possible in such systems.

This list could be easily extended by other P2P protocols with similar problems. The bottom line is that P2P protocols are tailored to their use. The only notable exception of a P2P protocol, that 1) is not closed and 2) can be used as a generic P2P protocol is JXTA. Unfortunately, for two reasons even JXTA is not suitable. On one hand, JXTA has a more-or-less fixed topology, using so-called *PeerGroups* to divide the P2P network. Hence, JXTA is not fully topology-independent. On the other hand, JXTA uses XML, which has two implications:

- XML is known to be very "verbose". The generic, hierarchical structure of XML makes it necessary to use additional space and information for organizing this structure. Hence, the messages would need more bandwidth, processing power and memory – resources that may not be extensively available on small devices.

- To parse an XML structure, additional libraries are needed. Typically, these libraries have a considerable footprint, which hinders their use in small devices with limited capacities. There do exist some XML parsers with a relatively small footprint (e.g., XMLtp [94] or MinML [95], which has a size of approx. 9 KB). These small parsers typically ignore the validation of XML documents using DTDs or Schemas. While this may be ignored in normal applications, this is a shortcoming when XML documents are received from remote peers (where the content should be validated before it is processed).

After eliminating the possibility of using existing P2P protocols, the next question is whether any generic communication systems (i.e., middleware) could be used. It is obvious that middleware systems like Java RMI oder DCOM are not suitable for Omnix because they do not provide platform independence. Reasonable alternatives would be open protocols like SOAP [96], XML RPC [97] or CORBA. The disadvantage that they have is that they are too heavy-weight for small devices (e.g., such as a mobile phone or an embedded device running Java), mostly due to their extensive use of XML (with the exception of CORBA, of course, which nevertheless has a lot of communication overhead, too).

Anyway, the Omnix P2P framework allows the replacement of the protocol used. An application programmer (or a provider of third-party components for Omnix) may use any protocol that seems fit. The protocol implementation provided with Omnix should be used because it is designed to meet all necessary requirements (and is most likely to be used by other applications).

The next chapter provides information about the programming aspects of Omnix. It demonstrates how various modules can be written to change the behavior and capabilities of the middleware framework. It also discusses special topics like Quality of Service (QoS), Firewalls, and the testing of applications using Omnix.

Chapter 6

Programming Aspects of Omnix

Never be afraid to try something new. Remember, amateurs built the ark; professionals built the Titanic.
Anonymous

When an application programmer wants to use Omnix as the underlying P2P communication infrastructure, she has to find an optimal combination of the various components that can be used with Omnix. Depending on the use case the most important decision is which topology module to pick for the P2P network. Since it is very cumbersome to change a topology module once it is installed on hundreds (or more) nodes, it is beneficial to choose the best topology module before the application is shipped. In order to find the right decision, the programmer could use the testing facilities provided by Omnix (see Section 6.4). With this P2P simulator, different topologies (and processing modules) could be tested and an optimal system configuration found.

This chapter shows the versatility of Omnix by demonstrating how it can be used with various topologies. Examples show how components can be written with a few lines of code. Furthermore, advanced issues and the application of Omnix in a pervasive environment are discussed. Eventually, a case study is presented to show how Omnix contributes to the problem of programming topology-independent P2P networks efficiently.

6.1 Changing the lower layers of Omnix: Transport and Processing

One of the major goals of Omnix was not to create YAPS ^{TM}(*Yet Another P2P System*) but to create a framework that would allow application programmers to change every aspect of a P2P system. Those aspects include the structure and ontology of messages, the routing algorithm, security issues (e.g., encryption, digital signatures, etc.), supported platforms, and much more. Hence, one of the requirements of Omnix was to make it a framework for a collection of exchangeable components.

The following sections show, on the basis of simple examples, how the modules in the transport layer and in the processing layer can change the functionality of the Omnix system.

6.1.1 Transport

While the purpose and functionality of the transport modules is explained in Section 4.2.2.1, this section deals with the practical advantages of using replaceable components for the network communication by showing implementation examples.

```
1  public interface Transport extends Runnable {
2
3      public void addInputMessageListener(MessageListener ml);
4      public boolean removeInputMessageListener(MessageListener ml);
5
6      public boolean sendMessage(Message msg);
7
8      public void start();
9      public void stop();
10 }
```

Figure 6.1: The Transport module interface.

Taking a closer look at the Transport interface

Figure 6.1 shows the most important features of the *Transport* interface defined in Omnix (which extends the `Runnable` interface so that every transport module is running in a separate thread, thus not blocking other parts of the middleware or the application).

The methods shown in lines 3 and 4 are necessary to allow other components to register (or to unregister) as a *MessageListener*. It is possible to register multiple MessageListener. Whenever a message has been received by a transport module, it iterates through the list of registered MessageListener and passes the message to every one of these. One MessageListener that will probably be registered is the *Core*.

The *sendMessage* is called – the name gives it away – when the transport module is supposed to send a message over the network. The *start()* and *stop()* methods are used to tell the transport module when to listen for messages and when to shutdown the network connection.

Implementing a Transport module with a few lines of code

In the following, an example transport module will be presented, which uses datagram socket communication to exchange information with other peers. Figure 6.2 shows how the example transport module is initialized. Note that it uses a *MulticastSocket* object so that it can also be used to transmit and receive multicast or broadcast messages.

The next step is to allow the Core (or any other component that has access to this transport module) to send a message to a remote peer. For this purpose, the transport module must implement the *sendMessage* method. This method just takes the message object, which contains all the information necessary to send the information (see Figure 6.3). The *getPayload* method of the message object is responsible for converting the actual message into a

```
public class MulticastTransport implements Transport {

    public MulticastTransport() {
        multicastSocket = new MulticastSocket();
    }
}
```

Figure 6.2: Initializing a transport module.

binary array that can be sent over the network (line 3). The transport module does not (and should not) have anything to know about the structure of the message. Hence, it is possible to replace the structure and ontology of the message in the layers above without changing the code in the transport layer.

```
1  public boolean sendMessage(Message msg) {
2
3      byte[] payload = msg.getPayload();
4
5      InetAddress toaddr = InetAddress.getByName(msg.getTo());
6      DatagramPacket dp = new DatagramPacket(payload, 0, payload.length,
7                                              toaddr, DEFAULT_PORT);
8
9      return multicastSocket.send(dp);
10 }
```

Figure 6.3: Sending a message in the UDP transport module.

The next two lines (5 to 7) retrieve the destination address from the message object and create the datagram packet (in this simplified scenario, the default port is used instead of retrieving the port information from the destination address as well). Finally the message is sent to the target peer. It does not make a difference whether the message was a request or a response, an Omnix message or a message with a modified structure, etc.

Every transport module is run in a separate thread (this is the reason why the receiving of messages is implemented in the *run()* method specified by the `Runnable` interface). The *receive* Method (line 6) blocks until a message is received. Once a message has been received, it is parsed by a *MessageParser* and put into a *Message object.* After that, the transport module passes the message to all subscribed *MessageListener*s.

What is left is to start and stop the transport module. To start the module, it has to create a new thread and start itself. [1] Stopping the thread is done by setting the variable *runLoop* `false` (as shown in Figure 6.5). This way, the *run()* method stops and the transport module ceases to accept incoming packets.

These are the main parts necessary to create a transport module. The separation of concerns in Omnix requires a transport module only to deal with the networking aspects (send-

[1]A design alternative would have been to start the transport module from the outside but the decision was made this way to increase the flexibility of the transport module.

```java
public void run() {

    while(runLoop) {

        DatagramPacket dp = new DatagramPacket(buf, buf.length);
        multicastSocket.receive(dp);

        Message msg = MessageParser.parse(dp.getData());
        msg.setTransport(this);

        for (int i=0; i<listeners.size(); ++i) {
            MessageListener ml = listeners.get(i);
            ml.receiveMessage(msg);
        }
    }
}
```

Figure 6.4: Receiving a message in the UDP transport module.

```java
public void start() {
    Thread t = new Thread(this);
    t.start();
}
public void stop() {
    runLoop = false;
}
```

Figure 6.5: Starting and stopping the UDP transport module.

ing and receiving of messages). Everything else is done in the layers above. The *Transport* module interface allows the seamless exchange of transport modules.

6.1.2 Message processing

The processing modules, which reside in the Processing Layer of the Omnix framework, are necessary to deal with incoming messages and to adapt outgoing messages. Processing modules are arranged in pipelines. Omnix uses pipelines for incoming and outgoing messages. When a message is put into the pipeline, every module in the pipeline gets a chance to process the message (the list of possible actions in given in Section 4.2.3.2, page 48). In this section, a few examples of how a processing module can be implemented to change the functionality of the P2P system are given.

Providing a default implementation for processing modules

Figure 6.6 shows the *Module* interface (which, in fact, is an abstract class) that all processing modules have to adhere to. An abstract class is used instead of a code-less interface to minimize the effort for creating new modules. This way, only the required code has to be inserted without having to care about all implementation details.

The *initialize()* method is used to pass parameters to the module. These parameters can be defined in the configuration file where the module is included. The most important methods of this interface are the four methods that deal with the transmission of messages. When a message has been received and put into the in-pipeline, the *receiveRequest* or *receiveReply* method of every module in the pipeline is called, depending on the type of the received message. The default implementation in the abstract class is just to tell the pipeline that the consecutive modules should take care of the message (i.e., by returning the pre-defined MOD_OK flag). If, on the other side, a module wants to process incoming messages, it changes the default implementation of one of the *receiveX()* methods.

The same is true for outgoing messages. If an application, a topology module or another processing module sends out a message, the message is also put into a pipeline: the out-pipeline. In the out-pipeline, the *sendRequest()* or *sendReply()* methods are called, again, depending on the type of the message to be sent.

Finally, the *start()* and *stop()* methods are defined to let the Omnix framework activate or shutdown processing modules. These methods could be necessary if at the beginning or the end of the lifetime of a processing module, special has to be taken (e.g., contacting a server or saving information on a disk).

```
public abstract class Module {

    public void initialize(Hashtable properties) {}

    public int receiveRequest(Request req) { return MOD_OK; }
    public int receiveReply(Reply rep) { return MOD_OK; }

    public int sendRequest(Request req) { return MOD_OK; }
    public int sendReply(Reply rep) { return MOD_OK; }

    public void start() {}
    public void stop() {}
}
```

Figure 6.6: The processing module abstract class.

In the following, examples of how processing modules contribute to the functionality of Omnix are given.

6.1.2.1 The Loopback module

Processing orphan messages

A special status has the so-called *Loopback* processing module, which is supposed to be installed at the end of the in-pipeline. Every message that has not been consumed by any other preceding processing module is collected by the Loopback module. It creates a response with a status code that indicates that the requested resource or service has not been found and sends it to the requesting peer. Figure 6.7 shows how this functionality is implemented in a few lines. In case a response has been received and consumed by any other module, it is ignored and removed from the pipeline.

```
public class Loopback extends Module {

    public int receiveRequest(Request req) {
        req.createReply(NOT_FOUND).send();
        return MOD_DONE;
    }

    public int receiveReply(Reply rep) {
        return MOD_DONE;
    }
}
```

Figure 6.7: The Loopback module.

Connecting the ContextSwitcher with the two Pipelines

The observant reader might notice that if this module is installed in the in-pipeline of the Processing Layer, the P2P Network Layer (in form of the *ContextSwitcher*) would never get any message. However, technically the ContextSwitcher itself is just another processing module in the pipeline. Hence, although hidden by the layered, higher-level, abstract architectural model of Omnix depicted in Figure 4.1, this is the interface between the Processing Layer and the P2P Network Layer. Therefore, the Loopback module is placed after the ContextSwitcher and gets only those messages, that even the P2P Network Layer did not consume.

6.2 Implementing topologies

The following sections give examples of how P2P topologies could be implemented in the Omnix P2P framework. To do this, all steps necessary to program a complete topology are discussed and the interfaces provided by Omnix explained. The implementations presented in this section are only shown with the help of (Java-like) pseudo code because featuring the complete code is not necessary for the understanding of the code. In addition to the code, this section also highlights which messages are used to convey search requests, system messages (see page 54), etc. over the network. To demonstrate the usage of Omnix, we take two simple topologies: Nimble groups and a wild mesh topology. More complex topologies can be built the same way, as shown in Chapter 8.

6.2.1 The NetworkModule interface

The interface that has to be implemented by a topology module (and thus defines the functionality of a topology module) is described in this section. A topology module has two major responsibilities:

- A topology module has to implement the methods described in Section 4.2.4.2 (page 51). Those methods provide means for inserting content into the network, searching for information, send messages to other peers, etc. These method comprise the interface to the application on top of the Omnix framework.
- On the other side, a topology module must also have an interface to the communication layer. It must be able to receive messages from and send message to other peers.

Hence, a topology module has two major access points where it can be used by the other components of the system: the application and the lower communication layers in the Omnix framework.

Providing an interface to the application above

The connecting link for the application is provided by the *Network* interface, which has to be implemented by every topology module. This interface, which is shown in Figure 6.8, provides the application all necessary methods for participating in the P2P network. Note, that it is not required for a topology module to implement all the methods depicted in Figure 6.8 (except for the *search* method).

```
public interface Network {

    public String search(SearchDescription sd, SearchResultListener srl);

    public String inject(ServiceDescription[] sd);
    public String remove(ServiceDescription[] sd);

    public ServiceDescription[] injectedLocally();

    public String subscribe(EventSetDescription esd);
    public String unsubscribe(EventSetDescription esd);
    public void notify(EventInfo ei);

    public byte[] invoke(Contact c, String method, byte[] data);
}
```

Figure 6.8: Network interface of topology modules.

Accessing the lower levels of Omnix

To be able to accomplish all these tasks, the topology module also needs to be accessed by the communication layers of Omnix (i.e., the communication layer must be able to pass messages to a topology module). This is all done via the *Module* interface. This interface is the same that is used in the *Processing Layer*. Hence, technically, a topology module could also be used as a standalone processing module. This *Module* interface is shown in Figure 6.9.

```
public abstract class Module {

    public int receiveRequest(Request req) { return MOD_OK; }
    public int receiveReply(Reply rep) { return MOD_OK; }

    ...
}
```

Figure 6.9: Excerpt from the Module interface.

A topology module has only to care about the two method shown in Figure 6.9: *receiveRequest* and *ReceiveReply*. Whenever a message is received by the Transport Layer and processed by the Processing Layer, the message is conveyed to the topology module through these two methods (depending on the type of the message). The topology module is then able to process the message (i.e., by searching its local database in case a search request has been received).

The result: the NetworkModule

The combination of the two interfaces described above results in the *NetworkModule* interface (Figure 6.10), which is the actual P2P network module interface. As exemplified in line 4 of this figure, all methods other than *search* throw an exception if they are not implemented by a topology module.

```
1  public abstract class NetworkModule extends Module implements Network {
2
3      public String inject(ServiceDescription[] sd) {
4          throw new NotImplementedException("Not␣implemented");
5      }
6
7      ...
8  }
```

Figure 6.10: NetworkModule interface of topology modules.

Equipped with this interface, a topology module is able to receive messages (sending messages does not require a interface for the topology module) and to be accessed by the application. What is left is the implementation of the functionality that is needed to create a P2P network (i.e., P2P topology). Examples for such topology modules are given in the next sections.

6.2.2 Nimble Groups

Nimble groups are the plainest form of P2P networks imaginable. They have only limited use due to their reduced scalability. Still, nimble groups may be interesting when few devices are connected without the existence of a server.

6.2.2.1 Topology

The topology of nimble groups is very simple. Every device is connected to any other device using broadcast (or multicast) algorithms or by sending the same information to each peer connected (see Figure 6.11). This way, all devices attached get the same view on the same information available (i.e., search requests, system messages, etc. – it depends on the implementation whether search responses are directly sent back to the initial peer or also shared with all other peers). An example for a nimble-style P2P network is Groove. In Groove networks, program instances send messages to all others in the same workspace. Hence, all Groove instances share the same information (e.g., a shared browser, a drawing board editable by every workspace members, etc.). To have a common view of the information provided, nimble groups do not use a central server.

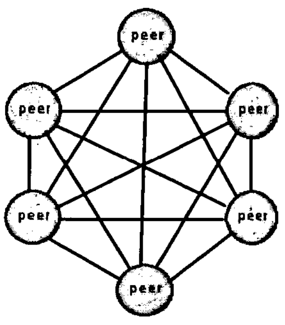


Figure 6.11: The nimble group P2P topology.

Nimble groups are typically not scalable because information has to be conveyed to every peer within the group. While this is easily achieved in a local network by using broadcast algorithms, this is almost impossible in an Internet-wide group with more than just a bunch of peers. The costs for sending messages increase exponentially with each peer joining the group.

6.2.2.2 Messages exchanged

What kind of messages are necessary in a Nimble group?

Depending on the complexity of the Nimble group, different message classes exist. What all systems have in common is that they provide means for searching content at other peers or to send direct messages to other peers (e.g., a chat message, a URL for the shared browser, etc.). These messages can be either sent via broadcast (or multicast) messages or directly to each of the peers in the group.

Unicast vs. broadcast in Nimble groups

If broadcast algorithms are used, there is typically no need for system messages to acquire information about other peers in the group. Still, these message might be convenient to get additional information on the network (e.g., the number of peers connected and their status). If messages are sent in a unicast form to the other peers, additional messages for detecting peers in the *network neighborhood* are necessary. Typically, these messages are broadcast messages (there is no other way how other peers can be detected without a central server). To detect a drop out of one of the peers, *IsAlive* messages have to be sent repeatedly.

If the complexity of the Nimble group is very high, additional messages are needed. If, for instance, a new peer joins the group, it must get all information that the other peers already have. Furthermore, if an (optional) central server is used for storing information persistently, messages supporting this functionality are also needed.

6.2.2.3 Implementation

The Nimble-group topology module presented in this section is a very simple one. It does not support anything else than placing content in the network and to search for this content.

Meta-data describing the actual shared data (or services) is represented by using the *SimpleServiceDescription* class, which is provided by Omnix. It is a simple class that provides key/value pairs and a method for converting these pairs in a binary array so that it can be sent in an Omnix message (and, of course, a method for parsing such an array to restore information transferred over the network).

Injecting meta-data in a Nimble group

The easiest thing to implement is the *inject* method, which is merely passing the *ServiceDescription* object to a local storage object (called *SimpleServiceDescStorage*), which is again a helper component that comes with Omnix. The SimpleServiceDescStorage component (with the name of *sds* in our code example) takes the service description and stores it in its memory. Figure 6.12 shows how an array of ServiceDescription objects is put into the storage component.

Searching for data in a Nimble group

The next step is to allow the application to search for data in the network. The topology module has to craft a search request that can be sent to the other peers. This search request must contain specific information so that the other peers are able to understand the request.

Figure 6.13 shows how a search request can be constructed. The variable *multicastAddress* is defined by the Omnix system. Since the topology modules does not know which transport module is used to convey the search request (which means that it does not know

```
public String inject(ServiceDescription[] sd) {

    for (int i=0; i<sd.length; ++i) {
        sds.addServiceDescription(sd[i]);
    }
    return null;
}
```

Figure 6.12: Injecting service descriptions in a Nimble group.

```
public String search(SearchDescription sd, SearchResultListener srl) {

    Request searchReq = new Request();

    searchReq.setTo(multicastAddress);
    searchReq.setMethod("SEARCH");
    searchReq.setProtocolName("OMNIX");
    searchReq.setProtocolVersion("1.0");

    searchReq.setContent(sd.searchTerm());
    searchReq.send();

    listeners.add(searchReq.getMessageID(), srl);

    return searchReq.getMessageID();
}
```

Figure 6.13: Searching within a Nimble group.

what addressing scheme is used), it can only use this variable as a multicast address. The values of the request entities method, protocol name and protocol version can be arbitrarily set by the topology module. But care has to be taken that the other peers use the same values.

The content, which is the specification of the search request, is provided by the *SearchDescription* component, which provides a method *searchTerm*. This method returns a binary representation of the search term (e.g., a regular expression).

Note that the search method is not blocking until a search result has been received. This would be impossible because there is no point in the time of execution where it can be determined whether all search results have been received. The application must provide a *SearchResultListener* component. This component is used by the topology module to report back search results received from other peers.

Eventually, the message is sent and the ID of the message is returned as an identifier for the search request.

Collecting and processing search results

The next step when searching in the P2P network is to process received responses from other peers (hopefully with appropriate search results). If the topology module receives a response, it calls the method *addResult* of the *SearchResultListener* component provided by the application and refers to the corresponding search request with the message ID.

```
1  public int receiveReply(Reply rep) {
2
3      Transaction t = rep.getTransaction();
4      Request r = t.getRequest();
5      if (r.getSenderModule() != this) {
6          return MOD_OK;
7      }
8
9      if (rep.getStatusCode() == OK) {
10         ServiceDescriptions sds = new SimpleServiceDescriptions();
11         sds.parseDescriptions(rep.getContent());
12
13         SearchResultListener srl = listeners.get(rep.getMessageID());
14         srl.addResult(rep.getMessageID(), sds.getDescriptions());
15     }
16
17     return MOD_DONE;
18 }
```

Figure 6.14: Receiving search results within a Nimble group.

Figure 6.14 demonstrates how messages containing search results are processed by the Nimble group topology module. The *receiveReply* method is part of the *NetworkModule* interface.

When a reply is received, a topology module must make sure that it is indeed the sender of the corresponding request. This is done with help of the *Transaction* component attached to the Request/Reply pair (lines 3 to 7). A *Transaction* is used to match Request and corresponding Replies (which have the same message ID; see Section 6.3.1.1 for more details on transactions in Omnix). The module checks whether it is the sender of the appendant request. If this is not the case, it simply returns a code indicating that other modules should take care of that.

Extracting the search result from the messages received

If the message is a reply to a request sent by this topology module, it further checks whether the reply has a status code of 200 (i.e., if its not an error – see Appendix A for a list of status codes defined in Omnix). Having done that, the service descriptions are extracted from the reply and parse by a *ServiceDescription* component. The resulting array of service descriptions is then passed to the application, which is done through the *SearchResultListener* the application has provided while calling the search method. The correct SearchResultListener is retrieved from a collection of result listeners and identified by the message ID.

That is everything necessary to implement the interface to the application on top of Omnix. The second part is to provide a search functionality to other peers. Figure 6.15 shows how this can be done in a few lines.

```
1  public int receiveRequest(Request req) {
2
3      if ((!req.getProtocolName().equals("OMNIX")) ||
4          (!req.getProtocolVersion().equals("1.0")) ||
5          (!req.getMethod().equals("SEARCH"))) {
6
7          return MOD_OK;
8      }
9
10     SearchDescription sd = new SimpleSearchDescription();
11     sd.parseTerm(req.getContent());
12     ServiceDescriptions result = sds.search(sd);
13
14     Reply rep = req.createReply(OK);
15     rep.setContent(result.getBytes());
16     rep.send();
17
18     return MOD_DONE;
19 }
```

Figure 6.15: Receiving search requests within a Nimble group.

Sharing data with other peers

When a topology module receives a request, it must first make sure that it understands the meaning of this message. To do this, it must know exactly about the method used in the message, the protocol version and the protocol number. If one of these values do not match the topology module, it should not process the message (as can be seen in lines 3 to 8). In this code example, the message is passed on to the next module.

If the message is understood by the topology module, it parses the content (i.e., the payload) of the e message into a *SearchDescription* component (if this fails – for instance, because the content has another format – an exception is raised and an error message is automatically returned to the remote peer). After that, the local database of *injected* service descriptions is searched and the result stored in a *ServiceDescriptions* component, which is just a container of multiple *ServiceDescription* objects.

The remainder is just creating a reply, putting the search result into this object and send it to the remote peer that requested the search. Finally, the method returns a status code indicating that the message should not be further processed (i.e., it has been consumed).

The next section shows how a little bit more complex topology can be implemented in Omnix.

6.2.3 Wild mesh

Although wild mesh P2P networks are a little bit trickier than Nimble groups, they still have a low complexity. They are deemed to be very silly topologies because they have limited scalability. Still, they have some advantages that other P2P network do not have (see Chapter 3).

6.2.3.1 Topology

In a wild mesh topology, a peer has only a small set of neighbor peers that it connects to. How this set is acquired and the size of this set depends on the implementation. As a reference, in Gnutella a peer typically maintains only four connections to other peers. The structure of the network itself cannot be predicted because a peer may connect to any other peer in the network. As Figure 6.16 indicates, a wild mesh P2P network does not necessarily have a perfect structure.

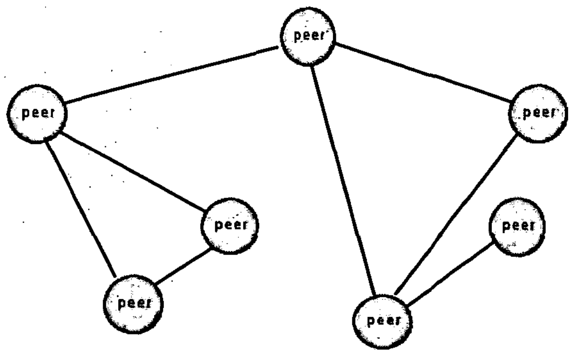


Figure 6.16: The wild mesh P2P topology.

Search messages are forwarded along the open connections a peer has. If a peer receives a search request, the message is forwarded to the other peers that is is connected to. After a pre-defined number of hops, the message is removed from the network. Replies typically take the same route back as the request took before. This way, intermediate peer may cache search results and learn about other peers in the network.

6.2.3.2 Messages exchanged

Keeping up the topology and searching for content

In a simple wild mesh network, three classes of messages are used. The first one is used to search for information in the network. A peer sends a search request to the set of connected neighbor peers. If a peer receives such a search request, it searches the local database and forwards the search request to its set of neighbor peers. This is repeated until the maximum hop count of the message has been reached.

The next type of message in a wild mesh network is the response message. It contains (hopefully) meta-data about and pointers to the desired information, files, services, etc. Response messages typically use the same route as the request, but in the reverse direction (obviously). This has two reasons: peers in between the sender and the receiver can store the content of the search result in its local cache so that future search requests can be enriched by this extended information. Furthermore, peers in the middle may learn about other peers in the network very easily (if the search result contains a list of peers visited on its way from the content provider to the content requestor). This way, a peer may compile huge lists of other peers in the network.

The last message class necessary in a minimal wild mesh network is the so-called *IsAlive* message, which is necessary to ascertain whether all peers in the set of connected peers are still available. Hence, peers send a message (in Gnutella, it is called a *Ping*) to their neighbors and await a response. If this response is not received within a certain time-interval, the connection to this peer is closed and a new one is opened to another peer (from the huge list of other peers mentioned above). This way, a peer has a consistent number of active links to other peers in the network.

6.2.3.3 Implementation

In this section, the most important aspects of how to program a wild mesh P2P topology module in Omnix are presented. These include the insertion of content into the network as well as searching in the network.

There are many similarities to the functionality of the Nimble group topology module presented in Section 6.2.2. Injection, for instance, is exactly the same. If the application wants to insert a service description, the topology module simply puts it into its local database, without the necessity of creating a network connection to any other peer.

Sending a search request to multiple peers

Searching in a wild mesh P2P network is a little bit more complicated than in a Nimble group. The reason for this is that search messages are not plainly sent to other peers in the nearby (e.g., by using broadcast communication) but sent directly to a pre-defined set of remote peers – those that the peer is connected to. The implication is that the peer first must acquire a list of connected peers. This is usually done by sending a request to a central server which holds a list of known peers. Once a peer has connected to one of these peers, it may collect new addresses.

So, when the application wants to search for information, it calls the method *search* of the topology module (here, we can see that it does not make a difference for the application which topology module is actually used – be it a Nimble group, a wild mesh network, or any other topology). The wild mesh topology module sends a request to a list of connected peers (as shown in Figure 6.17).

In contrast to the Nimble group topology module described above, the wild mesh topology module iterates through a list of connected peers (as shown from line 9 to 13) and sends the same search request to each one of these. Replies are processed the same way as in the Nimble topology module (see Figure 6.14), but with a little extension that will be explained later in this section.

Implementing multi-hop messages with minimal effort

```
public String search(SearchDescription sd, SearchResultListener srl) {

    Request searchReq = new Request();
    searchReq.setMethod("SEARCH");
    searchReq.setProtocolName("OMNIX");
    searchReq.setProtocolVersion("1.0");
    searchReq.setContent(sd.searchTerm());

    for (int i=0; i<activeContacts.length; ++i) {

        searchReq.setTo(activeContacts[i].getAddress());
        searchReq.send();
    }

    listeners.add(searchReq.getMessageID(), srl);
    return searchReq.getMessageID();
}
```

Figure 6.17: Sending search requests to connected peers in a wild mesh network.

If the wild mesh topology module receives a search request from another peer, it has to do two things. First, it has to search its local database and return matching results, and it has also to forward the message to its set of connected peers. Figure 6.18 shows how this functionality can be achieved in Omnix in a few lines of code.

The first part of the processing search requests is the same as in the Nimble group topology module. It has to check whether the received request has the correct protocol name and version, and is a search request. Then, the local database is searched and a reply containing the result is sent. In addition to that, the wild mesh topology module has also the duty to forward search requests to other peers (but only, if the search request has not already reached the maximum hop count, which is determined by the TTL value included in the search request). Figure 6.18 shows how the search request is forwarded to the list of connected peers (which are contained in the *activeContacts* component).

Preventing loops and message duplicates automatically

Two important things are missing in this code example. First, in a wild mesh P2P network a peer has to make sure that it detects search request duplicates (either caused by different peers forwarding a search request to the same peer or by loops). This is done by the Omnix framework by comparing the message IDs and is therefore not required in the topology module.

The second part would have been to check whether one of the connected peers (in the *activeContacts* list) has already received this message, which is determinable by the list of visited peers in the search request. This list of visited peers in the search request is managed by the Omnix framework. When a request is sent to another peer, the Omnix framework automatically appends a *Via* header (see page 67) with information about the sending peer

```
public int receiveRequest(Request req) {

    // check whether the message is a search request and
    // process the search request locally
    // (see Figure 6.15, lines 3-16)

    int ttl = StringUtils.parseIntSafely(req.getHeader(TTL), 0);
    if (ttl > 0) {

        req.setHeader(TTL, --ttl);

        for (int i=0; i<activeContacts.length; ++i) {
            req.setTo(activeContacts[i].getAddress());
            req.send();
    }

    return MOD_DONE;
}
```

Figure 6.18: Processing a search request in the wild mesh topology module.

(i.e., address and port information). This information is used to prevent a message from being sent to an already visited peer.

When a response is received, it automatically removes the top *Via* header which should be this peer's address. The information contained in the *Via* header is also vital for sending back responses in a wild mesh network.

Transporting a reply back on the route the request used

As shown in Figure 6.19, the wild mesh topology module forwards a received reply in case that it has not issued the appendant request (line 6). This is done by getting a list of peers the request has visited on its route to the peer that has originally sent the reply (this *Via* list is copied from the request to its response). The top entry of this list (lines 8 to 9) is the next hop of the response's route to the sender of the request. In addition to that, the topology module also makes use of the *Via* header to expand the list of known peers (line 11).

What is left is to check from time to time whether connected peers are still alive. This can be done in a separate thread which periodically sends requests to the connected peers. If the peers do not send a response within a certain time interval, the peer is removed from the *activeContacts* list and a new, random entry from the list if known peers is chosen.

6.3 Advanced issues

This section deals with more general (and more complex) aspects of Omnix, which are not apparent at first sight. It highlights how the more advanced features of Omnix contribute to creating a versatile P2P framework with a broad range of possible applications.

```
public int receiveReply(Reply rep) {

    Transaction t = rep.getTransaction();
    Request r = t.getRequest();

    if (r.getSenderModule() != this) {

        ViaList vias = rep.getVias();
        Via nexthop = vias.getLastVia();

        contacts.add(vias.getContacts());

        rep.setTo(nexthop.getAddress());
        rep.send();

    } else {

        // inform application about search result.
        // (see Figure 6.14, line 9-15)
    }

    return MOD_DONE;
}
```

Figure 6.19: Receiving search results within a wild mesh topology module.

6.3.1 Quality of Service

Transport QoS vs. P2P QoS

When talking about the QoS of networking in Omnix, two separate issues have to be dealt with. On one side, Omnix has to ensure that messages exchanged with other peers are guaranteed to be delivered. Hence, the framework must provide means to ensure that everything possible is done to deliver a message to the target peer. On the other side, QoS in a P2P network could also include the functionality of the P2P system itself. Hence, the quality of service of a P2P system could also be determined by its scalability or fault-resilience. Furthermore, a P2P system's ability to perform a successful search in the network if the desired information is available (e.g., in a wild-mesh network, this guarantee cannot be given, while in a server-based network this would be technically possible) might also be used as a measurement for the QoS.

In this section, Quality of Service (or, QoS) is only applied to networking. It does not include any other part of the Omnix P2P framework than the transport layer and the streaming component. The reason for this is the fact that the QoS of the P2P network itself entirely depends on the implementation of the topology module. The fault tolerance in case of lost connections between peers, for instance, is determined by the topology of the network, not by the Omnix framework. The number of messages a peer has to process per second to maintain the structure of the P2P network is also determined solely by the applied topology.

Hence, the Omnix P2P framework cannot influence any of these QoS aspects.

For this reason, this section concentrates on discussing how QoS can be achieved in the transport layer and in the streaming component.

6.3.1.1 Transport QoS

Maintaining QoS in the transport layer means that the communication infrastructure of Omnix makes sure that everything possible is done to successfully transmit a message to the destination peer. In Omnix, this could be done in two different ways, both have their advantages and drawbacks.

Where to place transport QoS in the Omnix framework?

The first solution would be to use so-called transactions.[2] The transaction module, residing in both pipelines of the Processing Layer, are used to re-transmit messages if no response has been received or a request is repeatedly received although a response has already be sent to the requestor. The most important advantage of putting this functionality into a processing module is that it works for all transport modules the same way. Hence, programmers of transport modules do not have to think about this type of QoS in their implementation because it is already provided by the framework. Furthermore, this helps also to reduce the size (i.e., footprint) of transport modules because common functionality is out-sourced to a single component in the processing layer. Omnix comes with a prototype Transaction module that provides this functionality. The drawback of this approach is that this Transaction module is not able to take the different implementations, network types, communication protocols, etc. used in the transport modules into account. If, for example, a network module uses a connection-oriented link (e.g., TCP/IP) to a remote peer, this module is not needed and may consume viable resources.

This problem leads to the second possible solution: implementing QoS directly in the transport module. Having QoS measures at the source of the problem, more complex features could be used to achieve a high QoS. Providing QoS could be as easy as using reliable communication protocols such as TCP. However, it fully depends on the implementation of the topology module and is therefore not part of the Omnix P2P framework.

6.3.1.2 Streaming QoS

The Omnix framework provides means for opening a stream between two peers (see Section 4.3). For this purpose, a *StreamManager* uses pluggable components (called *StreamProvider*) that are responsible for creating the stream and to return a handler to this stream (in form of a *StreamControl* object).

In Omnix, a reference implementation of a StreamProvider uses TCP/IP connections to stream data over the network. However, it is possible to add new StreamProvider, that provide more complex streams than just plain TCP/IP streams.

[2]The term *transaction* is typically used in, but not restricted to, the database area. We use this term in the same context as the authors of the SIP protocol [87], a logical unit of work.

What would be the reason to use another connection-oriented protocol than TCP, one might ask? The answer to this is that different applications call for different types of streams. A simple example is real-time multimedia streaming. In real-time audio or video streaming, it is more important to have timely delivery of data instead of re-sending lost packets (because they would be received long after they have lost their significance). Hence, for real-time multimedia transmission, TCP/IP would be a bad choice. It fact, real-time protocols mostly use UDP.

One example for a StreamProvider could be one that supports the Real-Time Transport Protocol (RTP). RTP has been designed to send data with real-time characteristics over a network. Real-time data may be audio or video information but RTP is not limited to that specific kind of data. Any other application that requires a protocol allowing continuous streaming of data might use RTP. RTP supports a wide range of underlying networks and protocols, but in the realm of the Internet, UDP is preferred for its ability of multicasting. A short introduction to RTP is given in [98].

RTP includes a companion control protocol, the RTP Control Protocol (RTCP), that is responsible for feedback on the quality of the data transmission, session member identification and providing information about all session members. This is achieved by the hosts sending information in RTCP packets to all session members periodically. Sender and receiver reports allow the detection of transmitting problems and their nature (whether they are local or global). The RTCP requires control packets to be sent periodically the same way as the data packets.

So, there are a lot of things to do to get QoS in a stream. Still, all this could be implemented in a *StreamProvider* and executed by a *StreamControl* object and is therefore completely separated from the application using it.

6.3.2 Firewalls and NAT

How a P2P system can deal with the problem of bypassing restrictive firewalls and NAT boxes is explained on page 62. Omnix is open for every one of the three possibilities presented there. It may be configured to use TCP over port 80, to maintain outgoing connection so that other peers are able to use these connections to contact the peer, or to use a proxy that forwards messages through the firewall.

The third possibility is considered to be the most "systemadministrator-friendly" approach (we don't want to annoy the sysadmin, do we?). It has the advantage that, when set up correctly, it works guaranteed. This cannot be said about the other two solutions. Hence, this section will mainly focus on how to create and use a proxy with the Omnix P2P framework.

6.3.2.1 The peer

Using a proxy to communicate with the world behind the firewall

There are two possibilities where a peer might need the help of a proxy peer. First, it may be the case that the firewall does not allow the sending of messages to remote peers. Second, the firewall may hinder remote peers to send messages to peers inside the firewall. In both cases, the Omnix framework allows the application and/or any module to cope with the situation.

If the client is not permitted to send messages directly to other peers in the network, it must be configured to send these messages to a proxy instead, which then forwards to the message to the target destination.

Differentiating between target-peer address and next-hop address

It is not the responsibility of a transport module to detect whether a message should be sent to the address defined in the message or to the proxy's address (as it is only concerned about sending the message directly to another communication endpoint). Hence, a processing module is used to cover this functionality. Such a module could replace the destination address of the target peer by the address of the proxy peer. For this reason, a message object must include two addresses. One of the target peer and one of the next *physical*hop (i.e., the proxy) of the message. The first address (the *To:* address) never changes in a message. The other one (called *Network-To:*) always denotes the address of the next peer on the route from the sender to the receiver. A simple proxy transport module is shown in Figure 6.20. If a message is about to be sent, the module simply sets the *Network-To* address.

```
public class Reroute extends Module {

    public int sendRequest(Request req) {
        req.setNetworkTo(proxy-address);
        return MOD_OK;
    }
    public int sendReply(Reply rep) {
        rep.setNetworkTo(proxy-address);
        return MOD_OK;
    }
}
```

Figure 6.20: Rerouting requests to a proxy.

Every transport module than has to check whether the *Transport-To:* field is set. If this is the case, this address will be taken instead of the address in the *To:* field as the destination. Note that the *Transport-To:* will not be transmitted over the network.

With these four simple lines of code, the peer is now able to use a proxy for every outgoing message. But how do the other peers know, that they may not be able to send requests to this peer directly? How does a peer announce that the remote peers have to send messages for this peer to the proxy peer instead? The solution is that peers include their contact information in every request sent out. This information can be used to tell other peers which proxy to use if they want to send messages to this peer. But for a client peer, it is hard to determine how the proxy peer should be contacted from the outside (e.g., which port should be used?). Hence, including this information in the message should be done by the proxy.

6.3.2.2 The proxy

Adding routing information to messages passing a proxy

If a proxy receives a request that should be sent to a remote peer outside the firewall (or, more general, outside the proxy's domain), it has several tasks to do before the message can be forwarded to the destination peer. It has to add a *Via* header field, which every hop on the route from the requestor to the destination peer should add to the message. This way, it is possible to trace the route from sender to recipient and it is also necessary to allow responses also be routed through the proxy (if responses take the same route back as the request took before; this is accomplished by running down the list of visited peer stored in the *Via* header fields).

If an incoming request wants to pass the firewall, it has to do this via a proxy. The question of how a remote peer gets the information that a peer inside the firewall can only be accessed via a proxy, is still not answered. The solution is provided by the proxy. If it forwards an outgoing request, it automatically appends information about how to contact the sender in future requests. This information is stored in the *Contact* field.

If a proxy receives a request, it simply forwards it to the peer indicated in the *To* header field. If, on the other side, a response is received, it removes the top *Via* header (which should be the address of the proxy itself) and forward the response to the next address in the next, now top, *Via* header entry.

6.3.2.3 Proxying streams

If normal messages have to be proxied, it might be as well the case that streams have to be sent through a proxy to bypass a firewall. As with transport modules, *StreamProvider* should not be required to support the usage of proxies. Hence, this functionality is put into the *StreamManager*, which is responsible for 1) setting up the stream to the proxy instead of the destination peer and 2) to instruct the proxy where to forward the streamed data to. For this purpose, the proxy must create two streams, one that accepts a stream from the sending peer and a second one opened to the target peer. When information is received from the sending peer, the proxy automatically puts this information in the stream to the target peer.

6.3.3 Bridging topologies

Connecting two heterogeneous P2P networks

Omnix supports a peer to be concurrently connected to multiple networks with different topologies. This fact could be used to build a bridge between two non-compatible P2P networks.

Currently, every P2P system uses its own protocol to maintain connectivity of the network, to search for information and to access information (e.g., downloading a file, drawing on a shared blackboard, etc.). The reasons for this diversity of protocols are manifold. Each

P2P system uses a protocol that best fits the requirements of the application. In the Gnutella network, for example, only a fixed set of 5 messages with a rigid structure are defined. The messages are structured in a way so that they use minimum bandwidth. Groove, on the other side, uses complex and encrypted XML messages to convey information to other peers. The protocol and the routing algorithm determine the functionality of the P2P network. Hence, they are chosen to meet the requirements in an optimal way. Peers of different P2P systems cannot be interconnected easily. They maybe 1) use different network protocols for communication (e.g., UDP vs. TCP), 2) use fixed, different port numbers, 3) use different encodings, ontologies and structures for messages, 4) have different semantics of messages, 5) support different services, 6) use different routing tables, and 7) have different security measures.

Is it always possible to connect two P2P networks?

So, is it still possible to interconnect different P2P networks? The answer is yes, but it depends on the systems to be connected. It only makes sense where there is an overlap of functionality. The Gnutella network and the FastTrack network could be easily interconnected. If a SuperPeer in the FastTrack receives a search request, it creates a Gnutella network and sends it to other peers in the Gnutella network. The incoming results are then forwarded to the requesting FastTrack client peer. Accessing the file in the next step must also be done via the same SuperPeer. Hence, it must present itself as the provider of all resources found in the other network. Otherwise, it would be impossible to access the files in the alien network.

It does not make sense, for instance, to connect an instant messaging P2P system with a file sharing P2P system. On the other side, it would be possible to connect different types of P2P networks, but then only a small subset of messages could be forwarded to the other network. For example, Groove and Gnutella could be connected although they have a completely different application domain. Still, it could be possible to use a Gnutella client to search for files in a Groove network (this would also be possible in the other direction). But every other functionality of Groove (e.g., chat, shared browser, etc.) could not be forwarded into the Gnutella network.

To interconnect two different P2P networks in Omnix for each protocol used, a separate transport module that parses and generates messages of this protocol is necessary. These transport modules decode incoming messages into a generic *Message* object. A topology module is then able to read the information stored in the Message object.

Where to place the connecting point of two P2P networks

There are two ways how two P2P systems could be connected. Either a topology module understands both networks and is able to translate between the two protocols, or an application on top of Omnix uses two different topology modules. In the following, advantages and drawbacks of both ways are explained.

- *Multiprotocol topology module:* It is possible to write a topology module that is able to support more than one topology (i.e., P2P system). If it receives a message from one system, it simply forwards it to the other system. It has full control over all messages exchanged with both P2P networks.

- *Application on top of two topology modules:* In this case, two separate topology modules are used to connect two P2P networks. If a topology module receives a message, it informs the application when then forwards the message to the other network by using the second topology module.

The first solution has clearly the disadvantage that this single module could not be used to connect one of the involved P2P systems with a third one. It would only support a fixed set of P2P systems. The advantage, however, of this approach is that it provides full flexibility over both networks. The drawback of the second solution, albeit being very flexible, is that it is restricted to the interface between topology modules and applications, which is defined by Omnix. Hence, the application has less flexibility than a topology module. Hence, the first solution would be preferable.

6.3.4 Different topologies, same network

Connecting peers with different topology implementations

This section deals with mixing different topologies in a single network. This situation is possible when a network is in the process of switching from one topology to another. But there are also scenarios where it might be desirable to mix two or more topologies in a single network.

An Omnix peer could be the member of multiple networks. The mapping between messages and the different networks is done by putting the network's name into the message (Section 4.2.4.1 provides a more detailed explanation of the notion of networks in the Omnix framework).

One might argue that a mix of different topologies is just another topology. But this is not valid in this context. In this section, mixing is not done statically. It is not clear for every peer in the network which topology the complete network has (unlike today's P2P networks, where every peer is aware of the overall topology – which does not necessarily mean that they know the actual peers).

Technically, it is possible that peers in a network use the same protocol, the same message semantics and syntax, but different topologies. This means that that depending on the peer's topology module, different routing algorithms are employed. Figure 6.21 shows a simplified example of how different topologies could be connected. In this figure, three network topologies are combined: a wild mesh network (e.g., as Gnutella), a server-based network (such as Napster), and a distributed hashtable like Chord [23].

If Peer A receives a message from the wild mesh *sub*-network, it forwards it (as a client) to the central server also available in the network (Peer B). If a result is found at the server, it returns it to the client, which then forwards the response back to the originator of the search request. In parallel, the server may also forward the search request to another node which is part of a third topology (Peer C). This is not only valid for search request. Any other message

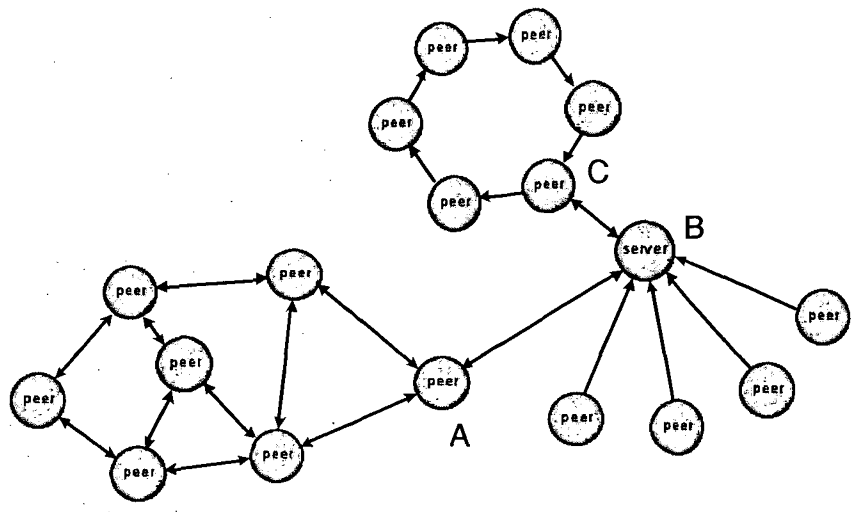


Figure 6.21: Mixing topologies in a single P2P network.

defined in the network could be transported beyond the borders of a single topology. All that is necessary is that the protocol supports the traversal of varying topologies (i.e., it must be able to carry the information required for all topologies involved). In a wild mesh network, for instance, a list of visited hosts and the time-to-live (TTL) information must be stored in the message.

An important aspect in a P2P network – *loop detection* – might be a challenge in a network of mixed topologies. A distributed hashtable usually performs loop avoidance by sending messages only to peers that have a higher chance of being at the target destination. In a server-based network, there is no need for loop-detection at all. These two topologies might have problems with a wild mesh topology attached to the network, because it requires that peers actively check whether a message has been received before. This is not only important for loop detection but it also necessary to detect repeatedly sent messages (in a wild mesh network, there is no routing strategy that forbids two different peers to send the same message to the same destination). Hence, it is not sufficient to simply connect two or more topologies together. Every topology module has to implement all measures necessary to avoid loop detection in general (e.g., as in the wild mesh topology).

Having established that it is technically possible to mix two or more topologies may be interconnected, the next question is whether it makes sense to purposely mix topologies or whether there are circumstances where topologies are mixed accidentally.

Adding another topology to a P2P network may be reasonable if some peers are not capable of meeting the requirements of the original topology. A handheld device is a good example for such a case. Having only limited resources (such as a low bandwidth connection, low CPU capacity, etc.), it is generally not a good idea to connect this device to a wild mesh network. The sheer number of messages would exceed the capabilities of the device. Hence, it is necessary to connect the handheld device to a server (e.g., within the group or the company), which is then connected to the wild mesh network. The handheld device stores all shared data on the central server and is not bothered with search requests from other peers, because all messages are sent to the server. The result is a mixture of topologies: a small server-based P2P network is connected to a wild mesh network.

If a group of peers has requirements that cannot be met by the original topology, these peers may use an alternative topology but stay connected to the original P2P network. A group of people exchanging large amounts of data may want to keep the traffic within their group. Still, if they want to search in the overall network, they are still able to issue search request to other peers outside their group. This scenario could be configured by having a large hybrid P2P network, while the group is connected as a Nimble group. A single peer is needed to forward "global" search requests to the hybrid P2P network.

Having multiple topologies in a single network may not only be needed due to special requirements but there exist also scenarios where it is inevitable – at least for a short time – to mix topologies. If a system administrator decides to change the topology of a network (e.g., because a new P2P topology would allow more search messages per second), it cannot change all peers at the same time. If lucky, the sysadmin may be able to automatically change the topology at all peers currently online. Still, the other peers would not be affected by this change and would still use the former topology. It may be necessary to support also these peers until all of them have switched topologies.

The example scenarios described above show that it is useful to support multiple topologies in a single P2P network. Still, it is worth noting that the routing algorithms of the various topologies only apply to search requests. Accessing shared files or services is still done directly in most cases (a notable example would be Freenet, where direct connections between provider and consumer do not exist).

6.3.5 Connection-oriented vs. connectionless communication

This section discusses the advantages and drawbacks of connection-oriented vs. connectionless communication protocols.

Connection-oriented communication means that when two devices exchange information, at first a handshake is performed. This is needed to create a virtual channel from one endpoint to the other. Packets are routed through this channel and are received in the same order as they were sent. Furthermore, connection-oriented protocols also support some sort of delivery guarantees. If a message has not been received, the sender automatically resends the packet. Packets typically have a continuous number that allows the receiver to detect missing packets. The handshake mechanism is used to synchronize the two endpoints. It could also be used to change the characteristics of the connection (e.g., QoS settings). Connection-oriented communication requires bi-directional communication. If communication only flows in one direction, it is not possible for the two endpoints to agree on a virtual channel.

Connectionless communication, on the other side, does not provide any of the aforementioned features, such as delivery guarantees or the correct ordering of received packets. There is no handshake protocol and no dedicated virtual channel. The sender simply sends packets to the receiver without making sure that the packets reach the receiver or even checking whether the receiver is still online. Delivery guarantees and packet ordering have to be implemented in the application layer (according to the ISO OSI layer model [99]).

There is no doubt, that in a P2P middleware framework, both kinds of protocols should be supported. But, the question is which protocol should be preferred? In Omnix, the default transport protocol used is UDP, a connectionless protocol. The reasons for this decision are given below:

- *Delivery guarantee:* Having delivery guarantees (if such a thing exists; it would be more appropriate to call it "best effort") is generally desirable in protocols. But connectionless protocols do not support such a feature. Hence, it must be implemented on top of the protocol (which is done in Omnix in the Processing Layer). Having such a service paves the way for small, effective protocols – hence it is no longer necessary to use connection-oriented protocols.

- *Performance:* Connection-oriented protocols typically need some time and bandwidth to establish the connection. If a peer only wants to send a single message, this overhead would not be justifiable. Furthermore, the feature typically supported in connection-oriented protocols (such as delivery guarantees or the correct ordering of received packets) also add to the bandwidth requirements of the protocol itself. It may be as well the case that these services are not required by the P2P network. In this case, a connectionless protocol would suit better (e.g., although being a streaming protocol, the Real-Time Transport Protocol (RTP) uses UDP instead of TCP due to the high overhead of the TCP protocol).

- *Agility:* In a P2P network, peer typically connect and disconnect at a frequent rate. When a connection-oriented protocol is used, a virtual channel between the two peers is created. This is typically done by the operating system that provides this functionality. A connection between two end-points has to be shutdown cleanly – otherwise, the operating system would have to wait for a timeout until it detects that the remote endpoint is no longer available (TCP, for example, tries to resend packets for a pre-defined period of time). This behavior may be undesired in a P2P network with frequent disconnections. Connectionless protocols may be used to fine tune this functionality.

- *Complexity:* When using a connection-oriented protocol, the handshake has to be performed first. In many protocols (such as HTTP, SIP, etc.) this overhead is reduced by keeping connections between endpoints open, even after the communication has finished. These lines are kept open to avoid the tedious handshake if the same two endpoints want to communicate again. Hence, every peer has a pool of open connections to other peers. The management of this pool (including keep-alive timeouts for connections, the maximum number of connections, etc.) is not too complex, but still is not reasonable in a P2P network, because connections may be cut off frequently.

- *Congestion:* In the TCP protocol, the receiver sends an acknowledgment upon the receipt of a packet. If the receiver has an asymmetric connection (e.g., DSL or Cable), upload bandwidth is typically lower than download bandwidth. Hence, the acknowledgment packets get slower to the sender than the actual content to the receiver. This has the effect that the sender has to wait until the acknowledgment packet has been received before it can send the next packet, although the receiver's download capacity would allow for a faster transmission of information. Using a connectionless protocol would mitigate this problem.

However, the Omnix protocol also supports connection-oriented transport protocols (e.g., by supporting header-fields such as *Connection*, which determines whether the connection should be kept alive or not).

6.4 Testing

The exceedingly distributed nature of P2P applications makes it very difficult to test a P2P application before it is installed on millions of computers on the Internet. But verifying the correctness of a P2P application is vital, because it cannot be easily exchanged by a never version once it is spread over the network. So what can be done to test a P2P application at development time?

Mathematical models vs. simulations

There are typically two ways: either the correctness of the software is verified using mathematical models or a simulator is used to simulate thousands or millions of peers interacting. Using mathematical models only is not desirable because it could only be used with a very simple model that does not necessarily reflect the real world (it would be too complex and therefore impossible). Examples for an analysis of a distributed system in general are given in Jogalekar et al. [100] and Keidar [101] whereas the analysis of a P2P network is exemplified in [102]. Simulations are more complex and can therefore be seen as a way to verify whether the simpler mathematical model is valid. Hence, simulation should be used in addition to analysis to better understand the dynamics and forces in a P2P application.

A closer look at existing network simulators

The problem of testing large distributed systems by using simulations is not new and has been widely addressed in the scientific literature. NEST [103], for example, simulates a network within a single Unix process, which has the advantage that no networking is needed to simulate a virtual network. Instead of sending packets from one virtual host to another, messages can be exchanged by passing reference pointer to the shared memory. In addition, debugging the tested application is easier if the instances of the application reside on a single machine. In NEST, the virtual network is accessed by using NEST-specific methods, such as *sendm()*, *recvm()* or *broadcast()* (for sending, receiving and broadcasting messages respectively). Since time-related commands like *time()* or *sleep()* are no longer usable in an emulated environment, replacements are offered. There are, of course, some differences to normal network operation. Since all emulated hosts run in a single process, NEST uses interrupts to iterate through all hosts. This means that virtual hosts get a limited amount of CPU cycles and are then interrupted. Special methods are provided to keep NEST from interrupting critical tasks that must be finished before the next virtual host gets its turn.

Another system, which is a little bit more sophisticated, is EMPOWER [104]. By modifying the network device driver of *nix systems, EMPOWER uses a local network of a few machines to emulate a way bigger network. If a virtual host sends a message to a (virtual) IP address, the simulator must translate this virtual address to the address of the actual computer that runs the virtual destination host. This is done by employing so-called *Virtual Routing Mappings*. Instead of using normal routing tables, virtual routing tables help finding the correct target machine. EMPOWER is even able to simulate packet delays, ranging from a few milliseconds to a second and above. Obviously, if two machines host a multitude of virtual hosts, the physical network link between these two machines is shared by all virtual hosts. The resulting network congestion is a problem for getting adequate results.

There exist many more systems, among them VINT (Virtual InterNetwork Testbed, [105]), ENDE (End-to-end Network Delay Emulator, [106]), One (The Ohio Network Emulator, [107]), Delayline ([108]), JEmu ([109]), and Seawind ([110]). Riley et al. [111] list further systems, such as Opnet [112] (which is able to simulate a few hundred nodes), GloMoSim [113] and the LBNL network simulator (*ns*) [114] (for a few thousand nodes), TED [115] (for tens of thousands of nodes), [116], [117], and the Parallel / Distributed *ns* [118] (for a few hundred thousand simulated nodes).

In the area of network simulation, a lot of problems are still unresolved. Floyd and Paxson [119] and Riley and Ammar [111] list the following common problems of existing network simulators:

- *Scalability:* According to the Internet Software Consortium [120], in January 2003 the Internet consisted of approximately 170 million hosts. It is quite a challenging task to simulate this amount of hosts. The authors of [111] state that simulations with packet-level detail would take too long to produce reasonable results and that the resulting amount of data would be too large. They estimate that approximately $2.9E11$ messages are sent in the Internet per second. Hence, if a simulator is able to simulate one million message per second, the simulation would take 290,000 seconds and about $1.4E13$ bytes of memory for each simulated second. Riley and Ammar further state that by 2008, estimated 300 million CPU seconds would be necessary to simulate a second of the Internet.

  Therefore, the Internet is simply too big for simulation.

- *Network heterogeneity:* Today's Internet is a carrier for a multitude of different protocols. Protocols such as HTTP, SMTP, FTP, IRC, and many more share the same physical links of the Internet. This protocol mix with a lot of interdependency makes it particularly difficult to simulate the Internet.

  But not only the tremendous diversity of protocols is a stumbling block for successfully simulating the network. The Internet itself, which is a composition of many diverse network technologies and administrative domains, is very heterogeneous. Different types of physical links, with different bandwidth and latency, exist. Furthermore, some links are point-to-point while other are broadcast links (e.g. in a WLAN). Routes are asymmetric and the packet loss rate is not deterministic. The congestion at routers in the Internet can not be easily simulated.

  Generally, it is very hard to understand how a large IP network reacts to some protocol or topology changes.

  Another example for the heterogeneity of the Internet are the different implementations of the TCP protocol. Different implementations have different behaviors under certain circumstances, which makes it nearly impossible to simulate such a simple protocol as the TCP protocol.

- *Accuracy:* As explained above, the major problem of simulating the Internet is its scale. Hence, a more simplified model has to be used to simulate the Internet. The next question is whether this simplified model is accurate enough. Simulation and analysis of the Internet can only be applied to a conceived model of the real world. It is restricted to the current form of the Internet, not taking into account what the Internet will look like in a few years. Important aspects like pricing, future killer applications (as was P2P a few years ago) and new devices will shape the future Internet. Furthermore, the Internet does not have a fixed topology but changes dynamically [121].

  It is not clear how to verify the accuracy of a simulation. Since Internet-scale simulation is not feasible, a simplified model of the Internet with just a few thousand hosts and data flows has to be applied. There exists no measure to determine the accuracy of such a model. There is a risk that a too simplified model would neglect key aspects of the Internet.

- *Significance:* Most simulators are subject to limitations in size and complexity. They must not be used to just produce numbers that show the performance of a distributed algorithm. Simulations are rather good for helping to understand the interactions between protocols and hosts. Therefore, it is questionable whether results drawn from simulations are significant enough.

Nevertheless, as stated in [119], this does not mean that simulating networks is futile. It is useful for understanding how protocols behave and to support mathematical models.

Simulating a P2P network

In the area of P2P computing, little is published about network simulators. Notable exceptions are the Query-Cycle Simulator [122] and the NeuroGrid P2P Simulator [123]. The Query-Cycle Simulator simulates a file sharing network such as Gnutella. In a single cycle, the simulator lets all virtual peers issue queries and collect the corresponding responses. After that, statistics of the collected data can be created. For each virtual node, the Query-Cycle Simulator models individual files that are shared in the network. The authors of [122] state that this is important because a random distribution of shared files would not exhibit real-world properties (e.g., it would not create clusters of peer interacting, as it is in the real Gnutella network). Furthermore, it is important to use multiple variations of ontologies and attribute values for meta-data to get a better understanding of the dynamics of the protocol.

Tailoring simulations to the requirements of P2P networks

Why is it necessary to have a dedicated P2P simulator if so many generic network simulators are available? The answer is that generic network simulators are too complex for simulating P2P systems. When testing a P2P application, it is reasonable to test the interactions of as many peers as possible. Furthermore, when testing a P2P application, other factors than network congestion, the simulation of routes or the interaction with other protocols (things that are typically simulated in network simulators) are especially important. It is more interesting, for example, to see how many messages a node has to process in a specific time frame. It is also worth testing how nodes react to changes in the topology, to check what happens if a connection has been dropped, how changing peer uptime, session duration, peer activity levels, and the number of search requests and responses influence the behavior of the P2P system. Typical network simulators do not aim at testing these properties.

For these reasons, a project called *Simix* has been started to help simulating P2P applications on top of Omnix. In Simix, multiple peers can be started in a single virtual machine. Instead of using real socket communication, Simix provides a transport module that translates the sending of messages into a method invocation. Due to the fact that all layers above the transport layer are completely independent from the layer below, no changes have to be made to the Processing and P2P Network layers. One of the biggest advantages of Simix is that not a single line of code has to be changed in the application (which is not possible in normal simulators, as, for example, explained above in the case of NEST).

Each simulated (virtual) host may have an individual connection (with parameterized upload bandwidth, download bandwidth and packet loss rate). It is also possible to start multiple instances of Simix and to connect them to a large network simulator. If a message has to be transported to a virtual host in a remote Simix simulator, RMI is used to transport the message to the destination host.

Simulating routing in the Internet

The number of routing hops between two peers is approximated by taking the Levenshtein distance between two virtual IP addresses. Table 6.1 shows an example of how the number of hops is computed between a peer at address 209.247.228.201 and another peer at address 209.245.127.64. This is,of course, a rough approximation to the real routing algorithm in the IP network. But it is virtually impossible to get an accurate model, as the routing in IP networks is highly dynamic (and we had to start somewhere). We are currently working on a refinement of this model by using virtual routers that build a graph. When a message is sent from one virtual router to another one, the shortest path between these routers is used to convey the message to the destination peer.

Hop #	**Address**
0	209.247.228.201
1	209.247.228.0
2	209.247.0.0
3	209.0.0.0
4	209.245.0.0
5	209.245.127.0
6	209.245.127.64

Table 6.1: Routing example in Simix.

Simix allows the creation of arrays of virtual peers. To increase scalability of the simulator, it uses lazy loading (i.e., a peer is started when it is first accessed or triggered).

The disadvantage of simulators is that there is no possibility for having a UI for each virtual peer running. Hence, peers must be either working autonomous or be triggered by an external automatism. At the time of writing, Simix does not provide such an automatism as it is unpredictable how to access the functionality of the P2P application on top of Omnix. This step requires additional coding from the application tester.

6.5 Omnix in a pervasive environment

Supporting small devices

Computers were once thought of as big expensive machines that were housed in large rooms and that were maintained by groups of skilled technicians. Today, computers not only have moved out of these large rooms and the protection of their keepers, but computing capability has moved into almost every device imaginable. Concurrently with the miniaturization and widespread availability of processors have come advances in mobile computing. Handheld devices such as PDAs (e.g., the new generation devices such as the Compaq iPAQ) and mobile phones (e.g., the Nokia Communicator) have been increasing in popularity and have also been getting more powerful.

The universal availability of mobile devices and their ability to communicate with one another (using communication technologies such as Bluetooth, Wireless LAN and UMTS) is now giving rise to a pervasive or ubiquitous computing environment (see [124], [125], [126]) in which devices autonomously detect each other's presence and attempt to interact with each other in order to provide some routine or other hitherto unimaginable services for humans. The missing piece to making this vision into reality, however, is comparable advances in fundamental software abstractions and technologies.

One important software abstraction that is still required for mobile devices is middleware that provides a higher level access to the networking and communication means provided by handheld devices. In the last couple of years, Peer-to-Peer (P2P) middleware support for handheld computing devices has been gaining attention. This is because the P2P paradigm has many advantages when it comes to ad-hoc mobile communication. The aspect of decentralization in the P2P paradigm makes it a good candidate for this type of communication. There is usually no need for a central server and the communication may start immediately without setting up a specific infrastructure.

P2P middleware systems for mobile devices aim to provide an abstraction between the P2P network and the applications that are built on top of it. These middleware systems offer higher-level services such as distributed P2P searches and support direct communication among peers. Such systems often provide a pre-defined topology that is suitable for a certain task (e.g., for exchanging files).

As described in Chapter 3, choosing the best topology depends on the intended application. This also applies to the field of pervasive computing. Pervasive computing per se is not a fixed use case but an umbrella of a large set of applications, each having its own characteristics, advantages, drawbacks, and – of course – best fitting network topology underneath. It is not very sage to lump together all possible applications by using only a single P2P network topology. Furthermore, it is very likely that multiple applications will run on a single device. It does not make sense that all applications are compelled to use the same P2P topology and it does not make sense either to let every application have its own P2P networking implementation, as resources are typically limited on a mobile device. In addition, different devices possibly provide different APIs for accessing the network. In the worst case, an application has to be adapted to each mobile device it is supposed to run on.

Omnix, with its layered architecture and its support for multiple P2P topologies at the same time would be a remedy for these problems. By only changing the transport modules at the bottom of Omnix, application can run on many different devices without the need to change a single line of source code. In addition, a single instantiation of Omnix would be sufficient to support a wide range of P2P topologies and therefore P2P networks. There is no need that every application comes with its own implementation of a P2P stack. This also helps to save limited resources on mobile devices.

To prove Omnix's ability to run on small devices, a transport module for the Java 2 Mobile Edition (J2ME) has been implemented.

6.5.1 Supporting J2ME

Java 2 Mobile Edition (J2ME) is a framework for bringing Java technology into small devices with network capability, ranging from pagers over mobile phones to high-end PDAs (Personal Digital Assistants). J2ME defines classes (or *configurations*) of mobile devices with similar features. For each class, J2ME defines a configuration of required classes and optional packages.

Currently, there are two configurations defined, each optimized for the processing power, network capabilities, and memory capacity given respectively: the *Connected Device Configuration* (CDC) and the *Connected Limited Device Configuration* (CLDC). While the first is intended for more capable devices such as TV set-top boxes and high-end PDAs (i.e., typically devices with a 32-bit CPU and at least 2 MB of RAM), the CLDC covers devices with even more limited resources, among them mobile phones and low-end PDAs.

On top of these configurations, different profiles can be defined. Each profile specifies which classes must be supported and which are optional. In CLDC, there is one profile currently defined, the *Mobile Information Device Profile* (MIDP, version 2). This profile includes a few utility classes (e.g., for networking and local data storage) and some additional packages that deal with the user interface of the supported devices.

Omnix provides full support for the much more demanding CLDC configuration, as it uses the classes provided by the MID profile to access the network. This is a good example for the necessity of using a component-oriented framework. By simply replacing the transport module, Omnix can be used on different devices. For the other components in the Omnix framework, it does not make a difference which transport modules is actually used. Figure 6.22 depicts an example of the incompatibilities between the Java 2 Standard Edition (a) and the Java 2 Micro Edition (b). They use different APIs to access to the virtual machine's networking functions (in this case, to open a datagram socket).

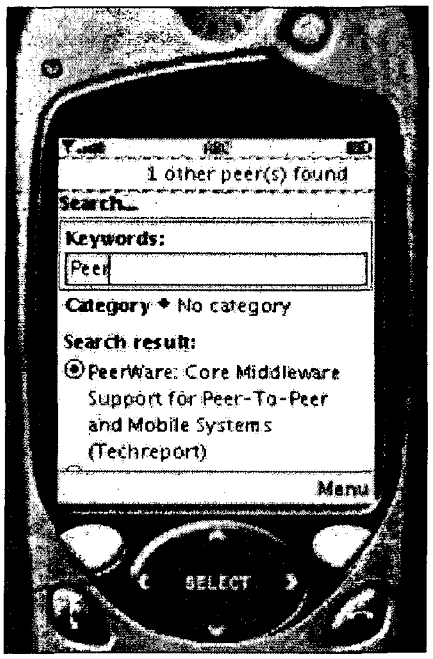
Figure 6.23 shows a real application of Omnix written for the J2ME MID profile. It

```
(a) MulticastSocket multicastSocket = new MulticastSocket();

(b) UDPDatagramConnection dc = Connector.open("datagram://:");
```

Figure 6.22: Opening a socket in the transport module (J2SE vs. J2ME).

allows to search and view BibTex references in a Nimble group P2P network.


1 other peer(s) found

Search...

Keywords:

Peer

Category ✦ No category

Search result:

◉ PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems (Techreport)

Figure 6.23: Omnix running on a J2ME device emulator.

In this chapter we have exemplified how to create various plugins (or modules) for the Omnix framework and discussed theoretical aspects. Chapter 7 gives an overview of applications that are actually using Omnix as the underlying P2P communication infrastructure.

Chapter 7

Applications using Omnix

This chapter gives a short introduction to some of the programs that are using Omnix as the underlying P2P middleware. Two applications written to show the applicability of Omnix and an experimental study of Omnix are presented.

7.1 Simplix – a P2P Service Engine

The first application we wrote on top of the Omnix framework was a service engine (with the name of *Simplix*) that can be seen as a very simple analog to a Servlet engine.

In Simplix, each peer is sharing information in form of files or dynamic services. When a peer searches for information, it can use one or more keywords to search among the meta-data stored at the remote clients. The number or type of services provided at a peer is not fixed. The use of so-called *SimplixServices* (which are comparable to Servlets) allows to extend the functionality provided by a peer.

Figure 7.1 shows a high-level view of the architecture of Simplix. On top of Omnix is the Simplix Service engine which is designed to manage arbitrary *Simplix* services. When a search request is received by Simplix, it passes this search request to each of the installed Simplix services, collects their results and sends back the combined result to the requesting peer. A list of existing Simplix services is given below.

Each Simplix service may be assigned to one or more *categories*. A *category* is simply a keyword like "Multimedia", "Person", or "Document". The list of categories defined in a Simplix network is not fixed and can be determined by the system administrator. It is not necessary that all peers share the same set of known category names (although it is clearly favorable). Categories allow peers to limit the search horizon, eliminating all search results that do not fit the requested categories.

The set of categories in a Simplix network can be hierarchical, meaning that one or more categories can be a subset of a common category. An example is given in Figure 7.2 where the search for an artifact with category "File" returns all search results that have either the category "File", "Multimedia" or "Document". If a peer searches for the root category ("/"), it simply gets all artifacts that match the given keywords, regardless of their category.

Some of the services existing at the time of writing are:

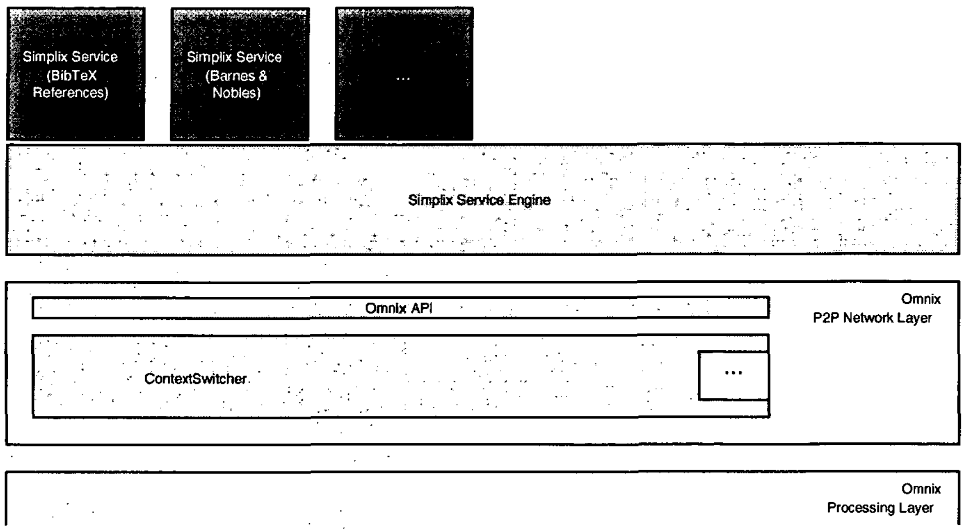


Figure 7.1: Simplix architecture.

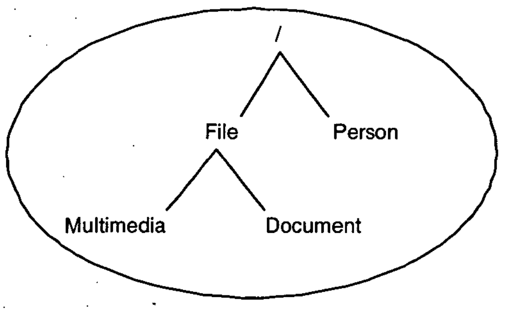
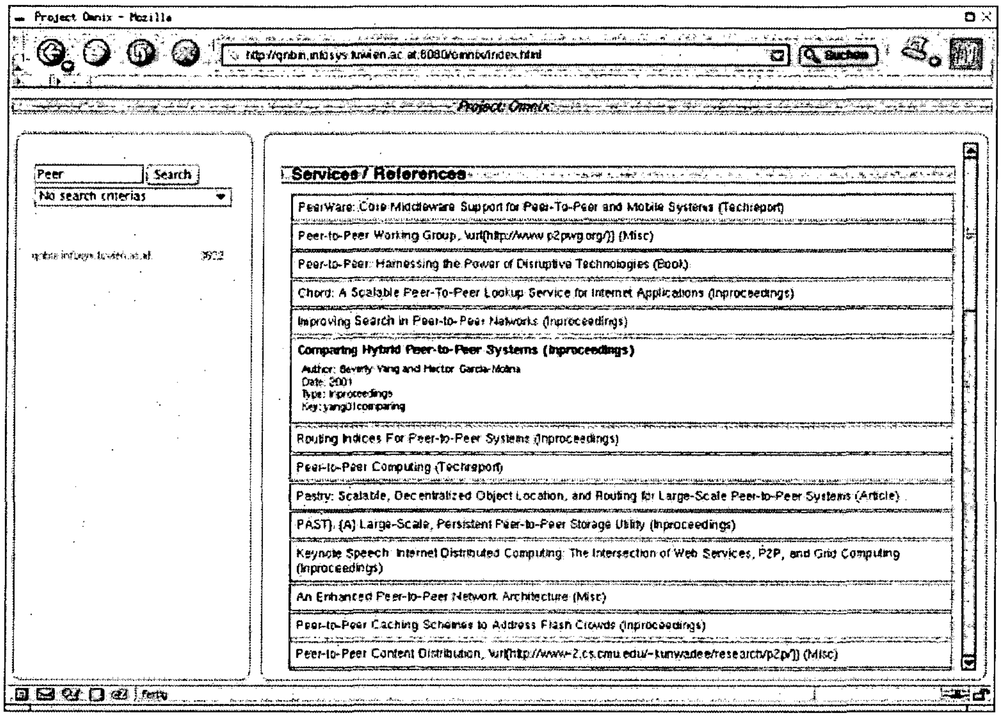


Figure 7.2: Example of categories in Simplix.

- *People:* This service takes a set of *contact* files in a directory and allows other people to search for this information. This Simplix service uses LDIF files (short for LDAP Data Interchange Format, see [127]), which is used by many personal information managers (PIM). The *People* service is useful in a large company when a person with a specific profile is needed (e.g., when searching for people who are experts in Windows security). The *People* service returns a short summary of the contents of the LDIF file together with a URL where the complete file can be downloaded.

- *References:* Since every researcher has her own set of BibTeX references, this service allows to share these BibTeX references with other researchers. It takes a BibTeX file and allows other peers to search for any field in the file and returns the complete matching BibTeX entry. Figure 7.3 shows an example of a search result returned by another peer. Figure 6.23 (page 105) shows this service used on a mobile phone.

- *Books:* This service acts as a proxy for the search facility provided by the online bookstore Barnes & Nobles (BN) [128]. It takes an incoming search requests, parses the keywords and issues a corresponding search request to the BN website. The result returned by BN is then transformed back into a Simplix search result containing the first ten matches and sent back to the requestor.



Services / References

- PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems (Techreport)
- Peer-to-Peer Working Group, \url{http://www.p2pwg.org/}} (Misc)
- Peer-to-Peer: Harnessing the Power of Disruptive Technologies (Book)
- Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications (Inproceedings)
- Improving Search in Peer-to-Peer Networks (Inproceedings)
- **Comparing Hybrid Peer-to-Peer Systems (Inproceedings)**
  Author: Beverly Yang and Hector Garcia-Molina
  Date: 2001
  Type: Inproceedings
  Key: yang01comparing
- Routing Indices For Peer-to-Peer Systems (Inproceedings)
- Peer-to-Peer Computing (Techreport)
- Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems (Article)
- PAST}: {A} Large-Scale, Persistent Peer-to-Peer Storage Utility (Inproceedings)
- Keynote Speech: Internet Distributed Computing: The Intersection of Web Services, P2P, and Grid Computing (Inproceedings)
- An Enhanced Peer-to-Peer Network Architecture (Misc)
- Peer-to-Peer Caching Schemes to Address Flash Crowds (Inproceedings)
- Peer-to-Peer Content Distribution, \url{http://www-2.cs.cmu.edu/~kunwadee/research/p2p/}} (Misc)

Figure 7.3: Web-Interface of the Simplix application.

With Simplix, a peer can search among registered persons, books, BibTeX references and much more with a single search request.

There exists also a Web interface to the Simplix system so that it is not necessary to have the Simplix system installed to search for information (i.e., it can be accessed remotely). Figure 7.3 shows a screenshot of this Web-Interface of the Simplix application. On the left side, the user can enter keywords and select one or more categories to search for. The search result is presented on the right side of the Browser window, grouped by the found categories.

The advantages of Omnix that we experienced when building Simplix was that it can be used on any devices that has a Java heartbeat (Figure 6.23 shows Simplix running on a mobile phone). The implementation on the mobile phone just required to write a small UI that accesses the services provided by Simplix.

Furthermore, we took advantage of the fact that the topology of the underlying P2P network is not fixed. When Simplix was built, a Nimble group was used as the P2P topology (because it was the only available at that time). Later on, we were able to replace this topology by a more scalable wild mesh topology seamlessly (i.e., without changing the code of any of the components or services of Simplix on any device).

7.2 Donare – a hybrid P2P file sharing application

The second application that has been built with the help of Omnix is *Donare* [129]. It is a sophisticated hybrid P2P network that has roughly the same technical structure as the currently most popular P2P network, FastTrack [19]. The system provides SuperNodes to collect meta-data on shared services and files, and to process or forward search requests to other SuperNodes.

Donare is located on top of the Processing Layer of Omnix, thus not using the P2P Network Layer. The reasons for this are that it uses many sophisticated networking techniques such as traffic shaping, load balancing, route optimization, etc. (which is much easier to handle when using the communication infrastructure directly), and to show that an application is not required to use only the top layer of Omnix.

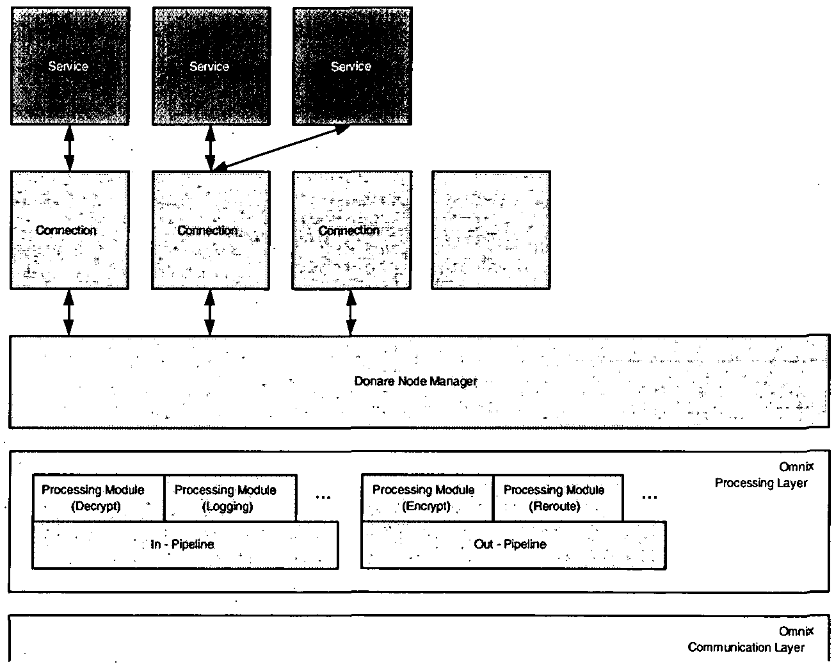

Service
Service
Service
Connection
Connection
Connection
Donare Node Manager
Processing Module (Decrypt)
Processing Module (Logging)
...
Processing Module (Encrypt)
Processing Module (Reroute)
...
Omnix Processing Layer
In - Pipeline
Out - Pipeline
Omnix Communication Layer

Figure 7.4: Donare architecture.

Figure 7.4 shows an overview of the Donare architecture. The *Node manager* is responsible for loading, configuring and managing all attached services (more on services in a moment) and the connections among services. A *Connection* is a point-to-point connection between two services (regardless of whether these services are on different hosts or not). This connection is responsible for providing stable communication to the other communication endpoint. Periodical ping messages ensure that communication failures are detected.

Services can be provided to remote peers, to other services or to the user. A UI *service*, for example, could contact a file searching *service* to search for files in the network (using remote *services*). Services, for example, may open connections to other peers to send search requests. The modularization of Donare allows to add arbitrary services to the system. The services implemented at the time of writing are a UI service and the services necessary to build a hybrid P2P network that allows peers to search and download files. It uses regular expressions to search for meta-data stored at the SuperNodes.

Donare shows that Omnix can be used to build very sophisticated P2P networks. It is planned to redesign Donare in a way that it can be used as a topology module.

7.3 An experimental study

This section deals with a evaluation of the reference implementation provided with Omnix. This experimental study is mainly to provide a better understanding of how the Omnix P2P middleware framework performs. Although the reference implementation of Omnix has not been optimized for speed, low memory consumption or scalability (e.g., the storage of meta-data is done in memory with the help of a single hashtable – not a very good solution once the data exceeds a few megabytes, but sufficient for the purpose of these tests), it is still useful for getting a lower bound of the performance of Omnix.

For the purpose of this study, we chose two different topologies – Nimble group and server-based topology – to determine whether the used topology influences the performance of the system.

#	**CPU** *[MHz]*	**RAM** *[MB]*
1	2000	512
2	1800	1024
3	800	256
4	700	256
5	500	256

Table 7.1: The testbed for testing the reference implementation.

To perform this study, five peers have been created in a local area network. Table 7.1 shows the list of computers used for this test. The connection between these machines is a 100 MBit Ethernet switched network. Java version 1.4.1 has been installed on all of these computers. In the server-based topology, computer # 1 has been used as the server. The slowest computer (# 5) has always been used to issue the search requests. In both topologies, we inserted 1,000 files into the network and ran 10,000 consecutive search requests to get meaningful average values of the time consumed by the Omnix components.

Table 7.2 demonstrates the performance of Omnix when using a server-based P2P topology. With a server-based topology, injecting a file results in a message sent to the server containing meta-data about the file. Search requests are also sent to the central server. These two operations, *inject* and *search*, are evaluated in Table 7.2. It shows how much time the Omnix stack took to process outgoing (*Local stack send*) and incoming (*Local stack receive*) messages, both at the client and at the server. The *Total local stack* value shows how much time Omnix consumed totally to process a request/reply transaction. The *Total operation* value shows the time the complete operation needed to perform the inject or search action (including the time necessary for searching in the database of shared files).

	Client (#5)	**Server (#1)**	**Client (#5)**	**Server (#1)**
	Inject [ms]		*Search [ms]*	
Local stack send	5.82	0.47	6.36	0.98
Local stack receive	2.15	0.38	0.79	1.50
Total local stack	7.98	0.86	7.15	2.48
Total operation	78.06	1.11	24.05	4.74

Table 7.2: Performance of Omnix in a server-based P2P topology.

As Table 7.2 shows, Omnix only needs a few milliseconds to send and receive messages before they are either sent over the network or passed to the application on top of Omnix. The relatively high total operation time of the client is due to the time necessary to send the request and the reply over the network.

	Client (#5)	**Peer (#1)**	**Peer (#2)**	**Peer (#3)**	**Peer (#4)**
	Search [ms]				
Local stack send	6.55	0.89	1.07	2.37	2.74
Local stack receive	2.60	1.17	1.35	2.50	2.60
Total local stack	9.16	2.06	2.43	4.88	5.35
Total operation	333.03	4.19	4.23	9.21	10.71

Table 7.3: Performance of Omnix in a Nimble group.

Table 7.3 shows the same test performed with a Nimble group. In this setting, four peers (#1 – #4) are sharing 1,000 files each. The "client" peer (#5) issues 10,000 search request to all members of the group. The time for injecting the files has not been recorded because in this topology, the meta-data is stored locally in the topology module. Hence, the time for storing this information can be ignored. For the four "server" peers, searching means to receive a search request, process it and send back the search results.

As in the case with the server-based P2P topology, the client consumed only a few milliseconds to process incoming and outgoing messages. The most interesting information in this table is the total operation time of the client peer. Due to the fact that all peers are contacted at the same time, the latency of the network increases tremendously. With all peers transmitting information at the same time, the average search for files requires more than 0.3 seconds.

	Client	**Server**	**Client**	**Server**
	Inject [ms]		*Search [ms]*	
Local stack send	11.76	–	7.07	–
Local stack receive	–	10.84	–	11.02

Table 7.4: Performance measures of JXTA.

In addition, we performed comparable tests with the reference implementation of JXTA

Version 2.0. We measured the time needed to publish an advertisement remotely and to search for an advertisement. The configuration for these tests was an Athlon 900 MHz PC with 256 MB of RAM. We used JDK version 1.4.1_02 and JXTA build *Jan 5 14:37 2004*. The results of these tests are shown in Table 7.4. All tests were performed local on this machine.

We did not collect information about the time responses took in the JXTA framework but the figures presented in table 7.4 are sufficient to show that Omnix is 4 to 5 times faster than JXTA (when taking the results of Peer #3 for comparison).

The purpose of this evaluation was to demonstrate the performance of Omnix (with of two different topologies). We have shown that Omnix only consumes a few milliseconds to perform its tasks, which is a result from our efforts to keep the Omnix P2P middleware as small and efficient as possible.

Chapter 8

Evaluating Omnix

We are going back to the roots, to what the Internet is really all about.
Ray Ozzie about P2P [130]

This chapter analyzes the Omnix P2P middleware framework and the concepts of topology and platform independence, and using a middleware for P2P networks. The evaluation of Omnix is particularly difficult because the P2P computing area is comparatively young and general evaluation methods for P2P systems are just starting to evolve. The main problem in the evaluation of Omnix is that it is – in the strict sense – not a P2P system itself. Since it is only a P2P middleware where, to make matters worse, nearly all components of the system can be replaced, it is quite a challenging task to assess the usefulness of Omnix.

The following sections deal with the question of what can be evaluated and what not. Eventually, they discuss the advantages and drawbacks of Omnix.

8.1 The role of Omnix in the P2P area

Although the Omnix framework is designed to be as versatile as possible, there are of course domains where Omnix is more useful than in others. In the following, those areas where we see a clear benefit of Omnix are listed. It shows where Omnix fits in the world of P2P middleware.

- *P2P Research:* The (intentionally) simple structure of Omnix allows to create new P2P topologies very easily. It enables P2P researchers to take existing P2P application and experiment with different topologies. An idea for a new topology can be implemented and tested this way. Other P2P middleware systems (e.g., JXTA) also provide topology independence, but with a considerable higher complexity (e.g., the JXTA Endpoint Service interface defines 31 methods alone; see page 131).

- *Repetitive application development:* If an application programmer is building P2P systems she should always use the topology that best fits the requirements of the target system. With Omnix, there is virtually no need for learning to work with the different topologies as the interfaces to the middleware do not change. Hence, it is not necessary to learn different APIs.

- *Development for heterogeneous devices:* The Java[1] reference implementation of the Omnix covers a wide range of devices, from PCs over handheld devices to mobile phones. This allows application programmers to use the same middleware on heterogeneous devices thus enabling various pervasive computing applications.
- *Rapid prototyping:* Omnix allows to build P2P applications with just a few lines of code. This enables application developers to build small P2P applications to assess the usefulness of P2P for a specific application. Together with the ability to replace the topology and the underlying network protocols, this provides a powerful tool for building simple P2P applications very quickly.

There are of course a few areas where Omnix does not fit very well. If, for example, a very fast implementation of a P2P network is needed, Omnix would not be a perfect candidate. The reason for this is the component-oriented design of Omnix which will always be inferior (in terms of timing) to a tailored P2P application. Another example would be an application that needs other networking primitives than the ones provided by the Omnix API (see Section 4.2.4.2).

8.2 Meeting the criteria

Before an evaluation of a system can be performed, it is necessary to define with regard to what it should be evaluated. (The question of what *can* be evaluated is not as much as important as the question of what *should* be evaluated in Omnix.) What are the important and novel aspects of Omnix that are worth looking at?

The best starting point is the assessment of whether the requirements for an open P2P middleware system specified in Section 4.1 are fulfilled. The requirements are as follows (for more details see page 38):

- *Platform-independence:* Omnix must run on a large variety of devices with different capabilities. Furthermore, it must not rely on any programming language-related structures or services (such as Java RMI, COM+, etc.).
- *Small footprint:* Since Omnix is supposed to be a middleware that should run on a large range of devices, it is one requirements that is has a comparatively small footprint (in terms of size of code and the usage of system resources).
- *Topology-independence:* One of the key requirements of Omnix is that it is topology independent. This is necessary so that Omnix can be used for a large variety of P2P applications.
- *Openness:* It is important that the APIs used in Omnix are sound enough so that other components can be written and included in Omnix.
- *Testing:* Omnix must allow for easily testing P2P applications. Hence, it must support the usage of P2P network simulations.

[1]It is of course feasible to implement Omnix in other programming languages; see Section 8.2.1

Furthermore, it is worth looking at the default implementations of the transport, processing, topology, and streaming modules provided with Omnix. Since they are not optimized for performance, throughput, low latency, etc. they are merely used to show differences between different topologies (and therefore give proof that different topologies have different advantages and drawbacks) and how the Omnix system itself performs.

8.2.1 Platform independence

To achieve platform independence, Omnix uses a component-oriented approach that allows to replace all parts of the framework that interact with the operating system. An example for such a component is the transport module. It is responsible for accessing the network (i.e., sending and receiving messages). It fully abstracts how the module accesses the network functions of the operating system. Furthermore, from outside the transport module it does not make a difference which type of network is accessed (be it an IP or Bluetooth network). Since these different types of network may have different types of addressing schemes, Omnix uses an abstract *Contact* interface that represents a remote peer. A Contact implementation could also represent a single ID (e.g., a number) along with the ability to lookup the actual address of the Contact in a (maybe central) database. Hence, the network addressing scheme and the network access itself is transparent behind a generic interface.

This is also true for the streaming facilities of Omnix. For every type of stream, a different *StreamProvider* may be used. There is no need for the application or other parts of the Omnix framework to directly access any network functionality of the operating system (or the virtual machine, as it is the case in Java).

The Omnix middleware is–to some extent–stateless, which means that it does not need any memory to save any state. If a peer is restarted, it does not require any information of prior runs. There are, however, conditions where a state might be necessary. If a processing module that ensures that messages reach their target (by resending a message if no response has been received) is inserted into Omnix, it clearly needs state information (where the state is the status of the sent message). Hence, it cannot be guaranteed that Omnix is fully stateless.

The next question is on which devices the Omnix implementation may be used. At the time of writing, Omnix has been successfully tested on desktop computers, on handheld computers (Compaq iPAQ H3660) and on a mobile phone simulator (running J2ME MIDP 2.0, see Section 6.5.1). Since the MIDP specification is very low profile, it runs on virtually anything that provides Java and a connection to the network. Note that this is only valid for the actual implementation of Omnix. Omnix, as a conceptual framework, is not bound to the Java language (as explained above). Omnix itself needs a system that allows to access the network and to parse text messages. Furthermore, it is required that the system supports multi-threading. Hence, it is not feasible that Omnix can be implemented on very small devices such as SmartCards in the near future.

A valid question would be whether it makes sense to provide a platform independent middleware platform as long as the application on top of it would probably not be platform independent. The answer to this question is that it is true that multiple devices would require different implementations of the application. But, the benefit of having a platform independent middleware is that it always provides the same API, the same services and the same communication protocol. Only the application has to be adjusted to the target device (evidently, as the GUI most probably has to change, too). The advantage of having a single system instead of multiple systems just talking the same network protocol is explained in Section 8.2.5.

Although the reference implementation of Omnix has been written in the Java programming language, it is completely independent from it. The reasons for this are as follows: it uses only standard socket networking (or whatever network protocol, such as Bluetooth, is used) to communicate with other peers. It does not use any third party middleware (e.g., CORBA or Java RMI) for communication. Using standard network communication has the advantage that it is very likely supported by a wide range of programming languages.

Omnix uses only plain-text communication to exchange information with other peers. It does not use any Java-related structures, such as serialized Java objects or the like. Every Omnix message can be parsed with a normal text parser. It does not require any conversion or additional libraries.

This does not mean that application cannot send binary or proprietary data with Omnix. The content of the messages is arbitrary and can be anything the application chooses to send. But the message headers (the envelope, so to speak) and all other messages sent by the Omnix framework (and its components) can be easily parsed.

One might argue that in times of existing Common Language Runtimes (see [131] or [132]) supporting different programming languages is not as much as important as it used to be. But, on the other hand, Omnix is also intended to run on very small devices (see next section) and therefore be implemented in special languages that cannot be covered by such Common Language Runtimes.

8.2.2 Small footprint

To be able to run on small devices (e.g., mobile phones, embedded systems, etc.), the implementation has to have a relatively small footprint. The Java implementation of Omnix has a size of approximately 70 kB (note that the implementation of the components included in this package have been optimized to be as small as possible). At the time of writing, a typical mobile phone allows applications to have a maximum size of 64 kB. Hence, the reference implementation is too big for a real mobile phone (not to mention the additional size required by the application on top of it). But it is very likely that this memory boundary will soon dissolve. Then, the advantage of Omnix is that multiple applications can use the same middleware.

Another aspect of a small footprint is the processing complexity of the implementation. When receiving a message of average size, it takes Omnix approximately 0,79 milliseconds until the message is delivered to the application. When an application sends a message, it takes about 5,62 milliseconds until the message has been sent over the network. These values [2] have been recorded on a 500 MHz AMD K6 machine with 256 MB RAM. Although these figures reflect only a reference implementation, they visualize that Omnix only needs a minimum of resources.

[2]In the tests performed, the size of message has been randomized. The average message size was approximately 200 bytes. The figures presented here are average values over a test series of 1,000 consecutive messages.

One of the main reasons for the small size of Omnix is that it does not require any third party libraries for processing messages (e.g., XML parsers). All it needs are the Java functions provided by the J2ME MIDP 2.0 specification.

8.2.3 Topology-independence

Topology independence is one of Omnix's major advantages over other P2P systems. For this reason, it is especially important to evaluate whether Omnix can really be used for any topology, and if, to what extent.

Topology independence can be exploited in two ways: statically and dynamically. In the first case, the application developers uses an arbitrary topology module for her application before it gets deployed. This way, Omnix allows to use the same services, supported networks, etc. without sticking to a specific topology. Topologies could be, theoretically, also exchanged after the application is deployed. But this task is very delicate and is described in more detail in Section 8.2.3.1.

Since it is impossible to evaluate the feasibility of implementing all existing topologies with Omnix, this section uses the classification of P2P systems introduced in Section 2.3. There exist three major classes of P2P network topologies: pure P2P networks (with wild mesh and structured networks), server-based P2P networks, and hybrid P2P networks. On the basis of this classification, for each class of topologies an example topology is picked for evaluation.

In the following, Omnix is evaluated whether it can be used in conjunction with all of the four classes of P2P topologies. This has to be done from two viewpoints: 1) is it possible to write a topology module that meets all the requirements of a topology, and 2) is the Omnix API presented in Section 4.2.4.2 sufficient to utilize the full functionality of the topology module.

- *Wild mesh:* A typical scenario for a wild mesh network is a file sharing application. To make things more figurative, an example for such an application may be beneficial: *Gnutella.* In Gnutella, peers can search for files at other peers, can download them, and can share files with other participants. To maintain connectivity of the system, peers keep open connections to a small number of other peers, sending periodic messages to check whether they are still reachable.

  The question is whether this functionality can be achieved with Omnix. When sharing files, the application uses the *inject()* method. The wild mesh topology module simply stores the meta-data locally. Removing this information is done the same way with the *remove()* method. When an application wants to search for files at other peers, it calls the *search()* method. The wild mesh topology module then sends a search request to a number of remote peers (as an Omnix request). Received responses are passed to the application. If a peer receives a search request, it searches its local database and returns the list of found items. In addition, it forwards the search request to other peers, until the time-to-live limit has been reached. All this can be easily done in a topology module by using standard Omnix messages. If an application has found something relevant, it can download it by using the *StreamManager* to open a connection to the remote host. The wild mesh topology module can send periodic messages to its neighboring peers to check whether they are still online and responsive. It is also possible to pack a list of known peers into a message to disseminate the address of other peers.

Therefore, it is feasible to create a P2P network with a wild mesh topology. A reference implementation of a wild mesh file sharing network has been created.

A special form of the wild mesh P2P topology is the nimble group, where all peers are directly connected (e.g., by a broadcast link, such as a local Ethernet network or a wireless LAN). The implementation of this topology is much more simple because peers do not have to maintain a list of alive peers. Search requests are simply sent to all other peers. A reference implementation for the nimble group exists as well.

- *Structured P2P:* What all structured P2P topologies have in common is that they use some sort of number to 1) identify the artifact shared in the network and to 2) find a path to the peer that most likely holds information about this artifact. Hence, a structured P2P topology module has to be able to create an unique, reproducible identifier from some data (since this is done by the topology module autonomously, this is not a problem). When an application wants to inject data into the network, an ID is created and the data is sent en route to the peer that is responsible for this data (as identified by the ID). This can be done with normal Omnix request/reply messages. A peer has to have an internal routing table. This table holds information about peers and their respective range of identifiers they are responsible for. When a peer searches for information (or has to forward an incoming search request), it looks up the internal routing table and sends the request to the best matching peer. The routing table is maintained by periodically sending lookup and is-alive messages to other peers. Removing files, if supported by the network, can be done by using the *remove()* method.

  The Omnix API provides the full functionality necessary to search in a structured P2P network, to share data in the network and to remove it. In addition, it also allows to check which data is stored on the local peer.

  The only problem that could arise when using a structured P2P topology module is that it is the only topology module that has to understand the meta-data semantically. This is necessary to create the ID number that is so vital for the system. It might be the case that a topology module has to be adapted to the meta-data ontology used by the application. On the other side, this is not necessary if both parties use the same ontology (e.g., as the one that is used by Omnix by default).

- *Server-based P2P:* In a server-based P2P network, injections are done by sending the data to the central server. If a search is performed, the search request is simply sent to the server, which processes the search request and sends back a response to the requestor. There is no third party involved in this communication.

  In this case, two different topology module implementations are necessary. The implementation of the client and the one of the server. If a client peer wants to inject data, it creates a request, packs the meta-data into the body of the message and sends it to the server. The server extracts the data from the message and stores it in its local database. If a client searches for information, it again sends a request to the server – this time with a search expression. For both cases, on the client side the API provides all means to accomplish these tasks.

On the server side, the topology module could be either completely network-oriented (which means it does not provide any functionality to the application on top of if) or it may also provide the functionality to inject and search data. Either way, both the messaging infrastructure of Omnix and the Omnix API provide full functionality for this topology.

Since instant messaging has been provided in Napster (version 1), a server-based P2P topology module might as well implement the methods for subscribing/unsubscribing to channels and to send information to other peers directly.

- *Hybrid P2P:* The hybrid P2P topology is a mixture of the wild mesh network and the server-based P2P network. In a hybrid P2P network, many normal peers are selected to be SuperNodes. Normal peers use this SuperNode as a server, which means that they send injected data to this SuperNode as well as search requests. If a SuperNode cannot fulfill a search request, it forwards it to a neighboring SuperNode.

  All these communication elements have been implemented either in the wild mesh P2P topology module or in the server-based P2P topology module. The election (or selection) of the SuperNodes is done by the topology module autonomously and does not require any application input (or action).

  Hence, implementing a hybrid P2P topology module is feasible in Omnix, with the exception that the application is not able to influence the decision of whether the local peer should become a SuperNode or not. Technically, this drawback is not severe as the topology module can be implemented to configure the system in an optimal way (i.e., by choosing the right SuperNodes). On the other side – especially in times of a DMCA (Digital Millennium Copyright Act) [133] – it is maybe important to disable the ability to become a SuperNode. However, this may also be done by changing the settings of the hybrid P2P topology directly.

What all topologies have in common is that they only provide the search functionality. They do not provide means for accessing the shared service of file. This can be done by using the *StreamManager* or by using the alternative commands of Omnix, such as *invoke()*. The reason for this decision is that Omnix does not have any knowledge about the nature of the shared service (or file). Hence, it does not make sense to provide additional methods for accessing it besides the *invoke()* method and the possibility to open a stream to the remote peer.

It is also a common characteristic that the Omnix API does not provide any means to change the behavior of a topology module. This would be sometimes advantageous (e.g., if the application could advise a hybrid topology module not to become a SuperNode). A possible solution would be to introduce a generic method for changing the settings of a topology module. The problem is that it would create a dependency between the application and the underlying topology module because it is possible that, when switching topologies, the new topology module might not understand what the application "is trying to say". Still, this possibility is an open issue and is considered as future work.

The result is that the Omnix P2P middleware framework is topology-independent (with minor exceptions). It cannot be predicted what P2P topologies the future will bring but it is can be expected that the Omnix framework will be generic (but still useful) enough to provide them. What has to be done when changing topologies and what the ramifications are is explained in Section 8.2.3.1.

8.2.3.1 Changing topologies

The ability to seamlessly change topology without the necessity of changing code in the application using the Omnix P2P framework is generally possible. Over the last five years, evolution in the P2P area has shown that P2P topologies matured over time. It all started with a fault-prone – because having a single point of failure – server-based P2P network (Napster) and continued with an ineffective wild mesh P2P network (Gnutella) which could not cope with the vast number of users. Then came the hybrid P2P networks which have considerable scalability and stability. Finally, scientific P2P system using distributed hashtables emerged, but never hit the big market. This short list is only valid for P2P applications in the file sharing area. In other areas, development is still in its infancy. Groove, for instance, is a very good example for a successful P2P application outside the file sharing domain, but its lack of scalability calls for better topologies that support the application domain of groupware.

When designing a P2P middleware (what Omnix is intended to be), there cannot be any assumptions made on the use case nor the requirements of the application. For a P2P middleware, it is important to be able to adapt to the needs of the application. Thus, it must provide a way to use arbitrary topologies (supporting a fixed pool of varying topologies was not an convincing and future-proof option).

This section deals with the question of what happens when a topology has to be replaced. There are two ways how topology modules could be replaced: either statically (i.e., by changing a configuration file and restarting the application) or dynamically (i.e., at run-time).

Changing topologies statically

The easiest way to change the topology of a network is to change the configuration file of Omnix and restart the application. The Omnix framework then uses the new topology module to communicate with the other peers. This sounds very simple, but has some implications not visible at first sight. Clearly, it is not sufficient to change the topology module in a single peer. It is necessary to replace the topology modules at all peers in the network for the following reasons:

- *Message types:* Every topology module may use its own set of messages exchanged. If a peer changes replaces its topology module, it might be the case that the new topology module issues messages that are not understood by the other peers that still use the prior topology module. For example, in a server-based P2P network, a normal peer would not understand an incoming search request because it does not expect to receive one.

- *Meta-data ontology:* The meta-data ontology and structure used to describe services or files shared may differ from one topology module to another. In a distributed hashtable, a single number is sufficient to search for content but in a typical file sharing network, complex meta-data may be used for describing search requests and shared services. If the topology modules use different meta-data models, they cannot communicate with each other.

- *Routing:* Routing in a topology module not only may but for sure is different from other topology modules. Hence, the overall routing process no longer works and produces unpredictable message flows. Even if topology modules use the same type of messages and share the same meta-data syntax and semantics, it may be the case that search requests cannot be fulfilled successfully because the messages never arrive at their target destination. Furthermore, it may be the case that additional information in messages is necessary for some routing algorithm (as an example, a wild-mesh network needs the time-to-live (TTL) information in messages, which is not present in server-based networks or nimble groups).

Hence, if the topology of a network has to be changed, every peer in the network has to replace its topology module. Otherwise, the operability of the network would not be guaranteed.

The issue of changing topologies raises the question of where meta-data information is made persistent in Omnix. Applications are able to *inject* files or services into the network. But what happens if the application stops and then restarts? Does the application layer have to *inject* all the information again or does the topology module store this information? The answer is that it is clearly the responsibility of the topology module to store injected meta-data persistently. The reason for this is simple. In some P2P networks, topology modules do not store injected information locally but sends this information to other peers in the network (e.g., in distributed hashtables). Hence, the application at the peer that actually holds the (alien) information does not have any knowledge about it. If this peer restarts, the previously stored information would be lost. Therefore, the topology module is responsible for storing meta-data information persistently. Why is this important? If a topology module is replaced by another one, all data contained in this topology modules is lost. Hence, the application must *inject* all data into the new topology module (i.e., in the newly established P2P network).

Changing all topology modules in a P2P network concurrently might be a demanding task, to say the least. It is only possible in a comparatively small network, such as a P2P network in a small company. In large P2P networks (e.g., with several thousand peers up to Internet-scale networks), this is not feasible. In this case, it would be necessary to automate the task of changing the topology module.

Changing topologies dynamically

Supporting dynamic updates (or replacements) of topology modules is clearly favorable in medium to large P2P networks. It means that the peers in a P2P system collectively detect that a new topology module has to be installed. Topology modules could then be downloaded or automatically propagated through the network and then installed at every peer. This may sound like a straight forward process, but comprises some challenging problems.

The most important question is how the system "detects" that there is a necessity for updating or replacing a topology module. There are two possibilities: either a system administrator responsible for the P2P network decides to change the topology or the system detects this need in an autonomous way.

The first solution is easier and more feasible than using an autonomous network. If the system administrator wants to use a new topology module, it simply informs all peers (in a P2P way, of course) that they have to install this topology module. After a short while, a peers should have received the message and installed the new topology. Reasons for a topology change might be a lack of scalability, the addition of new services, the need for increased bandwidth, etc.

The latter solution (having an autonomous network) implies that all peers in a Internet-scale P2P network come to a consensus about the need to change the topology module. In a large P2P network it is clearly impossible to reach an agreement with all peers. Hence, only a subset of peers must be able to build a qualified authority that is allowed to change the topology of the network. Some P2P networks employ special peers that watch over the functionality of the P2P network itself (such as the FastTrack network, where peers in a layer above the SuperNodes control the SuperNodes). These peers may use some sort of election algorithm to decide whether another topology should be applied. This is theoretically correct, but is almost impossible in practice. How do peers determine the state of all peers in a – by definition – highly decentralized network? How can it be measured automatically when a new topology should be applied; how are the thresholds defined? Even if it is possible to determine that a new topology module should be used because the old one is not sufficient enough: Which topology module should be chosen to solve the problems? If a peer is told to update its topology module: how does it know that this update request is authorized (and not issued by a single, malicious peer)? Since all these questions are not answered, autonomous P2P are very unlikely in the near future. Hence, the only feasible way is to have a central authority that has control over all peers in the network.

But there are additional problems ahead. How does the word get to every peer in the network? Peers are not required to be online all the time. If the request to update the topology module is not received by a peer, how does it know about it? A possible solution would be to periodically contact a central server to check whether important update messages are pending. Another solution would be that peers are storing these requests and send them periodically until they get acknowledgments from all peers in their contact list. This is definitely not the best solution because peers may vanish for a long time. In addition, it might be the case that the network is separated, which means that the message would not reach all peers in the network.

Anyway, since a new topology module would use another protocol name or, at least, another protocol number, peers with old topology would then be forced to update their topology module to be able to communicate with the other peers.

If a peer changes its topology, the information stored at this peer is lost (because it is stored in the topology module). If this is the case, the Omnix framework must inform the application that it has to re*inject* the data into the network.

8.2.4 Openness

Omnix has been designed as an open framework with exchangeable topologies. One of the main reasons for this approach was to provide maximum flexibility in how the middleware is used. Furthermore, it makes it possible to improve individual components over time without touching the other components.

The main *technical* characteristics of a P2P system are: the protocol used, the topology of the system (including the routing algorithm) and the ontology of the meta-data. This section deals with the question whether Omnix is fully flexible and extensible with respect to these characteristics.

The protocol (be it UDP, TCP, Bluetooth, etc.) is arbitrary selectable in Omnix. The notion of transport modules allow to use any protocol [3] Since it is not necessarily the case that other protocols use a URI-schema for addressing resources (or peers), Omnix uses a general *Contact* interface as a reference pointer for remote peers (in most cases, this Contact implementation would be an IP address). The main shortcoming of having the abstraction of the transport layer is that there cannot be any proprietary access to the transport module (via the normal transport API). If, for example, an application wants to influence whether a TCP transport module keeps connections open (instead of tearing them down after the message has been transmitted), it is not possible because there cannot be any standardized way of communicating this information. The reason for this is that this kind of information would be unusable (and maybe unintelligible) for other transport module than the TCP module. If this customization is inevitable, the transport module could provide proprietary methods for modifying the module. Another solution would be to create a generic method for changing the settings of a transport module. This, however, has the same negative impact of creating a dependency between the upper layers and the transport module. A solution to this problem is subject to further research.

The structure of a message determines what information is conveyed in the message. A very simple message structure (e.g., a Gnutella message, where only a small set of information could be stored) is always inferior to a more complex message structure (say, for example, an XML message) but has the advantage of being smaller and easier to process. Depending on the use case of the P2P application, different message structures would be more favorable. Therefore, Omnix does not use a fixed message structure but lets the application programmer to change it arbitrarily.

In addition to the structure of a message, it is also interesting to evaluate the ontology of the content of an Omnix message. While the structure of the message represents the *envelope* of the content of a message, the content itself is the information (with all its semantics) provided by the application. The standard Omnix message allows arbitrary bytes to be placed in the content. Hence, it is possible to send plain-text messages as well as serialized objects. It is further possible to replace the complete message object by a new one that allows other forms of content (e.g., a real-time media stream – whether it makes sense or not). In Omnix, the application programmer can use any arbitrary ontology for the data transported.

The ability to use the Omnix middleware as a framework where third parties can contribute transport modules, processing modules or topology modules was one of the design

[3]Although it is very likely that other modules than TCP or UDP will not be needed, as almost every other protocol supports the conveyance of IP packets.

goals. Whether this has been achieved is tightly coupled to the evaluation of the usability of the components in the three Omnix layers in question.

Since Omnix uses well-defined interfaces for all three layers (including additions like the StreamManager component), it is very easy to include additional components (as long as they adhere to the interfaces, of course).

But Omnix is not limited to the replacement of components only. It would also be possible to replace everything below the Omnix API by another P2P system. As an example, a Gnutella peer implementation (just as an example, any other P2P system implementation could be used, e.g. JXTA) could use the Omnix interface to communicate with the application. In this example, there would be nothing left but the Omnix API. The advantage of this is that the Gnutella implementation could be easily replaced by another system without changing the source code of the application on top of it. Section 8.2.3 shows that the Omnix API is adequate for this purpose.

8.2.5 Testing

The theory on how to test a P2P application is covered in Section 6.4. Using mathematical methods for testing an application using Omnix cannot be done without having the application at hand. For each combination of transport, processing, and topology modules, along with the application on top of it, different mathematical models apply. Hence, it is not possible to provide any useful means for mathematically testing an Omnix P2P application.

What remains is testing an application by simulating a large network of virtual peers. By replacing the transport module of the Omnix framework, it is very easy to create a virtual network. Multiple instances of the same application could be run on a single computer, connected by the same network-simulating transport module. Omnix provides a testing tool called *Simix*, which translates network connections into method invocations at other instances of the application. More information on Simix is given in Section 6.4. At the time of writing, Simix is a working prototype.

8.2.6 What cannot be evaluated?

The evaluation of the framework and the evaluation of the actual implementation have to be separated. The first one deals mostly with concepts and methodologies and those parts of the implementation, that are not replaceable components. The latter is to assess the soundness and usefulness of the implementation at hand. This includes all kinds of modules such as transport modules (e.g., for UDP, TCP, etc.), processing modules (e.g., transactions, logging, etc.), and topology modules (e.g., server-based, wild mesh, etc). These modules include also stream related modules (e.g., the StreamProvider) and message related modules (e.g., the MessageParser).

The second part, the evaluation of the application, cannot be done generally because Omnix is, after all, a framework. Hence, it is designed so that many parts of the system can be replaced by other components. Those modules that are used by default by the Omnix middleware framework have not been optimized for latency, throughput and other evaluation criteria used in distributed systems. Furthermore, they can easily be replaced and therefore make a general evaluation useless. By replacing one or more components, a completely different P2P system with different properties and behavior is created. [4] However, Section 7.3 provides some evaluations using these reference implementations but they are merely performed to demonstrate the performance of Omnix with different topologies.

So, what exactly cannot be evaluated? The following list answers this question in more detail:

- *Topology implementation:* It is not the aim of this evaluation to compare the effectiveness of different topology module implementations. Note that this only refers to the implementation of topologies. It is quite within the scope of this dissertation to compare different topologies but this does not include reference *implementations* of these topologies.

  Hence, this evaluation does not include figures such as the number of messages exchanged between peers (globally or during a session) because this value might even differ between two different implementation of the same topology. In a wild mesh network, for example, the number of pings per second, the time-to-live of a message or the number of open connections to other peers depend on the implementation. Another example is the size of a routing table, as this also depends on the implementation and can vary, even if two implementations have the same topology. Furthermore, the efficiency of loop detection algorithms and the avoidance of repeated messages is also not influenceable by the Omnix framework.

- *Size and structure of messages:* The size and the structure of the information transported in messages sent through Omnix is not determinable. It depends completely on the ontology used to define the structure of the message content. Omnix does not impose any ontology but uses interface abstractions to access the content of messages. What theoretically could be evaluated is the minimal size of a message as an empty message still needs some information (e.g., the address of the recipient). On the other side, the structure of the message itself is also not fixed (e.g., by using alternative *Message* and *MessageParser* components).

- *Usefulness of the meta-data:* A major aspect of any P2P network is its use of meta-data to describe the content (e.g., files, services, etc.) shared. If a P2P network uses a poor meta-data ontology (e.g., such as the one used in Gnutella where only the filename can be used to search for files) there is less a chance to find the desired content.

  Omnix does not use any kind of meta-data as it is only the carrier of information. It does not have semantic information about the content transported between peers. The creation and parsing of meta-data lies within the responsibility of the application on top of Omnix. Hence, there is no way how Omnix could be evaluated in respect to anything related with the ontology of meta-data.

- *Scalability:* The scalability of a P2P system mostly depends on the topology used (as shown in Chapter 3). As Omnix is topology independent, it cannot evaluated to which extend the P2P framework scales. Scalability is tightly coupled with the system's

[4]This is like evaluating the performance of an operating system without caring what kernel modules are installed.

ability to distribute traffic among the participating peers, according to their capabilities and the distribution of the content (among other things). These aspects are not specified (and therefore not influenceable) by Omnix.

- *User activity, session duration, etc.* For the evaluation of a P2P network, it is important to have information about the typical user activity, the average session duration or the number of leechers [5] and many other things that influence the scalability, usefulness and performance of a P2P network. These figure depend on the P2P application on top of Omnix and the engagement of the P2P participants. Since the type of application is not imposed by Omnix, these figure cannot be evaluated in Omnix. From this it follows that Omnix cannot be evaluated as a normal P2P application.

What cannot (and should not) be evaluated at all is the application on top of Omnix because it is a generic purpose middleware which does not impose any use case on the application.

8.3 Comparison to other middleware systems

This section discusses those areas of distributed computing that are related to P2P networking in general and P2P middleware systems in particular. Of the distributed system concepts that are related to P2P, we identified mobile agents and overlay networks as being closest.

8.3.1 Mobile agents

Mobile agents have a similar concepts as P2P. It consist of a network of independent nodes that both consume services provided by others and provide services themselves. There is (technically) no need for a central coordination of the participating nodes of the network. Nodes can search for information (or services) provided by other nodes and, if desirable, access these services. Hence, mobile agents are P2P systems (if no central authority is needed).

The major difference between P2P and mobile agent systems is, that mobile agent systems do not send static messages to the other nodes in order to find or access services but send little pieces of executable code that runs on the target machine (i.e., where it accesses the services provided locally). The resulting distinction between P2P and mobile agents is that:

- Mobile agent perform their own search on the data provided by the hosting agent system. Therefore, mobile agents are much more flexible in searching the network for data. It is not restricted to a specific search protocol or ontology of meta-data. With this flexibility, mobile agents could perform complex search queries over multiple nodes.

[5]Leechers are participants in a P2P network that only consume content or services shared by others but do not share anything themselves.

- It is possible, that the routing algorithm is no longer implemented in the node but on the mobile agent itself. This would enable mobile agents to find their path through the network that best fits the requirements of the assigned task. In some P2P systems, this is not possible as the topology of the system imposes where the agent should travel to (e.g., in distributed hashtables). It also implies that the agent hosting system provides information about the communication links available.

One of the drawbacks of mobile agents system, which is probably the reason why mobile agents never hit the big market, is security. In P2P systems, static messages are sent that are parsed and processed by each node. In mobile agent networks, executable code is transferred and executed locally on a peer. There are a lot of security problems related to that (e.g., the danger of malicious code, the problem of safe resource allocation to agent processes, etc.).

An example for a system that builds a bridge between P2P and mobile agents is Anthill [134]. An anthill, a complex adaptive system (CAS), is used as an metaphor for this mobile agents system. A node is called a *nest* in the Anthill system. A nest receiving a request from the local user creates one or more *ants* to process this request. These ants are autonomous mobile agents that travel through the network to process the request. The service provisioning is delegated to ants.

A nest consists of three components: the *ant scheduler*, the *communication layer* and the *resource managers*. The ant scheduler is responsible for running ants on the local system securely (i.e., in a sandbox) and in a fair way. The communication layer has the task to find new nests (i.e., peers) in the network and to move peers to other nests. The resource manager offers the resources and services provided by the nest (e.g., the file system of CPU cycles).

When a task is assigned to one or more tasks, these ant travel through the network until the request has been successfully processed or the TTL (time-to-live) has been reached. Ants do not communicate directly but can leave information at a nest that can be accessed by succeeding ants. Routing is performed by the ants. The communication layer of a nest manages a set of neighbors that can be accessed by ants. An ant can then choose the next hop on its journey.

The communication layer is the lowest layer of the Anthill architecture. The access to the network, the detection of new peers (by using its own routing algorithm) and the detection of unreachable peers are combined in this single layer.

8.3.2 Overlay networks

An overlay network is a network on top of another network. An example for such a setting is any existing P2P network. It creates a network of peers on top of the IP network. Typically, the nodes in the upper network are connected independently of the underlying infrastructure (an exception is, for example, PAST), which means that a connection between two nodes in the upper layer require two or more connections between nodes (e.g., routers) in the lower layer. The reason for this new layer of network is that the nodes can be connected based on other criteria than physical connectivity. An overlay network could, for example, connect nodes based on the distance between their unique IDs, thus creating a distributed hashtable. Nodes could also be connected according to their geographical distance (which not necessarily relates to their underlying network proximity), as done in the FastTrack network. It is the node content but not the location within the physical network that is the decisive factor for the decision which nodes to connect.

Older P2P networks such as Gnutella, which use flooding algorithms for searching among the peers, do not fully utilize the advantages an overlay network could provide. The main purpose of distributed hashtables, on the other side, is to build an optimal overlay network that routes messages most effectively to the nodes where the desired information resides (which introduces some other limitations).

The main difference between an overlay network per se and a P2P network is that the nodes in an overlay network may be part of a fixed infrastructure. A good example for this is the MIT RON (Resilient Overlay Networks) project [135]. It builds an overlay network over the IP network with the aim to improve robustness of communication link. If, for example, a communication link between two university breaks down, the RON system tries to find another route via a third party, thus bypassing the failed link.

8.3.3 Peer-to-Peer middleware

Chapter 3 discusses most of the P2P systems currently available by breaking them down into groups of P2P systems. This section gives an overview of the P2P systems that are explicitly targeted at being a P2P middleware. They do not have a specific use case (e.g., file sharing) nor is their number particularly high.

MoCha

MoCha [136] (short for Mobile Channels) provides simple communication mechanisms for mobile hosts. Two hosts are connected by a channel. Hosts can create channels, write into channels, read from channels, and finally terminate a channel. This channel acts as a communication buffer between the two endpoints. It does not require a central authority to maintain these channels. Each endpoint has a reference pointer to the other endpoint's part of this shared buffer. If a host moves from one location to another, existing buffers are not moved but a new buffer is created and linked to the previous buffer. The following example demonstrates how this channel linking works:

Imagine two peers, A and B, linked together by a buffer X. If peer B moves to a new location, it creates a new buffer between the new location and the previous one (but does not inform A about the location change). When peer A wants to send a message to B, it first tries to write into the buffer of B at the former location. Before doing this, A detects that B is no longer available at the previous location and follows the pointer to the new location of B. A immediately stores B's new location.

MoCha does not provide any higher level services. It is a very low level communication infrastructure. It does not provide any mechanisms for finding other peers or for searching among other peers. Hence, there is no overall topology or interconnection between peers.

eComP

The extrovert-Computing Platform (eComp, [137]), provides a very promising communication infrastructure for tangible, communicating devices (so-called *eGadgets*). It is part of the EU-funded Disappearing Computer Initiative [138].

It creates a pervasive environment by providing communication facilities for passing messages between peers. Each eGadget (which represents a device) has a set of plugs (or, connection points). Two peers can interact by creating a connection between two plugs (thus, creating a *synapse*, which is comparable to the concept of pipes between two peers).

A combination of eGadgets creates a *Gadgetworld*. An example for a Gadgetworld could be a room, where each device (e.g., a lamp, a light sensor, the refrigerator, etc.) could be an eGadget. The main target of eComP is to combine these individual eGadgets to a single service, which has a higher value than the sum of all individual services.

In the room example (where a light sensor and a lamp are part of the Gadgetworld), the light sensor might use the plug of the lamp to switch it on when it becomes too dark.

For this P2P system, it is crucial, of course, to find other eGadgets in the nearby. This is done in two ways. First, an eGadget broadcasts a message to all those devices that are within its communication range (depending on the type of communication used – e.g., Infrared, Bluetooth, WLAN, etc.), this could range from a few meters to a few hundred meters). To improve the results, it can also issue a search request which then is forwarded by all peers in the nearby, hence creating an indirect link between those devices that are not able to communicate directly (often called a *scatternet*). The scalability of this system is therefore very limited.

eComP uses XML as the carrier of information (see Section 5.5 for the ramifications of using XML). It uses a very small subset of the possibilities of XML and therefore does not require a full XML parser. The protocol stack of eComP is as follows (top-down): the interface layer, the resource management layer, the routing layer, and the physical layer. It supports various platforms, such as PDAs, mobile phones, etc. The implementation of eComP is compatible with the Connected Device Configuration (CDC) of J2ME (see Section 6.5.1), which is not as restrictive as the CLDC configuration.

Mobile MOM

The Mobile Message Oriented Middleware (Mobile MOM, [139]) provides a messaging middleware for mobile hosts. For this purpose, a central *Message Broker* is used that stores all messages that are to be delivered to mobile hosts. Once the mobile hosts connects to this message broker, it can download all waiting messages. To bring the message nearer to the client, *Mobile MOM message agents*, are located at the base-stations that services the mobile host. This combination of mobile host and message agent allows to adjust the exchange of information to the properties (e.g., low bandwidth) of the link between the base station and the mobile host.

Mobile MOM is more a client/server architecture than a P2P system. There is no direct communication whatsoever between two hosts. The communication completely depends on the central server.

Virtual Plant Protocol

The Virtual Plant Protocol (VPP, [140]) is mainly aimed at industrial plants, where sensors, machines, etc. are automatically combined to a P2P system. It is, however, not limited to this purpose.

VPP can be used as a communication middleware between Virtual Industry Devices (VID). These devices are not necessarily connected by a IP network (in fact, it is very unlikely) but often use alternative networks, such as the Controller Area Network (CAN, [141]) or a fieldbus. Hence, it cannot rely on a specific addressing scheme. Therefore, it uses a generic addressing scheme that can be used in arbitrary networks.

The Virtual Plant Protocol does not use any central authority for maintaining connectivity among the peers but is self-configuring, which means that routing tables are created dynamically. Every peer can request information about other peers in the network. If a peer forwards a message, it automatically checks the content of the message for the existence of other peers. The topology created by the Virtual Plant Protocol is not scalable, but it is sufficient for the intended purpose.

If a peer sends a message, it automatically includes the route to the destination peer (so that intermediary peers do not have to have knowledge about the route to the destination peer). Along the path from the sender to the receiver, each peer in between adds its address to the message, so that a response message takes the same way back (thus assuring that the response will find a way back). The message body follows the SOAP specification. Hence, it uses XML for structuring the content of the messages.

XMIDDLE

XMIDDLE [142] is a P2P middleware for mobile hosts. It does not assume the existence of a fixed network or a central authority. It connects peer directly, peers are not used to forward messages to other peers.

In XMIDDLE, each peer organizes its content in a tree structure (i.e., XML). It provides primitives for operating on these trees and to share branches of them. If a peer wants to access the content of another peer, it connects to one of the shared branches. This is comparable to the mounting of a remote file system share. If a remote tree branch has been "mounted", the peer can read and manipulate the data offline. XMIDDLE provides the mechanisms for reconciliation when the mounting peer and the owner of the branch are again connected. If a peer wants to modify an alien branch and has a connection to the owner of that branch, it requests the owner to perform the modification. After that, the owner informs all connected peers about this change. The modification itself is represented by using XMLTreeDiff [143]. For specifying the links between peers, XMIDDLE uses standard XML techniques, namely XLink [144] and XPath [145].

XMIDDLE provides the primitives *Connect*, *Disconnect*, *Link*, and *Unlink*. It does not provide a functionality for searching content in the overall data tree.

iMobile ME

iMobile Micro Edition [146] – which is based on the iMobile Standard Edition – provides

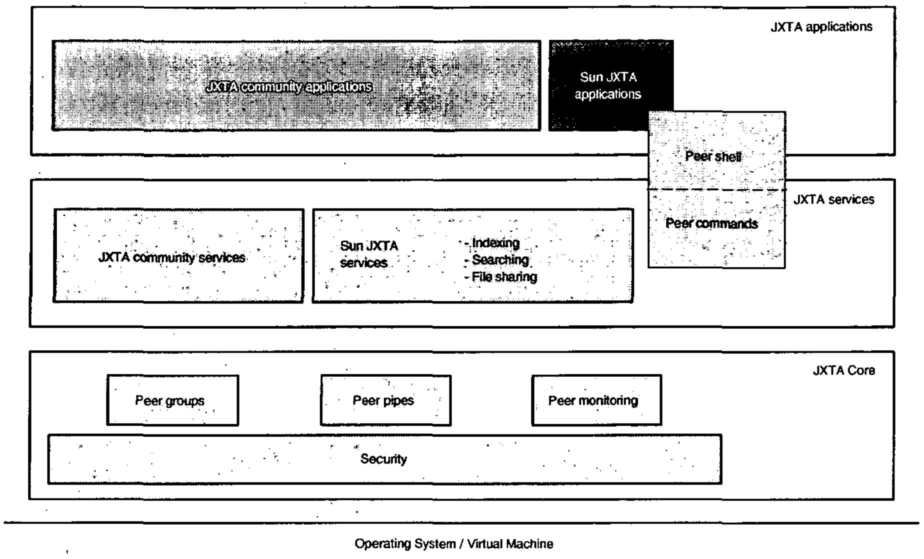


Figure 8.1: The JXTA architecture.

a communication platform for mobile devices.

It consists of a small service platform on each mobile device (consisting of so-called *devlets* and *infolets* for user interaction and service provisioning respectively). Each device has an inbox and an outbox queue. When a device sends a message, it puts it into the local outbox queue. When it connects to the network, it synchronizes with a central server, getting all waiting messages while sending those in the outbox to the central server.

This system uses a server-based P2P topology. The reason why this is not really a P2P system is that all communication is performed via the central server. In Napster, the central server was used for searching only – the download of the actual data has been performed directly between the peers. However, it is also planned to integrate direct communication between devices (with broadcast or multicast) into the system where a central authority would no longer be necessary.

The implementation of the iMobile ME for Palm handheld devices is based on the CLDC configuration of J2ME.

JXTA

The only other P2P middleware that is closely related to Omnix is Project JXTA [2][147], which was initiated by Sun Microsystems. JXTA itself is, as Omnix, primarily a system architecture, but also an actual implementation.

Figure 8.1 shows the architecture of JXTA on a conceptual level. It can be broken down in three layers: the *core* layer (which is mainly responsible for establishing the P2P network structure, the communication, etc.), the *services* layer (which provides P2P functionality such as indexing, searching, and file sharing) and the *applications* layer (where the P2P application resides).

The reference implementation architecture, which is far more complex, proposes six protocols:

Endpoint Routing Protocol: helps peers to route a message from the sender to the receiving peer.

Peer Resolver Protocol: is used to send queries to other peers within the group.

Rendezvous Protocol: transports messages between peers in a peer group.

Peer Discovery Protocol: permits to discover any published peer resources within a peer group.

Peer Information Protocol: provides a set of messages to obtain a peer status information. This protocol can be used to assess whether a peer is still available.

Pipe Binding Protocol: To create a virtual channel between two peers, the Pipe Binding Protocol creates a pipe between them. A pipe is a virtual channel where data can flow in one direction from the sending peer to the receiving peer.

At all levels, JXTA uses XML for the communication between peers. To reduce the size of messages exchanged, it allows to binary encode the messages. When two peers want to exchange information, they create a *pipe* for communication. The specification states explicitly that pipes are not reliable and not connection oriented. To get streams in JXTA, a project called *p2psockets* [83] has been created. It has the goal to reimplement Java sockets on top of JXTA.

In JXTA, peers are organized as groups mainly to limit the propagation of queries (among other reasons). JXTA does not provide any means to manage membership in a group (such methods have been removed from the specification).

The scalability of JXTA cannot be assessed as it does not enforce any strategies for discovery.[6] However, the JXTA system relies on the discovery protocol every time 1) a message is sent, 2) a peer binds itself to a pipe, 3) a peer is looked up, and 4) an advertisement is looked up. Hence, the discovery protocol is used very often.

Comparing JXTA with Omnix

JXTA requires every peer to parse and generate XML messages. This may be a serious problem on computing devices with resource limitations such as mobile phones and Personal Digital Assistants (PDAs). (Section 7.3, presents an experimental study of Omnix where performance measures of JXTA are also shown for reference.) [148] gives a short overview of the main problems of using JXTA with J2ME. The verbosity of XML and the need for a XML parser makes it a bad candidate for devices with low CPU power and slow network connections. In order to cope with the problems JXTA has on handheld devices, the JXTA for J2ME project [26] tries to use normal JXTA peers as proxies to JXME peers to take over some of their responsibilities.

The most important difference between JXTA and Omnix on a conceptual level is that JXTA does not provide any abstraction of the underlying network primitives, which is also reflected in the reference implementation, as the following example shows: components

[6]JXTA version 2.0 uses a hybrid topology for this purpose.

of higher level layers make use of packages and classes, which are not provided on J2ME devices. Hence, if JXTA has to be used on a J2ME device, many parts of the system have to be replaced, instead of just the lowest layer.

JXTA provides a large set of services (e.g., security), which are attractive to high-level P2P application programmers (i.e., normal users). The aim of Omnix is to provide interfaces for the lower layers of the OSI model. The advantage of Omnix is that its implementation is much thinner than JXTA (e.g., the implementation of the Endpoint Service Protocol has twice as much lines of codes as the complete Omnix implementation). It is, of course, possible to replace most parts of the JXTA system, but this comes with a high level of complexity (e.g., the Endpoint Service interface defines 31 methods combined). Omnix not only enables application programmers to replace all layers of the system but also aims to reduce the complexity for this task[7].

Groove

Groove is a collaboration software based on the principle of a shared workspace, where all members of a group (i.e., those in the workspace) share the same view. Tools are used to operate in this shared workspace. Typical tools are a shared browser, a shared drawing board, or a file archive. The tools can be extended by third parties.

For the communication between peers, XML is used entirely. In addition, all communication in Groove is encrypted. Whenever a user changes something within the workspace, all other members of the workspace get informed about this change. If a peer works offline, it stores all changes and commits them en bloc. When changes are made while the peer is online, this information is transmitted to all other peers in real time. Hence, they see the changes as they are made, allowing people to collaborate on the same artifact. Since this can be very costly, Groove also uses the so-called "Asymmetric Files feature", which does not automatically upload files to all members of the group if a peers places it in the shared workspace.

Therefore, all peers of a workgroup communicate with each other directly and individually (as the transmission between two peers is encrypted point-to-point). Groove provides servers that are used to detect new peers in the network and to help peers with lower bandwidth to distribute changes. These servers are also used to store content if one or more peers are offline (or not reachable – possibly due to a firewall) and therefore cannot see the changes made at that time.

It provides higher level services, such as distributed searches, workflows, offline working, and much more. Groove is also planned to be a "Web Services Access Point", which means that it exports its functions as web services (using the SOAP protocol). Several servers are available for Groove: Relay Server, Enterprise Integration Server, Enterprise Management Server, and an Audit Server.

Groove is targeted at small workgroups, as the communication takes place between all peers. With the increasing number of workgroup members, the communication overhead increases tremendously. Hence, it does not scale very well. At the time of writing, Groove is

[7]As an example, version 2 of the JXTA implementation, which has been released in mid 2003, now uses a hybrid P2P topology. Thus, resolver queries are no longer forwarded to edge peers in the network.

only available for the Windows platform.

8.3.4 Overview of middleware comparison

Table 8.1 and Table 8.2 compare the various P2P middleware systems with Omnix. At the time of writing, no information about an implementation of some of the P2P middleware systems has been published. Hence, only a limited number of comparison criteria apply. Hence, all answers marked with an asterisk are assumptions (based on possibility, not on probability) that could not be verified at that time.

The following evaluation criteria have been chosen:

- *P2P middleware:* It is a real P2P middleware or just a normal communication middleware? As a litmus test, it is not a P2P middleware if two communication endpoints have to know each other before the middleware is used.
- *Higher level P2P services:* Does it provide other services than message passing? Typical examples for higher level services are distributed searches and the access of shared services or content.
- *Support for mobile hosts:* Does it support the mobility of peers? This is usually the case when the system allows a host to move during an ongoing communication.
- *Support for small devices:* Is the system theoretically running on a small device? If it uses XML or does only run on Windows, this is usually not the case.
- *Topology independent:* Does it support arbitrary topologies?
- *Server-less:* Does it require a central server for the communication? Hence, a P2P middleware is considered server-less if the server is not absolutely necessary.
- *Programming language independent:* Is it possible to implement the system in other languages than the one for the reference implementation? This is usually not the case if programming language specific mechanisms have been used (e.g., RMI, DCOM, etc.)
- *Platform-independent:* Is it possible to run the middleware on various platforms (i.e., devices)? This requires that the implementation is not bound to a specific hardware or operating system.
- *Scalable:* Is the middleware scalable? If a system uses a central server, it usually not as scalable as a fully decentralized system. If a middleware system is not P2P, the question whether it is scalable is futile (and therefore marked with the †symbol).
- *Network/protocol independent:* Does it support different communication protocols and networks? Typical examples are Bluetooth, TCP/IP, UDP/IP, etc.
- *Support for testing:* Does it allow to test the application in a virtual P2P network without changing the application?

To summarize the evaluation presented in this chapter: For each of the requirements defined in Section 4.1, we discussed how and to what extent Omnix addresses it. The evaluation of topology independence, as it is one of the most important aspects, has two parts: 1) we show that Omnix is able to work with different topologies and 2) we discuss the possibilities of changing topologies at design-time and runtime. The evaluation shows that Omnix meets all the stated requirements. A comparison of Omnix with P2P systems and related technologies is also presented, showing the advantages of Omnix over other P2P middleware systems.

	MoCha	eComP	Mobile MOM	VPP
P2P middleware	No	Yes	No	Yes
Higher level P2P services	No	Yes	No	No
Support for mobile hosts	Yes	Yes	Yes	No
Support for small devices	Yes	Yes	Yes	Yes
Topology independent	No	No	No	No
Server-less	Yes	Yes	No	Yes
Programming language independent	Yes (*)	Yes	Yes	Yes
Platform-independent	Yes (*)	Yes	Yes	Yes
Scalable	Yes (†)	No	No	No
Network/protocol independent	Yes (*)	No	No	Yes
Support for testing	No	No	No	No

Table 8.1: Comparison of P2P topologies 1/2.

	XMIDDLE	iMobile ME	JXTA	Groove	Omnix
P2P middleware	No	No	Yes	Yes	Yes
Higher level P2P services	Yes	No	Yes	Yes	Yes
Support for mobile hosts	Yes	Yes	Yes	Yes	Yes
Support for small devices	No	Yes	No	No	Yes
Topology independent	No	No	Yes	No	Yes
Server-less	Yes	No	Yes	Yes	Yes
Programming language independent	Yes	Yes	Yes	Yes	Yes
Platform-independent	Yes	Yes	Yes	No	Yes
Scalable	Yes (†)	No	Yes	No	Yes
Network/protocol independent	Yes	Yes	Yes	No	Yes
Support for testing	No	No	No	No	Yes

Table 8.2: Comparison of P2P topologies 1/2.

Chapter 9

Conclusion and Future Work

I think there is a market for maybe five computers worldwide.
Thomas J. Watson, Chairman of IBM, 1943

The second important trend we are preparing for is called "pervasive computing"[...] So the networked world [...] will extend further to interconnect perhaps a trillion "intelligent"devices.
Louis V. Gerstner, Chairman and CEO of IBM, 1999 [149]

Peer-to-Peer (P2P) computing has been increasingly gaining popularity. According to CacheLogic [150], 70% of the traffic over ISP networks is caused by P2P file sharing. 95% of the upstream traffic over the last mile is due to P2P networking. On average, 5 million people are participating in a P2P network at any point in time. 35 million Europeans have used a P2P system to download music files. Approximately 50% of campus traffic is used by P2P file sharing systems [85].

This is not only an impressive list of numbers but also shows that it is essential to improve the way in which P2P systems work, how they are connected, how they take advantage of the different capabilities of the peers involved. All these aspects are represented by the topology of a P2P network. P2P is a relatively new area. The past five years have seen a lot of movement in this field. After Napster and Gnutella, further improvements emerged: hybrid P2P topologies, hypercubes, distributed hashtables and many more. [1] These advances in P2P computing have been mainly aimed at file sharing. In other application domains, different characteristics apply (e.g., lower bandwidth utilization). It is also likely that the recent improvements of P2P topologies are not the end of the road.

Also relatively new in the P2P area are upcoming P2P middleware systems. P2P middleware systems offer an application programming interface for a P2P network. Services such as distributed searches, direct communication among peers, etc. are transparently hidden behind a higher-level interface. The fast development of new P2P systems makes it somewhat hard for an application developer to choose which P2P system to use as an underlying communication middleware. Those few P2P systems that can be used as a P2P middleware have

[1]An example for the fast pace of P2P development is the Gnutella network: although it was one of the most influencing P2P systems it has been used for approximately 2 years only before it virtually died.

a fixed topology. As shown in Chapter 3, the topology has a high impact on the usability for specific application domains and use cases. Conventional P2P middleware system lack a great deal of flexibility by using a fixed topology. It is possible to achieve most use case scenarios by almost any P2P topology, but this comes at a high price (i.e., increased bandwidth utilization, less scalable, etc.). There is no such thing as a *perfect* topology than can be used for any use case.

What is missing is an open standard for P2P networks, where applications can access the services provided by P2P systems in a uniform way. If a P2P middleware would exist that allows the replacement of the underlying topology without changing the interface to the application using it, it would be possible to write a P2P application without having to deal with the shortcomings of one or the other P2P topology. If a P2P topology turns out to be a bad choice, it would be easily possible to replace it by another one.

As an example: Gnutella (once a state-of-the-art P2P system) was scalable to a few tens of thousands of nodes. FastTrack, a successor which uses a more sophisticated topology, has more than 2.5 million users at any point in time (without any hint that this is the upper limit). With an open middleware, applications on top of Gnutella could easily switch to the hybrid topology of FastTrack without changing the application itself.

Omnix is the first step in this direction. It allows the creation of P2P applications that are independent from the topology, the programming language and the device that it is running on.

9.1 Analysis of this Dissertation

To get a better understanding of the commonalities and differences of the various P2P topologies, we compiled a list of P2P topologies and classified them in a survey. The result are the three main topology classes: pure P2P (with wild mesh and structured topologies as subgroups), server-based P2P and hybrid P2P. Using this classification, we evaluated the various P2P networks with respect to their scalability, performance, and many more criteria. The resulting conclusion is that there exists no P2P topology that can be used for all use cases effectively. It shows that there is a need for an abstraction between the P2P application and the underlying P2P middleware and its topology.

The consequential product from this evaluation is a generic API. It allows to use higher-level services of P2P networks without the necessity of differentiation between the various topologies beneath. An application programmer using this API does not have to know which topology is used to provide the services he/she is accessing. The validity and usability of the interface has been evaluated in Section 8.2.3.

To support the thesis that topology independence is important for a P2P middleware, an implementation of such an open, generic P2P middleware has been written. This application provides the aforementioned interface to hide the underlying topology implementation. To raise the level of the application and to further prove that the generic interface is also valid for other devices than normal PCs (with respect to the lower device capabilities), it has been build in a component-oriented way that allows to use the middleware on small devices such as mobile phones.

Since Omnix is an *open* middleware, plugins can be used to change the behavior of the P2P system. It is possible to change the topology, the message structure, the ontology of the meta-data, the way in which streams are created and maintained, and much more. In the course of this implementation, a protocol has been specified that can be used with arbitrary topologies and message content. Omnix deliberately does not use any higher-level standards such as XML, SOAP, RMI, CORBA and the like to keep peer implementations as small as possible. It is, however, possible to adopt these techniques if necessary. The component-oriented design of Omnix allows the testing of P2P applications by simulating a virtual network with thousands of nodes. By only replacing the lowest layer of Omnix, a non-intrusive testing of an application is possible (which means that the application does not have to be changed at all for testing).

A comparison of Omnix with conventional P2P middleware systems has been performed. The result is that there exists not a single P2P middleware system that has the same characteristics as Omnix.

Along with this Dissertation, several sub-projects have been started to explore some details of the Omnix middleware in more depth. One of these projects was *Donare*, a master thesis about the application of Omnix for a hybrid P2P system with traffic shaping and load balancing. Another interesting project, *Simix*, dealt with the creation of a testbed for Omnix applications. What is still ongoing work is the creation of a security P2P infrastructure (based on the notion of trust) in Omnix. In Chapter 7, an experimental study has been presented to demonstrate how Omnix performs with different topologies.

Along with the detailed discussion of topologies in Chapter 3, the results of the experimental study confirm the initial thesis that the topology has a tremendous impact on the use case and that it is important for a P2P middleware to be independent of the underlying P2P topology.

9.2 Ongoing and Future Work

The concepts and implementation presented in this dissertation are the basis for several projects currently going on or planned in the future. As Omnix is a topology-independent middleware, there are many ways how it can be extended. In the following, a list of ongoing work and open research problems is given.

We are planning to create additional topology modules to explore whether the Omnix API can be improved (if necessary). This is definitely a process over time because it is very likely that new P2P topologies will emerge frequently. It is also planned to create a module that allow the Omnix to communicate with JXTA peers.

One of the most interesting aspects of the Omnix API is the question of how the application can have access to the proprietary mechanisms of topology modules or other components of the middleware (as discussed in Section 8.2.4).

An ongoing project deals with security aspects in Omnix. It tries to create a so-called *Web of Trust* to authenticate peers in a P2P network. The main challenge in a P2P environment is that there is no central authority to check the credentials of a remote peer. It is part of this project to examine how different P2P topologies affect the usefulness of such algorithms.

The *Simix* project, which is a simulation environment for Omnix, is also ongoing work.

We are currently working on an improved routing algorithm for the connections between two virtual peers. Furthermore, we are looking into possibilities to improve the usability of the system and the creation of statistics. We also plan to look into more detail how individual user behavior can be simulated and configured.

Since Omnix has been designed to work on small devices such as a mobile phone, we are planning to look into possibilities to port Omnix to even smaller devices, such as embedded devices or SmartCards. With the widespread availability and miniaturization of mobile devices, the demand for P2P architectures in ubiquitous computing environments grows. It is also for this reason that we want to port Omnix to other programming languages (e.g., C). This step is necessary to make Omnix available to a wide range of application programmers.

The support for platform-independent storage of meta-data is also an open issue. Topology modules are responsible for storing the shared meta-data persistently. Because this has to be done in a platform-independent way, we are planning to create an interface for components that are responsible for storing the shared meta-data persistently, e.g. on a disk, in a database or in memory.

One of the most demanding challenges in Omnix is the seamless transition from one topology to another. While it is comparatively easy to manually replace a topology module by another (e.g., by changing a single line in a configuration file), it is harder to do this automatically. How does a peer know that it has to change the topology module? How can the peer be sure that the request for changing the topology has been issued by an administrator with the right credentials (this is also the rationale behind creating a security infrastructure for Omnix)? Where does a peer get the new topology module from? How does a peer get informed when it was not online at the time the topology change request has been issued to all peers? For some of these questions we do have answers but it still requires a lot of research necessary until automatic change of the topology becomes reality.

Currently, the definition of the meta-data ontology lies within in the responsibility of the topology module. We are looking into ways of creating a generic meta-data ontology that can be used with any topology. This would help to make the topology completely independent from the application. Omnix provides a very simple ontology with key/value pairs that is probably sufficient for most topologies.

9.3 Concluding Remarks

P2P is an idea, not a system. Many P2P systems with different topologies evolved in the recent past. In this dissertation, we have shown that the topology of a P2P network has a great deal of impact on the usability, scalability, meta-data, and bandwidth utilization (among other aspects as well). It is safe to assume that other P2P topologies will emerge in the future. To cope with the challenges that come with the increasing usage of P2P technologies even in small, embedded devices (in the area of pervasive, ubiquitous computing) it is necessary to adapt the P2P topologies to the respective use cases.

The Omnix P2P middleware provides an abstraction from the underlying P2P topology and thus allows the application programmer to always use the same API and the same services provided by the P2P network without depending on a single topology or the device it is running on.

We proposed a layered architecture that is loosely based on the ISO OSI model. Omnix defines the components of the four lowest layers of the model: *physical, data link, network,* and *transport.* The upper layers were deliberately not touched as we do not believe it to be feasible to specify interfaces for these layers that would not impose restrictions on the usability of the system.

Appendix A

Protocol Statuscodes

The following list shows the various classes of status codes.

- *1xx Informational:* Responses with status codes starting with a "1" indicate that the request has been received by the target peer and is now being processed. This may be necessary if the processing may take more time. If this response is not sent back by the processing peer, the sending peer may assume that the request has not been received by the target peer. A 1xx response does not indicate the end of a request/reply transaction. The initial peer (i.e., the requestor) must still wait for a final response. If, for example, a peer is forwarding a search request to another peer, it may send back a message reporting this fact. The response received from the remote peer may be sent back to the initial peer as a final response.
- *2xx Success:* This class of messages indicate that the request has been received, accepted and processed. Typically, a 2xx response contains also the desired information (if applicable).
- *3xx Redirection:* With this kind of response the target peer informs the initial peer, that the required resource is not (or no longer) available at this peer. This information could also augmented by a pointer to another location where the requested information can be retrieved.
- *4xx Client Error:* If a peer sends a request which is contains bad syntax, uses a wrong protocol name or version, or cannot be processed by the server (e.g., because the requested file does not exist), the target peer sends back a response with a 4xx status code.
- *5xx Server Error:* A 5xx error code is returned if an internal error in the server-peer has occurred. Reasons for such an error could be a defective component or misconfiguration.

Status Code	Reason Phrase
100	Trying
181	Message forwarded
182	Queued
200	OK
300	Multiple choices
301	Moved permanently
302	Moved temporarily
305	Use Proxy
380	Alternative Service
400	Bad Request
401	Unauthorized
403	Forbidden
404	Not found
405	Method not allowed
407	Proxy authentication required
408	Request-timeout
413	Request-entity too large
480	Temporarily unavailable
482	Loop detected
483	Too many hops
485	Ambiguous
486	Busy here
500	Server internal error
501	Not implemented
503	Service unavailable
505	Version not supported

Table A.1: Protocol status codes.

Appendix B

Message Syntax in Augmented Backus-Naur Form

In the following, the message structure of the reference Omnix protocol (described in Chapter 5) by using the Augmented Backus-Naur Form [89], which is an often used extension of the Backus-Naur Form for Internet specifications.

```
message          =  requestline / responseline
                    requiredheaders
                    *messageheader
                    CRLF
                    [ messagebody ]

messagebody      =  *OCTET

requestline      =  method SP requestURI SP
                    protocol "/" version CRLF

method           =  1*CHAR

requestURI       =  1*CHAR

protocol         =  1*CHAR

version          =  1*CHAR

responseline     =  protocol "/" version SP
                    statuscode SP reasonphrase CRLF

statuscode       =  3DIGIT

reasonphrase     =  1*CHAR
```

```
messageheader    =  headername COLON *WSP headervalue CRLF

headername       =  1*CHAR

headervalue      =  *OCTET

requiredheaders =  to
requiredheaders =/ from
requiredheaders =/ 1*via
requiredheaders =/ msgid

to               =  "To" COLON *WSP headervalue CRLF

from             =  "From" COLON *WSP headervalue CRLF

via              =  "Via" COLON *WSP headervalue CRLF

msgid            =  "MsgID" COLON *WSP headervalue CRLF

COLON            =  ":"
```

Bibliography

[1] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, 2003.

[2] Project JXTA, http://www.jxta.org/, 2003.

[3] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[4] Groove Networks. http://www.nfcr.org/, 2003.

[5] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. Base – a micro-broker-based middleware for pervasive computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 443–451, March 2003.

[6] Roman Kurmanowytsch. An Overview of Peer-to-Peer Topologies. Technical Report TUV-1841-2003-04. Distributed Systems Group, Technical University of Vienna. 2003.

[7] NewScientist.com news service. May 23, 2003, http://www.newscientist.com/news/news.jsp?id=ns99993764.

[8] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly & Associates, Mar. 2001. http://www.oreilly.com/catalog/peertopeer/.

[9] P. Mockapetris. Domain Names - Implementation and Specification (RFC 1035), http://www.ietf.org/rfc/rfc1035.txt, 2002.

[10] Peer-to-Peer Working Group, http://www.p2pwg.org/, 2002.

[11] Webopedia, http://www.webopedia.com/, 2003.

[12] SETI@Home. SETI@Home homepage, http://setiathome.ssl.berkeley.edu/, 2001.

[13] Napster. Napster homepage, http://www.napster.com/, 2002.

[14] ICQ, http://www.icq.com/, 2003.

[15] threedegrees homepage, http://www.threedegrees.com/, 2003.

[16] Jabber Software Foundation. Jabber homepage, http://www.jabber.org/, 2003.

[17] Gnutella: The Gnutella homepage, http://gnutella.wego.com/, 2001.

[18] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–??, 2001.

[19] The FastTrack Protocol, http://www.fasttrack.nu/, 2002.

[20] Groove. Introduction to Groove, Groove Networks White Paper, 2000.

[21] genome@HOME. genome@HOME homepage, http://genomeathome.stanford.edu/, 2001.

[22] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII, Schloss Elmau Germany*, May 2001.

[23] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM, San Diego, August 27-31 2001*, 2001.

[24] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. 2001. http://lsirwww.epfl.ch/publications/CoopIS2001.pdf.

[25] GnuNet. Gnunet homepage, http://www.ovmj.org/GNUnet/, 2003.

[26] JXTA for J2ME, http://jxme.jxta.org/, 2002.

[27] Li Gong. Project JXTA: A technology overview. Technical report, SUN Microsystems, April 2001. http://www.jxta.org/project/www/docs/TechOverview.pdf, 2002.

[28] O'Reilly P2P Directory, http://www.openp2p.com/pub/q/p2p_category, 2003.

[29] Detlef Schoder and Kai Fischbach. Peer-to-peer prospects. *Communications of the ACM*, 46(2):14–??, February, 2003.

[30] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. In *The VLDB Journal*, pages 561–570, Sep 2001.

[31] Rüdiger Schollmeier and Gero Schollmeier. Why peer-to-peer (p2p) does scale: an analysis of p2p traffic patterns. In *Second IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks (P2P, Grid, Clusters)*. IEEE, 2002.

[32] Jordan Ritter. Why Gnutella Can't Scale. No, Really, http://www.darkridge.com/~jpr5/doc/gnutella.html, 2001.

[33] Mihajlo A. Jovanovic, Fred S. Annexstein, and Kenneth A. Berman. Scalability issues in large peer-to-peer networks - a case study of gnutella http://www.ececs.uc.edu/~mjovanov/Research/paper.html. Technical report, University of Cincinnati, 2001.

[34] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, 2002.

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 161–172, 2001.

[36] Karl Aberer, Philippe Cudre-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, Roman Schmidt, and Jie Wu. Advanced peer-to-peer networking: The p-grid system and its applications. *Praxis der Informationsverarbeitung und Kommunikation*, 25, 2002.

[37] Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. A scalable and ontology-based p2p infrastructure for semantic web services. In *Second IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks (P2P, Grid, Clusters)*. IEEE, 2002.

[38] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing, 2001.

[39] N. J. Gunther. Hypernets - good (g)news for gnutella, February 2002.

[40] Mayur Datar. Butterflies and peer-to-peer networks. In *Proceedings of ESA 2002 (LNCS)*, 2002. http://dbpubs.stanford.edu/pub/2002-33.

[41] Grokster homepage, http://www.grokster.com/, 2002.

[42] Dejan S. Milojicic et al. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, 2002.

[43] Kazaa Media Desktop. Kazaa homepage, http://www.kazaa.com/, 2002.

[44] Kelly Truelove. Gnutella: Alive, well, and changing fast http://www.openp2p.com/pub/a/p2p/2001/01/25/truelove0101.html, 2001.

[45] LimeWire homepage, http://www.limewire.com/, 2002.

[46] John Kubiatowicz. Extracting guarantees from chaos. *Communications of the ACM*, 46(2):33–38, 2003.

[47] Karl Aberer and Manfred Hauswirth. An overview on peer-to-peer information systems. In *Proceedings of Workshop on Distributed Data and Structures (WDAS-2002), Paris, France.*, 2002.

[48] Ilya Zaihrayeu. Data-sharing p2p systems: Open problems. UNITN, June 2003.

[49] David Barkai. Keynote speech: Internet distributed computing: The intersection of web services, p2p, and grid computing. In *Second IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks (P2P, Grid, Clusters)*. IEEE, 2002.

[50] Bugra Gedik. Determining Characteristics of the Gnutella Network, http://www.cc.gatech.edu/~bgedik/research/mini-projects/mini-project1/F%inalReport.htm, 2002.

[51] R. Cohen, K. Erez, D. ben Avraham, and S. Havlin. Resilience of the internet to random breakdowns. *Physical Review Letters*, 85, 2000.

[52] P. Krishna Gummadi, Stefan Saroiu, and Steven Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems.

[53] Lada Adamic, Rajan Lukose, Amit Puniyani, and Bernardo Huberman. Search in power-law networks. *Physical Review E*, 64(046135), 2001.

[54] Shengquan Wang, Muralidhar Krishnamoorthy, and Dong Xuan. Analyzing resilience of structured peer-to-peer systems. Technical report, Department of Computer Science, Texas A&M University, 2002.

[55] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1), 1985.

[56] Qin Lv et al. Search and replication in unstructured peer-to-peer networks, http://www.cs.princeton.edu/~qlv/download/searchp2p_full.pdf.

[57] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 5–14. IEEE, 2002.

[58] Peter Backx, Tim Wauters, Bart Dhoedt, and Piet Demeester. A comparison of peer-to-peer architectures. In *Proceedings of Eurescom Summit 2002*, Heidelberg, Germany, 2002.

[59] Serguei Osokine. The flow control algorithm for the distributed 'broadcast-route' networks with reliable transport links, http://www.grouter.net/gnutella/flowcntl.htm.

[60] Clip2 DSS. Bandwidth Barriers to Gnutella Network Scalability, http://dss.clip.com/, 2000.

[61] Patrick Feisthammel. Explanation of the web of trust of PGP, http://www.rubin.ch/pgp/weboftrust.en.html, 2002.

[62] Todd Sundsted. The practice of peer-to-peer computing: Trust and security in peer-to-peer networks, http://www-106.ibm.com/developerworks/security/library/j-p2ptrust/?dwzone=security, 2002.

[63] Netscape. Introduction to SSL, http://developer.netscape.com/docs/manuals/security/sslin/contents.htm, 2002.

[64] Locutus. Locutus homepage, http://locut.us/, 2003.

[65] Laura Chappell. Security alert, just say gno!, http://www.nwconnection.com/2001_09/gnutel91/. *Novell Connection*, pages 33–35, 2001.

[66] Neil Daswani and Hector Garcia-Molina. Query-flood DoS attacks in Gnutella. In *Proceedings of the 9th ACM conference on Computer and communications security*, 2002.

[67] Unwanted Links. Gnutella Peer to Peer File Sharing, http://www.unwantedlinks.com/Guntella-alert.htm, 2002.

[68] Vincent Berk and George Cybenko. File Sharing Protocols: A Tutorial on Gnutella, http://www.ists.dartmouth.edu/IRIA/knowledge_base/p2p/p2p_full.htm, 2002.

[69] Demetris Zeinalipour-Yazti. Exploiting the security weaknesses of the gnutella protocol, http://www.cs.ucr.edu/~csyiazti/cs260-2.html, 2002.

[70] John Borland. Gnutella girds against spam attacks, http://news.cnet.com/news/0-1005-200-2489605.html, August 2000.

[71] PayPal. The PayPal homepage, http://www.paypal.com/, 2003.

[72] Hilary Rosen. Statement of Hilary Rosen (Chairman and CEO Recording Industry Association of America) before the Subcommittee on Courts, the Internet and Intellectual Property Committee on the Judiciary U.S. House of Representatives, http://www.house.gov/judiciary/rosen092602.htm.

[73] James Evans and James Niccolai. Peer-to-Peer Will Have Life After Napster, http://www.pcworld.com/news/article/0,aid,41642,00.asp, February 2001.

[74] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems*. Prentice Hall, 2002.

[75] IBM alphaworks. BlueDrekar, http://www.alphaworks.ibm.com/tech/bluedrekar, 2003.

[76] AXIS Communications. AXIS OpenBT Stack, http://developer.axis.com/software/bluetooth/, 2003.

[77] J. Ott, C. Perkins, and D. Kutscher. A Message Bus for Local Coordination (RFC 3259), http://www.faqs.org/rfc/rfc3259.txt, 2002.

[78] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, Berkeley, CA, February 2003.

[79] Ora Lassila and Ralph R. Swick, Editor). Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222, 1999.

[80] Adam Rifkin and Rohit Khare. The Evolution of Internet-Scale Event Notification Services: Past, Present, and Future, http://www.ics.uci.edu/~rohit/wacc, 1998.

[81] Gian Pietro Picco and Gianpaolo Cugola. PeerWare: Core Middleware Support for Peer-To-Peer and Mobile Systems. Technical report, Dipartimento di Electronica e Informazione, Politecnico di Milano, 2001.

[82] Carles Pairot, Pedro García, and Antonio F. Gómez Skarmeta. DERMI: A Decentralized Peer-to-Peer Event-Based Object Middleware, 2003. Submitted to IEEE ICDCS 2004.

[83] p2psockets. http://p2psockets.jxta.org/servlets/ProjectHome, 2003.

[84] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications (RFC 1889), http://www.ietf.org/rfc/rfc1889.txt, 1996.

[85] Jintae Lee. An end-user perspective on file-sharing systems. *Communications of the ACM*, 46(2):49–53, 2003.

[86] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616), http://www.ietf.org/rfc/rfc2616.txt, 1999.

[87] J. Rosenberg, et al. SIP: Session Initiation Protocol (RFC 3261), http://www.ietf.org/rfc/rfc3261.txt, 2002.

[88] P. Resnick, Editor. Internet Message Format (RFC 2822), http://www.ietf.org/rfc/rfc2822.txt, 2001.

[89] D. Crocker, Ed., P. Overell. Augmented BNF for Syntax Specifications: ABNF (RFC 2234), http://www.ietf.org/rfc/rfc2234.txt, 1997.

[90] International Organization for Standardization (ISO). ISO/IEC 10646-1, International Standard, Information technology, "Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", UTF-8 is described in Annex R, published as Amendment 2, 1993.

[91] J. Galvin, S. Murphy, S. Crocker and N. Freed. Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted (RFC 1847), http://www.ietf.org/rfc/rfc1847.txt, 1995.

[92] Clip2. The Gnutella Protocol Specification v0.4, http://www.aduni.org/courses/java/handouts/Gnutella_Protocal.pdf, 2001.

[93] Name unknown (drscholl@users.sourceforge.net). Napster Messages, http://opennap.sourceforge.net/napster.txt, 2000.

[94] Thomas Weidenfeller. A tiny XML parser/processor in Java, http://mitglied.lycos.de/xmltp/, 2001.

[95] John Wilson. MinML a minimal XML parser, http://www.wilson.co.uk/xml/minml.htm, 1999.

[96] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/SOAP/, 2000.

[97] Dave Winer. XML-RPC, http://www.xmlrpc.com/, 2003.

[98] H. Schulzrinne and J. Rosenberg. Internet Telephony: Architecture and Protocols. *Computer Networks*, 31(3), February 1999.

[99] J. Day and H. Zimmermann. The OSI Reference Model. *Proceedings of the IEEE*, 71(12):1334–1340, December 1983.

[100] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–??, 2000.

[101] Idit Keidar. Challenges in evaluating distributed algorithms. *Future Directions in Distributed Computing, Lecture Notes in Computer Science*, 2584:40–44, 2003.

[102] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *ACM Conf. on Principles of Distributed Computing (PODC), Monterey, CA*, July 2002.

[103] D. F. Bacon, J. Schwartz, and Y. Yemini. Nest: A network simulation and prototyping tool. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 71–78, Berkeley, CA, 1988. USENIX Association.

[104] Pei Zheng and Lionel M. Ni. Experiences in building a scalable distributed network emulation system. In *9th International Conference on Parallel and Distributed Systems, Taiwan, ROC*, December 2002.

[105] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *Computer*, 33(5):59–67, 2000.

[106] Ikjun Yeom and A. L. Narasimha Reddy. ENDE: An end-to-end network delay emulator tool for multimedia protocol development. *Multimedia Tools and Applications*, 14(3):269–296, 2001.

[107] M. Allman and S. Ostermann. One: The Ohio Network Emulator, 1996.

[108] David B. Ingham and Graham D. Parrington. Delayline: A wide-area network emulation tool. *Computing Systems*, 7(3):313–332, 1994.

[109] J. Flynn, H. Tewari, and D. O'Mahony. Jemu: A wireless network emulator for mobile ad hoc networks, proceedings of the first joint iei/iee symposium on telecommunications systems research, dublin. pages 6–, November 2001.

[110] M. Kojo, A. Gurtov, J. Mannner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a wireless network emulator. proceedings of 11th gi/itg conference on measuring, modelling and evaluation of computer and communication systems (mmb 2001), rwth aachen, germany, 2001.

[111] George Riley and Mostafa Ammar. Simulating large networks: How big is big enough? In *Proceedings of First International Conference on Grand Challenges for Modeling and Simulation*, Jan. 2002.

[112] S. Bertolotti and L. Dunand. Opnet 2.4: an environment for communication network modelling and simulation. In *Proceedings of the European Simulation Symposium*, October 1993.

[113] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

[114] S. McCanne and S. Floyd. The lbnl network simulator, 1997. http://www.isi.edu/nsnam.

[115] Kalyan S. Perumalla, Richard Fujimoto, and Andrew Ogielski. TED - a language for modeling telecommunication networks. *SIGMETRICS Performance Evaluation Review*, 25(4):4–11, 1998.

[116] James Cowie, David M. Nicol, , and Andy T. Ogielski. Modeling the global internet. In *Computing in Science & Engineering*, volume 1, pages 42–50, January 1999.

[117] Dhananjai Madhava Rao and Philip A. Wilsey. Simulation of ultra-large communication networks. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 112–119, 1999.

[118] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. Parallel / distributed ns, 2000. http://www.cs.gatech.edu/computing/compass/pdns/index.html.

[119] S. Floyd and V. Paxson. Difficulties in simulating the internet. ieee/acm transactions on networking, to appear. available at http://www.aciri.org/floyd/papers.html.

[120] Internet Software Consortium. Internet domain survey, 2003. http://www.isc.org/ds/.

[121] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[122] Mario Schlosser, Tyson E. Condie, and Sepandar D. Kamvar. Simulating a p2p file-sharing network.

[123] NeuroGrid. Peer to peer simulation, http://www.neurogrid.net/twiki/bin/view/Main/PeerToPeerSimulation, 2003.

[124] Debashis Saha and Amitava Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31, March 2003.

[125] Andrew C. Huang, Benjamin C. Ling, and Shankar Ponnekanti. Pervasive computing: What is it good for? In *MobiDE*, pages 84–91, 1999.

[126] M. Satyanarayanan. Pervasive computing: Vision and challenges, August 2001.

[127] G. Good. LDAP Data Interchange Format, Request for Comments 2849, http://www.ietf.org/rfc/rfc2849.txt, 2000.

[128] Barnes & Noble. http://www.barnesandnoble.com/.

[129] Reinhard Steiner. Donare - ein flexibles, modulares peer-to-peer system. Master's thesis, Vienna University of Technology, Institute for Information Systems, Distributed Systems Group, May 2003.

[130] Eric Pfeiffer. Peer-to-peer computing is all about access. http://www.forbes.com/2001/05/15/0515p2p.html, May 2001.

[131] Erik Meijer and John Gough. Technical overview of the common language runtime.

[132] Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2002.

[133] DMCA. The digital millenium copyright act of 1998, pub. l. no. 105-304, 112 stat. 2860 (oct. 28), 1998.

[134] O. Babaoglu, H. Meling, and A. Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *22th Int. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.

[135] D. Anderson, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay network. In *18th ACM Symp on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

[136] Farhad Arbab and Marcello Bonsangue. Mocha: A middleware based on mobile channels. In *Proceedings of 26th Annual International Computer Software and Applications. 26-29 Aug. 2002 Oxford, UK*, pages 667–673. IEEE Comput. Soc, Los Alamitos, CA, USA, August 2002.

[137] Achilles Kameas, Irene Mavrommati, Dimitris Ringas, and Prashant Wason. ecomp: An architecture that supports p2p networking among ubiquitous computing devices. In *Second IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks (P2P, Grid, Clusters)*. IEEE, 2002.

[138] The Disappearing Computer Initiative. http://www.disappearing-computer.net/, 2003.

[139] Do-Guen Jung, Kwang-Jin Paek, and Tai-Yun Kim. Design of MOBILE MOM: Message Oriented Middleware Service for Mobile Computing. International Workshops on Parallel Processing. pages 434–??, Wakamatsu, Japan, September 1999. ACM.

[140] Manel Velasco, Pau Martí, and Josep M. Fuertes. Peer-to-peer communication for virtual industrial devices. In *Proceedings of the IEEE International Workshop on Factory Communications Systems*, Västerås, August 2002.

[141] K. Etschberger. *Controller Area Network (CAN), Basics, Protocols, Chips, Applications*. IXXAT Automation, English edition, 2001. ISBN: 3-00-007376-0.

[142] C. Mascolo, L. Capra, and W. Emmerich. An XML-based Middleware for Peer-to-Peer Computing. In *IEEE International Conference on Peer-to-Peer Computing, Use of Computers at the Edge of Networks (P2P, Grid, Clusters)*, 2001.

[143] IBM alphaWorks. XML TreeDiff http://www.alphaworks.ibm.com/tech/xmltreediff, 1999.

[144] Steve DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) 1.0. Technical report, World Wide Web Consortium, Jun 2001.

[145] James Clark and Steve DeRose, Editors. XML Path Language (XPath), Version 1.0, W3C Recommendation, http://www.w3.org/TR/xpath, 1999.

[146] Yih-Farn Chen, Huale Huang, Bin Wei, Ming-Feng Chen, and Herman Rao. imobile me - a light-weight mobile service platform for peer-to-peer mobile computing. In *Workshop on Internet Technologies, Applications and Social Impact (WITASI 2002), Wroclaw, Poland (October 10-11)*, 2002.

[147] Sun Microsystems Inc. JXTA v2.0 Protocols Specification, 2003.

[148] Jonathan Knudsen. Getting Started with JXTA for J2ME, http://wireless.java.sun.com/midp/articles/jxme/jxta, 2002.

[149] Louis V. Gerstner. Shareholders Speech 1998, Chicago, Illinois, April 28, 1998, http://www.ibm.com/lvg/annual98.phtml.

[150] CacheLogic. http://www.cachelogic.com/, 2003.

CURRICULUM VITAE

PERSONAL INFORMATION

Name	**KURMANOWYTSCH, ROMAN**
Address	Technical University of Vienna, Information Systems Institute, Distributed Systems Group, 1040 Vienna, Austria
Telephone	[+43-1] 58801 – 18419
Fax	[+43-1] 58801 – 18491
E-mail	roman@infosys.tuwien.ac.at

MAIN RESEARCH INTERESTS

Peer-to-Peer Computing:	Engineering an Topology-Independent Open Standard for Peer-to-Peer Systems
Mobile Computing:	Mobile collaboration, Pervasive Computing
Distributed Systems:	Load balancing, Self-adaptation, Distributed Searches

EDUCATION AND ACADEMIC ADVISORS

• May 2000 – to date	PhD: Distributed Systems Group, TU Wien (Advisor: Mehdi Jazayeri)
• October 1999	Dipl-Ing.: Distributed Systems Group, TU Wien (Advisor: Mehdi Jazayeri)

RECENT RESEARCH PROJECTS AND COLLABORATIONS

• 2001, 2002, 2003	Vienna International Festival Web presence
• 2001	MOTION (Mobile Teleinformation Network), 2 Year EU-funded project
• 2000-2002	Austrian Academy of Science Web presence
• 2000	SPARTA (Security Policy AdaptationReinforced Through Agents), 2 Year EU-funded project
• 1999-2000	Visiting researcher at the Hewlett Packard Laboratories in Bristol, UK

TEACHING

Lab	Computer Networks (SS 2001, SS 2002, SS 2003)
Seminar	Mobile Computing (WS 2001), Software Engineering (SS 2002), Pervasive Computing (SS 2003), Software Engineering (WS 2003)
Misc	Research Supervision of MS Students

SELECTED RECENT PUBLICATIONS (MAXIMUM OF 3)

With Engin Kirda, Clemens Kerer, and Schahram Dustdar, *OMNIX: A topology-independent P2P middleware*, Ubiquitous Mobile Information and Collaboration Systems (UMICS), Klagenfurt/Velden, Austria, 2003

With Mehdi Jazayeri and Engin Kirda, *Towards a Hierarchical, Semantic Peer-to-Peer Topology*, IEEE International Conference on Peer-to-Peer Computing, Linkoping, Sweden, September 2002

With Clemens Kerer and Engin Kirda, *A Generic Content Management Tool for Web Databases*, IEEE Internet Computing, August 2002