



University of Zurich
Department of Informatics

A Peer-to-Peer Network Framework with Network Address Translation Traversal

*Daniel Eichhorn
Ottenbach, Schweiz
Student ID: 98-912-579*

Supervisor: Thomas Bocek, Martin Waldburger,
Prof. Burkhard Stiller
Date of Submission: May 15th, 2006

Kurzfassung

Innerhalb kurzer Zeit ist das Peer-To-Peer (P2P) Paradigma erfolgreich auf viele Gebiete in der Informatik angewandt worden. Es erlaubt eine sichere und effiziente Verteilung von Ressourcen und ist auch gegen mutwillige Störungen und teilweisen Netzwerkausfall gut gewappnet.

Auf der anderen Seite ist die P2P-Architektur noch eine junge Technologie und ist den Kinderschuhen noch nicht in allen Bereichen entwachsen. So müssen P2P Anwendungen in einer Netzwerkumgebung funktionieren, die eigentlich viel mehr für das traditionelle Client/Server-Modell entwickelt worden ist. In diesem Zusammenhang stellen besonders Network Address Translation (NAT) Geräte eine besondere Hürde dar. Diese weit verbreiteten Geräte vermitteln Datenpakete zwischen lokalen (privaten) Netzen und dem öffentlichen Internet. Während sie mit klassischen Client/Server Anwendungen meist tadellos zusammen arbeiten, bilden sie für P2P Netze ohne weiteres Zutun ein fast unüberwindbares Hindernis.

Werden die NAT Geräte entsprechend konfiguriert, laufen P2P Anwendungen oftmals trotzdem, jedoch ist diese Konfiguration für den Durchschnittsanwender meist nicht so einfach. Fehlerhafte Manipulationen können zudem zu ernstzunehmenden Sicherheitslücken für das private Netzwerk führen.

Aus diesen Gründen stellt die vorliegende Diplomarbeit Konzepte vor, welche in Zukunft P2P-Anwendungen den Umgang mit NAT Geräten erleichtern werden. Diese Konzepte wurden in einem Framework zusammen gefasst, die Entwickler von P2P Netzen für ihre Anwendungen benutzen können. Anhand einer einfachen Chat-Anwendung wird gezeigt, wie das Framework verwendet werden kann.

Abstract

Peer-To-Peer (P2P) architectures have become a very successful networking paradigm over the last decade. For what ever reason applied, they support efficient distribution of resources and are stable against disturbances such as denial-of-service-attacks or local network failures.

Nonetheless, the P2P paradigm is still young and shows teething problems in a network environment that was originally designed for other purposes. In this context, Network Address Translation (NAT) is a widely used technique to connect local area networks with the public internet. While it works very well for networking applications following the traditional client/server paradigm, it poses a serious problem for P2P networks.

Many of today's P2P applications require manual device configuration to work properly with NAT devices. This is complicated and error-prone, especially for the average computer user. Misconfiguration can even introduce severe security issues to the local network. In addition, the group of internet users behind a NAT device is expected to grow to an important—but by many P2P applications neglected—resource for distributed computing-power or storage space.

For all these reasons, the diploma thesis at hand will present necessary concepts to enable future P2P applications to handle NAT devices efficiently and in a configuration-free way. These concepts are then applied and collected in a NAT traversal framework for P2P developers. In addition, a simple chat application using this framework will provide a simple example of how the framework can be used.

Acknowledgments

First of all, I would like to thank Professor Burkhard Stiller and his two assistants Thomas Bocek and Martin Waldburger for allowing me to write this diploma thesis and for giving me great support over the last six months.

I would also like to mention all the students and employees at the *Institut fuer Informatik* at the University of Zurich, who were there for having a chat, drinking coffee or just spending a smile when I needed it the most.

I would also like to thank Joan Kägi for her invaluable support in reading this thesis and correcting many grammatical mistakes I would not have found without her help.

Last but not least I want to thank my girlfriend Alessandra and my family for supporting me in all these years of my studies and especially for encouraging me and giving me new energy in all situations over the last six months when my batteries were running low.

I could not have come this far without all your support! Thank you very much!

Contents

Kurzfassung	i
Abstract	ii
Acknowledgments	iii
1 Introduction and Motivation	1
1.1 Motivation	2
1.2 Problem Description	2
1.3 Thesis Objectives	3
1.4 Thesis Outline	3
2 Technological Background	5
2.1 Peer-To-Peer Architecture	5
2.1.1 P2P Networks Topologies	6
2.1.2 Distributed Hash Tables	8
2.2 Network Address Translation	12
2.2.1 Full Cone NAT	12
2.2.2 Restricted Cone NAT	13
2.2.3 Port Restricted Cone NAT	13
2.2.4 Symmetric NAT	14
2.2.5 Summary	14
2.3 NAT Traversal Techniques	15

2.3.1	Port Forwarding	15
2.3.2	UPnP	16
2.3.3	Application Layer Gateways	16
2.3.4	STUN	16
2.3.5	TURN	17
2.3.6	Hole-Punching	17
2.4	P2P in NAT Environment	18
3	Related Work	21
3.1	NATTrav	21
3.2	STUNT	22
3.3	JXTA	23
3.4	Skype	23
3.5	Summary	24
4	Framework Design	27
4.1	Requirements	27
4.1.1	High Level Requirements	27
4.1.2	Low Level Requirements	29
4.2	Architecture	30
4.2.1	Peer Roles	31
4.2.2	Components	34
4.3	Alternative Concepts	37
4.3.1	Locating the Rendezvous Peer	37
4.3.2	Layer Design	39
4.4	Summary	39

5	Implementation	41
5.1	Used Libraries	41
5.1.1	XStream - XML Object Serialization	41
5.1.2	Apache MINA	43
5.1.3	Overlay Weaver	44
5.2	Other Evaluated Libraries	45
5.2.1	STUNT	45
5.2.2	UPnPLib	46
5.3	Implementation of the Components	46
5.3.1	NATContinuation Interface	47
5.3.2	NATBroker	47
5.3.3	Discovery Component	48
5.3.4	Traversal Component	49
5.3.5	Rendezvous Lookup Component	53
5.3.6	Chat Application	54
5.4	Summary	56
6	Evaluation	57
6.1	Testing	57
6.1.1	Test Setup	58
6.1.2	Test Run	58
6.1.3	Qualitative Test Results	60
6.1.4	Deployment	61
6.2	Real World Tests	62
6.3	Validation and Verification	62
6.3.1	High-Level Requirements	63
6.3.2	Low-Level Requirements	63
6.4	Prototype: Pros & Cons	64

6.4.1	Security	64
6.4.2	Reliability	65
6.5	Summary	65
7	Summary	67
7.1	Conclusions	68
7.2	Future Work	68
	Abbreviations	75
	List of Figures	75
	List of Tables	77
	List of Listings	79
A	Installation Guidelines	83
A.1	Requirements	83
A.2	Running the example chat application	84
A.3	Using the framework	84
B	Contents of the CD	85

Chapter 1

Introduction and Motivation

The Peer-To-Peer (P2P) paradigm has emerged successfully as a promising new technology in computer science over the last decade. In an application designed following the P2P paradigm, every host participating is more or less equal. Every peer can offer resources at the same time as it requests them from others. This stands in contrast to the conventional client/server architecture, where the client usually only acts as a consumer and the server as a producer of services or resources and therefore underlies an asymmetric hierarchy of hosts.

While it seems that P2P will rather complement the conventional client/server paradigm than replace it, it still brings new—so far not entirely solved problems—to the existing networking infrastructure. This infrastructure was designed to meet the needs of a clearly hierarchical order of hosts: servers needed to be available at a well known address to offer resources to their clients upon request. Following the P2P paradigm, every host becomes client and server at the same time and the network infrastructure has to suit both needs.

A widely used technique in today's networking infrastructure is called Network Address Translation (or NAT). It is usually used on devices that connect local area networks (LAN) with the public internet to reduce the demand for the scarce IPv4¹ network addresses and to further improve security of clients in the LAN.

NAT devices can be seen as a galvanic isolation of the data current between the local network and the internet. Every data packet in IP based communication has a destination and a source IP address. The IP addresses in the local network are different from the

¹Internet Protocol version 4 (IPv4) is an important protocol for world wide network communication. Every host directly connected to the internet gets one unique address out of 4,294,967,296 possible addresses, which will be depleted in the near or far future. IPv6, it's successor, is supposed to solve this problem, but has not yet been successful in replacing IPv4.

ones used in the public internet and a NAT device translates addresses from one kind to the other. While this translation scheme worked well for conventional client/server based applications, it is no longer accurate for P2P architectures and usually requires manual configuration by the user.

1.1 Motivation

Many of today's P2P applications don't really deal with NAT devices. The user has either to configure the NAT device over a web interface or he has to accept lower performance of the services offered by the P2P network. Configuring those devices for an application is complicated and can even be dangerous, if the user disables security features by accident.

But users that are behind a NAT device cannot just be neglected, they are a valuable resource and should be treated as such. While it is hard to get accurate estimations of the share of NATed users connected to the internet, it can be expected that this group is growing rapidly and already constitutes more than 50 percent of internet users in western countries.

That NAT-aware P2P applications are possible has been proved by applications like Skype [29]. The huge success of this voice communication application² can be accredited mainly to its easy setup. Skype does not need the user to configure its NAT box, it just runs. But the developers put a large effort in protecting the mechanisms used in Skype from the public. While several reverse-engineering attempts have lightened this up, the concepts in Skype are still not available to the P2P developer community.

1.2 Problem Description

With this motivation in mind, the work at hand has to find and define the necessary concepts which can help P2P applications to become aware of NAT devices and to handle them without further manual configuration. In order to facilitate future development of P2P applications, a framework containing these mechanisms has to be designed and implemented. Applications using this framework will be able to establish direct—not relayed—connections between two peers residing behind a NAT device.

By resolving this issue, P2P applications will become far easier and stable to handle, their efficiency and reliability will increase, since no more configuring of network equipment will be necessary.

² More than 100 million registered users by April 2006

1.3 Thesis Objectives

The following list specifies the objectives for this work in regard to the thesis' task description and the previously described motivation. This work has to:

- Describe necessary concepts for transparently traversing NAT devices in P2P applications
- Design a Java-based framework with an universal architecture. It should be possible to use the framework by P2P applications to overcome the NAT device issue. The resulting code should be made available under the GNU Public License (GPL [10])
- Scalability: the implementation of the framework should be scalable and therefore avoid single-point-of-failures and traffic-bottlenecks
- Implement a sample application on top of the framework in order to demonstrate its functionality.

1.4 Thesis Outline

The remainder of this document is structured as follows: the next chapter will further introduce the used concepts and will provide the reader with the necessary background. The third chapter will then put this work into the context of other scientific work done on this topic and will compare existing frameworks and their functionality with each other.

The following two main chapters of this work focus on the design and the implementation of the NAT traversal framework and will also give developers of P2P applications a brief insight of how to use the framework.

The subsequent chapter measures the work done in regard to various aspects and evaluates the result. The concluding chapter will then finally summarize this work and give an outlook to improvements and future work.

Chapter 2

Technological Background

This chapter will give a brief theoretical introduction to the two main areas of this work. The first section covers the basics of the Peer-To-Peer paradigm, while the second addresses Network Address Translation. A concluding section will put those two areas into relation to each other.

2.1 Peer-To-Peer Architecture

While P2P applications experienced their first hype in the late 1990s, early, highly distributed applications are much older. USENET for instance, the software that manages message flow for newsgroups, was written in 1979 by Tom Truscott and Jim Ellis for a small network of three machines in North Carolina. This early system was already able to exchange news items among its members, independent of where they were originally injected. USENET kept a data-set up-to-date, distributed over several machines.

Many years later, when the former networks for military use had become available to the broad public known as the internet, the first modern peer-to-peer applications like Napster became more and more popular in a very short time. Because they usually allowed (or even more accurately encouraged) their users to break copyright laws by sharing music files, the peer-to-peer technology became synonymous for digital piracy. While those first representatives of the P2P file sharing class still had a single-point-of-failure (i.e. a central coordination server), later ones became completely distributed and were therefore much harder to control by the authorities if they were used for illegal purposes.

But since then, the P2P technology has proven to be capable of doing much more than just supporting copyright piracy. The concept is successfully used by distributed applications

for storing data as in OceanStore [16] or looking up Voice-over-IP (VoIP) counterparts as used by Skype [29], just to name a few.

Essentially, P2P applications abandon the conventional client/server paradigm, where a central server offers resources or services to a group of clients. In this paradigm not all hosts are equal according to the used or offered resources. Or in other words, under the aspects of communications, a server has to be reachable to its clients at a well-known address and usually only acts upon requests from clients, whereas clients do not have to be reachable all the time.

In contrast to this, every host in a P2P network is server and client at the same time and its peers are more or less equal. But since every peer can offer its services to the other peers in the network, there has to be a way to find the peers that hold the desired resource or service.

2.1.1 P2P Networks Topologies

As stated by [3] all P2P architectures have one thing in common; the actual data transfer between two peers is done directly through a connection between the peer offering the resource and the peer requesting it. The process of how those two peers find each other can be implemented in different ways, however. [5] classifies them into four groups and some hybrid forms that will be explained in the following sections.

Centralized Topology

This first P2P topology resembles the conventional client/server architecture quite a lot. A central server manages the user and resource database. Every peer in this topology has to log in to this central server and then stays connected for the whole time the application is running. Search queries from the peers are directed towards the central server which searches its internal database for matches. If this search succeeds, the querying peer gets a result list with the peers offering the desired resource. To access this resource, a direct connection between the two peers is established. This direct connection between the peers distinguishes the centralized P2P topology from the client/server architecture, since no actual resources are stored on the central peer. The two architectures have one drawback in common, however: the central server can be seen as a single point of failure. In case of disruption on the central node, the whole system suffers. In case of heavy load on the central server, this bottleneck limits the efficiency of the whole network. Napster is often cited as an example for applications that use (or used) such a topology.

Ring Topology

While it became evident that the topology used by Napster is vulnerable to failure on a single node, a new extended topology emerged: the ring topology. Here the services of the central server are distributed over a cluster of servers, which are arranged in a ring. These nodes work together to improve load balancing and availability.

Hierarchical Topology

In a hierarchical topology, the services of the central node are delegated from a root server in a tree-like fashion to its children, which delegate it to their children and so on. Often-used examples for such distributed systems are the Domain Name Service (DNS), Certification Authorities (CAs) or the earlier-mentioned USENET. All of them have in common that there is always a more important node, except for the root node and that the work is distributed over that hierarchy.

Decentralized Topology

Other than in the three earlier-described topologies, a pure P2P network does not have one central instance or a group of distributed central instances; it is completely decentralized. Usually, a joining node has a set of possible bootstrapping nodes, which it contacts sequentially. If at least one of them responds on the contacted socket address, the joining node gets all the data it needs to function in the P2P network. Gnutella, as an early example for this topology class, used query flooding for searches, since there were no more central servers to handle them. But query flooding has one major drawback: efficiency. Flooded queries cause a large amount of overhead traffic, which may, in extreme cases, even exceed the actual peer-to-peer data transfer. This problem was solved by the later introduced distributed hash tables (DHT).

Hybrid Topologies

In addition to the basic topologies, some mixtures from them emerged after a while. They tried to overcome the drawbacks of the pure basic topologies and only bring their advantages together in a hybrid form. An important hybrid out of the possible mixtures is the centralized/ decentralized form. Its advantages can be explained by the ancient strategy of divide and conquer. Instead of having only equal peers, some of them get promoted to local leaders for other nodes. These leaders are known by many names.

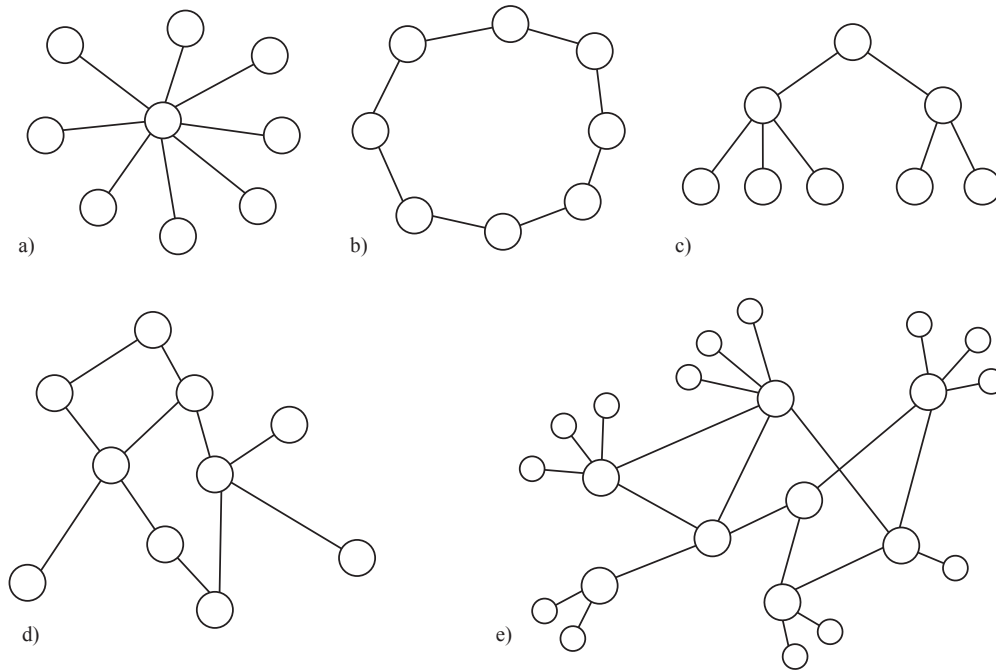


Figure 2.1: P2P network topologies according to [5]: a) Centralized, b) Ring, c) Hierarchical, d) Decentralized, e) Hybrid Centralized/ Decentralized

Sometimes they are called *group leader nodes*, *super nodes* or *ultra nodes*. The important fact about them is that they locally behave like central servers for a group of other nodes. But among the group of super nodes every one of them is equal. What makes a normal node a super node depends on the function of the application. Sometimes it is enough for a normal peer to become a super peer, if it has enough bandwidth to share, in other applications also the availability over time, or its reachability in the network, matters.

On one hand the super node keeps a list of peers attached to it and on the other hand it exchanges maintenance messages with its neighbor peers in the flat hierarchy of the super nodes. Since super nodes only have to keep track of a relatively small group of attached peers, the bottleneck is reduced by far and the failure tolerance is increased. Examples for this hybrid topology are Kazaa or its technological successor Skype. Figure 2.1 depicts the different P2P network topologies.

2.1.2 Distributed Hash Tables

The previous sections described possible P2P network topologies on the level of network connections. However, there are other structures used on layers above the network layer. According to [28], an overlay in an P2P network is the organizational structure at the

application layer. Overlays can be grouped in structured and in unstructured ones. While the unstructured overlays are relatively easy to maintain and also allow more sophisticated search queries, they have a huge disadvantage: they do not scale well. This is because queries flood all over the network and cause a huge amount of overhead traffic. Peers with a weak connection (e.g. dial-up) get flooded with queries and maintenance messages until they can not handle any more connections. Then they seem to be offline and this endangers the network is becoming fragmented. A linear growth of the node number causes a much higher growth of the query traffic in that overlay, which is the actual reason why unstructured overlays do not scale.

In opposition to unstructured networks, structured ones have a tightly-controlled overlay topology. All the nodes work together to maintain a useful structure while all the time nodes in the network continually join, leave and fail. The maintenance costs are often quite significant, but growth by the node number only increases this maintenance traffic marginally [28]. And even more important, data look-ups are guaranteed to finish within a limited number of messages and they will return a result, if the data is available somewhere in the network. Despite all these advantages, structured networks usually offer only limited options for queries.

Distributed Hash Tables (DHTs) are structured overlay networks that particularly provide two primitive operations: *put(key, value)* places a value attached to a hash key in the DHT and *lookup(key)*, which routes the query to the node responsible for that key.

Here are some features that most DHT variations have in common:

- Every node has an unique identifier
- Efficiency of the overlay network has a direct impact on the scalability of the system
- Each node maintains a routing table containing pointers to a small number of other nodes
- Incoming queries are forwarded to the node that is closest to the look-up key
- Every DHT based system measures closeness differently, the metrics change

The following sections briefly describe the most important DHTs and their characteristics.

Plaxton Mesh

In a Plaxton Mesh [21], all objects and nodes have unique numeric identifiers of the same length and the resources are replicated at different nodes. The node that shares the

longest prefix with an object is considered to be the root for that object. The replicas of that object are arranged to a virtual height-balanced tree, where every node keeps a table for the copies of the object in its sub-tree.

Every node maintains a routing table and a pointer list to objects. Because the Plaxton algorithm only updates the pointer list when an object is inserted or deleted, it assumes a relatively static node population and performs weakly in networks with frequently joining, leaving and failing nodes.

Tapestry

Plaxton's algorithm had to be improved to overcome the assumptions about static nodes. One of the improved algorithms was Tapestry [35], which introduced several roots for one object in order to avoid single-points-of-failure as in the Plaxton Mesh. A back pointer list was added that points to nodes where the current node is a neighbor. To detect failed nodes, heartbeat messages are sent to peers in the routing table. If a peer does not respond, it gets a second chance and is only removed from the routing table after a given time. The object pointers can expire and need to be updated periodically. This tries to avoid pointers to nonexistent objects.

Pastry

The index space in Pastry [26] is circular and is ranging from 0 to $2^{128} - 1$. The ID of each joining node is generated randomly in order to get a uniform distribution over the index space. Every object is also mapped to a 128 bit key and located at the node that is numerically the closest to its object key.

Every node in Pastry maintains a routing table, a neighborhood set and a leaf set. To take advantage of the geographical distribution of the P2P network, the neighborhood set contains nodes that are closest to the local node. Various proximity metrics can be used for this. e.g. number of IP-hops or roundtrip-time. The leaf set on the other hand contains the nodes numerically closest to the local node, one half with smaller IDs and one half with larger IDs than the local node.

Chord

In the Chord overlay network [31] every node is assigned to an m bit index in a virtual circle. Every node maintains information about its direct neighbors in the ring (called

	Plaxton	Tapestry	Pastry	Chord	CAN
Decentralized	Yes	Yes	Yes	Yes	Yes
Space cost	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$2d$
Data read	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N^{1/d})$
Data insert	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N^{1/d})$
Data delete	$O(\log^2 N)$	$O(\log^2 N)$	$O(\log N)$	$O(\log N)$	$O(N^{1/d})$
Node insert	-	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(N^{1/d})$
Node delete	-	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$
Node failure	No	-	$O(\log N)$	$O(\log N)$	$O(1)$
Locality	Yes	Yes	Yes	No	No
Load balance	No	No	No	-	Yes

Table 2.1: Comparison of the five previously described DHT algorithms according to [12]. N stands for the number of nodes in the network, d for the number of dimensions in CAN

predecessor and *successor*). In addition each node keeps information about succeeding nodes in an exponential distance to the own index. This table is called *finger* table and allows lookups with an order of $O(\log(n))$.

CAN

In contrast to the other described overlay networks, CAN [22] uses a d -dimensional Cartesian coordinate space on a d -torus. Each peer is responsible for one zone in this space and peers are called neighbors, if they are responsible for adjacent zones to the local peer. Resources are mapped deterministically to a point in this space and *belong* to the node responsible for that area.

DHT Summary

The previous sections described five of the most important DHT algorithms, such as the Plaxton Mesh, Tapestry, Pastry, Chord and CAN. They differ mostly in the virtual shape of their index space and in what information is maintained on every node about the network. Table 2.1 is a shortened version of the one in [12] and gives a good comparison between these five basic DHT algorithms. The lines with $O(\cdot)$ describe the average costs for the given operation in one of the five DHT algorithms. N stands for the number of nodes in the overlay network, d for the d -dimensional index space in the CAN algorithm.

2.2 Network Address Translation

In the days of Napster, most internet users at home were connected to the world wide web using dial-up links. Since the users had to pay (at least in western Europe) for the time they stayed connected, they did not remain online 24 hours a day. With the advent of broadband technologies such as ADSL and internet-over-TV-cable, most internet service providers (ISPs) changed their business model to a flat-rate system. Access to internet was charged with a constant monthly rate, sometimes with a limited monthly data transfer volume in addition.

Since it became obvious that the available IP address pool would run out soon without further sanctions, the ISPs made their customers pay for extra IP addresses on one hand and reused or shared addresses for different customers. This situation led to the invention and broad use of a technique called network address translation, often also called NAT, as described in 1994 in [8, 30]. In these first publications about NAT, the authors already stated that it was only meant to be a short-term solution against IP address depletion and would introduce several new problems to networking. But it was easy to implement for network routers and soon most manufacturers offered devices using it. Because RFC1631 was more like a description than a standard, it was implemented in many different ways and today's NAT devices have a large variety of behavior, which is difficult to predict if the exact model and the manufacturer are not known.

According to [25], the behavior of NAT devices can be separated into four different groups. Because it is fundamental to understand which problems can arise when handling with NAT devices, those four groups will be further described here:

2.2.1 Full Cone NAT

In a full cone NAT, all requests from the same internal address and port are mapped to the same external address and port. External hosts can send packets to this external socket without any preceding connection attempts from behind the NAT device. Figure 2.2 shows an example: Host A is connected in the LAN to the NAT device B, which translates its internal address to an external WAN address. Hosts C and D can now send data packets to the public IP address 217.31.55.10 with port 12345, which will be forwarded to host A by the NAT gateway.

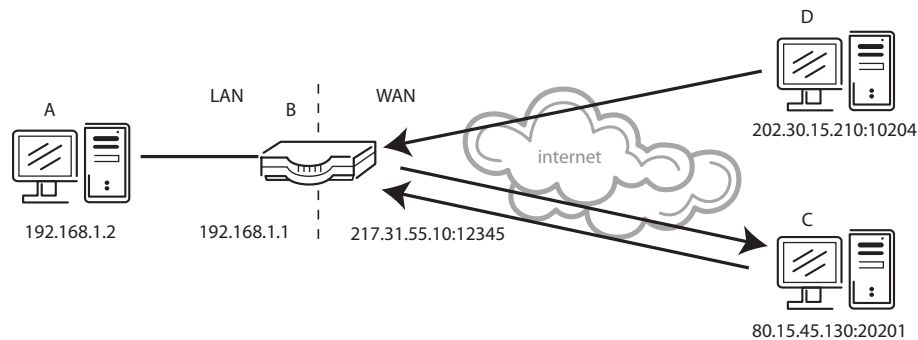


Figure 2.2: Full Cone NAT

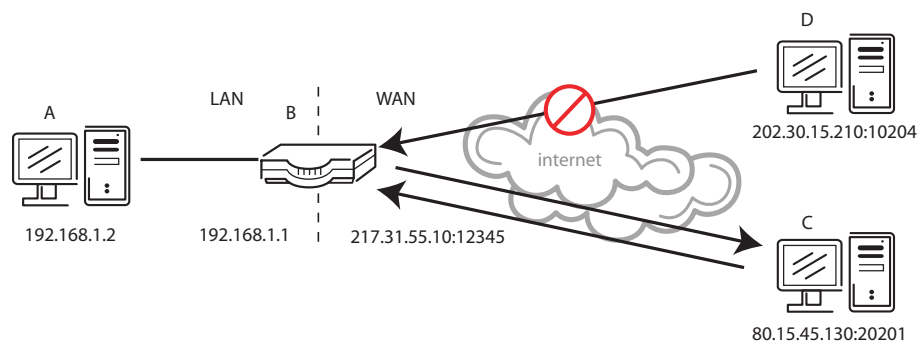


Figure 2.3: Restricted Cone NAT

2.2.2 Restricted Cone NAT

As with the Full Cone NAT, in a Restricted Cone NAT internal IP address information is mapped to an external one. The difference is that here no unsolicited packets from outside will be forwarded into the LAN. This behavior introduces additional privacy (and therefore security), but it also makes it harder to establish peer-to-peer connections. Figure 2.3 shows the difference to the Full Cone NAT: an external host can only send packets to the host behind the NAT if initial packets have been sent to it. That is the case for host C, but not for host D; its connection attempts will be blocked by the gateway.

2.2.3 Port Restricted Cone NAT

In a Port Restricted Cone NAT, the constraints are even tighter than in the previous one. Here not only the IP address matters for response hosts from the WAN side, but also their port. Therefore, even if an application on a dedicated IP address/ port is allowed to send packets through the gateway, it is not necessarily the case for other outgoing ports from the same machine.

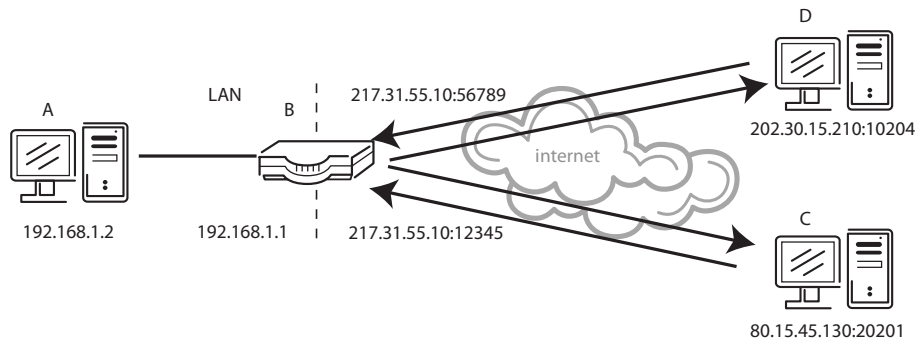


Figure 2.4: Symmetric NAT

2.2.4 Symmetric NAT

A Symmetric Cone NAT maps a request from an internal IP address and port to a specific host, always to the same external address and port. But when the same host sends a packet from the same source address to a different destination, a different NAT mapping is used. In addition, as with Port Restricted cone NATs, unsolicited packets from the outside are dropped. This mapping rule makes the Symmetric NAT the most difficult of all four types to handle, because it is difficult to predict what mapping rule the NAT device will use for a specific connection and the reliable prediction is a corner-stone for the hole punching technique later described. Figure 2.4 shows the difference of the Symmetric NAT to the previous three types.

2.2.5 Summary

The previous sections have described four different classes of NAT devices; there are full cone, restricted cone, port restricted cone and symmetric NAT devices. Table 2.2 shows how internal entries of the NAT device's translation table would look like according to the examples illustrated in figures 2.2, 2.3 and 2.4. As it can be seen by the entry for the full cone NAT, any external host can send data packages to the external address. In a restricted cone NAT only packets from a specific IP address are forwarded and in a port restricted cone NAT only, if they come from this specific IP and a given port. The symmetric NAT has two entries and their external ports are different for every outgoing connection.

	Internal Addr.	External Addr.	Remote Addr.
Full Cone	192.168.1.2:12345	217.31.55.10:12345	Any
Restricted Cone	192.168.1.2:12345	217.31.55.10:12345	80.15.45.130
Port Restricted Cone	192.168.1.2:12345	217.31.55.10:12345	80.15.45.130:20201
Symmetric	192.168.1.2:12345	217.31.55.10:12345	80.15.45.130:20201
	192.168.1.2:12345	217.31.55.10:56789	202.30.15.210:10204

Table 2.2: The four different NAT types summarized with their internal translation entries according to the examples in the previous figures.

2.3 NAT Traversal Techniques

As described in [14], the presence of NAT devices can pose a problem if either the communication protocol encapsulates IP address information in the payload instead of the header or if a certain host behind a NAT device has to be reachable from within the public network. If, as in the first case, the IP address information (e.g. about the sender), is part of the payload, the NAT device cannot translate the local address into the public one. Answers from the endpoint therefore cannot reach the proper NAT device and communication fails. On the other hand, if a peer behind a NAT device should be addressed from the outside, the NAT device does not find any entry in its translation table and therefore cannot know to what address to send the data packages within the local network. There are several techniques to solve this dilemma, and all of them have pros and cons, which will be explained in the following sections.

2.3.1 Port Forwarding

Port Forwarding can be found within most NAT devices, because it is a straight-forward solution to make a given service inside a local network available to the outside. The user sets up port forwarding over the NAT's configuration interface, which is usually either over telnet or using a web interface. He then can choose what ports on the public side of the NAT box will be forwarded to a client within the local network. To make port forwarding work, two assumptions must hold for most NAT devices: first, the host that the data packets are forwarded to must have a fixed IP address and second, only one client behind that NAT box can offer its service at the same time to the public on a given port.

While the first assumption can hold if the given client does not get its IP address dynamically from a DHCP server, the second assumption weighs heavier for P2P applications, because they often have an application-specific port range. In other words, if port forwarding was configured for one host in the local network, it would no longer be possible to

enable a second host for the same P2P application. Having a large company or university network in mind, this would disable many computers from participating (which might even be a welcome side-effect for the network administrator).

2.3.2 UPnP

The Universal Plug 'n Play Forum is a group of companies and individuals across multiple industries who are authoring specifications for UPnP aware devices and services [32]. Among other advantages, UPnP-enabled devices provide a programmatic interface to applications in order to control the flow of network packets through NAT devices. More and more network devices provide this UPnP interface, but the UPnP functionality is often disabled by default and this for a good reason. In 2001, the eEye company published a press release about three highly dangerous security vulnerabilities existing in all versions of Windows ([7]). For instance, a machine running Windows XP accessible from the internet could be easily manipulated using a buffer overflow, if the UPnP service on that machine was enabled and the machine was directly exposed to the public internet. Even if this was *only* an implementation specific security issue the faith in the reliability of UPnP was hurt with it.

2.3.3 Application Layer Gateways

According to [14], a wide range of applications fail if NAT is on the way between sender and receiver. One reason for this is that NAT only inspects and translates IP address information when it is in the header of a data packet. If these addresses are also in the payload, the NAT devices do not detect it and cannot translate it to the appropriate correct address. In this case, application-specific devices called Application Layer Gateways (or short ALGs) can be introduced to the network infrastructure, either on the same node where the NAT resides or on different nodes along the communication path.

But since ALGs are application-specific, it is expensive to use them and they are usually only found in enterprise environments. The task of ALGs becomes even harder if the payload is encrypted. How can the ALG translate an internal address to an external one if it cannot read the message by which it is transported?

2.3.4 STUN

As described in [25], the STUN protocol (Simple Traversal of User Datagram Protocol Through Network Address Translator) is a relatively simple mechanism for applications

to discover the presence and types of NATs and firewalls between them and the public internet. The public IP address of their NAT device can be determined and as a result with the proper setup, two applications, both behind a NAT device, can establish a direct UDP-based connection between them. STUN is often used for Voice-over-IP (VoIP) telephony together with the Session Initiation Protocol (SIP, [23]) to allow the telephony applications behind a NAT device to be reachable from the public internet.

2.3.5 TURN

Traversal Using Relay NAT (TURN) is a protocol that enables NATed clients to receive incoming data over TCP or UDP as is described in [24]. TURN can be seen as the fallback strategy for STUN, i.e. if a specific NAT device refuses to work as demanded by STUN. A client willing to be available for other clients registers on a TURN server. The server then forwards all incoming data addressed to the client from a third party over the already existing connection.

Because with TURN all traffic between two peers has to be relayed by the TURN server, it can only be used as a last resort. Why should any peer suffer from heavy traffic if it does not have any benefit in doing so?

2.3.6 Hole-Punching

Hole-punching is the term for a method to *trick* a NAT device into allowing two peers behind a NAT device to establish a direct connection. The word *trick* is used, because hole-punching is not a method that was built into NAT devices deliberately. Hole-punching rather more tries to take advantage of properties existent in many NAT implementations to overcome a design flaw. As described in section 2.2, NAT devices keep translation entries in a table to decide where data coming from the outside has to be sent to on the inside. Hole-punching targets for creating an entry on the NAT device's translation list for a connection that is not established yet, but soon will be.

In case of UDP-based hole-punching for two involved NATed clients, the first one sends packets to the public address of the other client. Since the remote NAT device does not have a translation entry for this connection yet, it drops all messages. Now the second client sends data packets to the public address of the first one, to the exact IP and port where the previous message came from. Although the message of the first client was dropped, it created a translation entry on its NAT device which now allows the message from the second client to pass and with this the connection is established.

For completeness it has to be mentioned that UDP-based hole-punching is also used by the previously described STUN protocol. But hole-punching is also used for TCP-based hole-punching and therefore exceeds the function range of STUN in the matter of pure NAT traversal. STUN, in contrast, offers other services in addition.

2.4 P2P in NAT Environment

The main problem for P2P applications in a NAT environment is that NATed peers are usually not reachable for arbitrary peers. Unsolicited connection requests from outside are denied by the NAT device because the requests are sent to the public internet address of the gateway and the NAT device has no clue to which internal address the request should be forwarded to.

Of course, this is different if the NATed peer already has successfully sent packets to the other peer, since the NAT device keeps this translation-rule in a table for a given time and responses from outside can be forwarded appropriately.

Because in P2P applications like in file-sharing the set of running peers usually suffers from high fluctuation—joining, leaving and failing nodes—it is crucial that peers are reachable easily. Data lookups and information about the always changing overlay topology are transported by messages which depend on reaching their destinations. Some P2P algorithms even use iterative message routing which means that they start with a peer they already know. This peer answers with a list of peers that are closer to the destination than the peer itself and the starting peer sends message to one of them and so on, until it reaches the destination.

Figure 2.5 illustrates an example for iterative message routing: Peer A is looking for D and asks peer B about it. B responds that C is closer to D and A repeats the question with C. This time, peer C answers with the address of D, since D is in its routing table. With both C and D, peer A had no previous message exchange. If they are behind a NAT device A will not be able to contact them directly, since their NAT device does not know anything about A and will deny the connection attempt.

Iterative message routing is good for the overall network efficiency and especially for the intermediate peers, since the requesting peer has to carry the main traffic-burden by himself instead of leaving it to the nodes along the path. But this also makes it unavoidable to contact so far unknown peers with every iteration step, up to the destination.

Switching the message routing from iterative to recursive propagation could at least partially solve this problem. In recursive message routing the original sender passes the

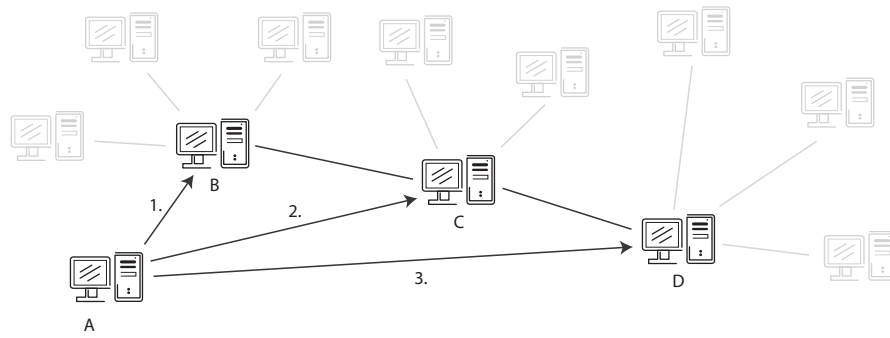


Figure 2.5: Iterative Message Routing: Peer A has to contact unknown peers until it reaches destination D

message to the node in its routing table that seems to be numerically closest to the destination. From now on, every peer along the path just forwards the message to his neighbor-nodes closest to the destination, without further notifying the sender. Here, the share of connections made to unknown nodes is much smaller and so are the possible failures due to intermediate NAT devices. But, as previously explained this means additional burden for the forwarding nodes between sender and receiver of a message.

Chapter 3

Related Work

After the previous chapter has introduced the necessary background to the topic of interest, this chapter continues with an overview about frameworks or applications that aim to solve as well the problems that P2P applications face in a network environment with present NAT devices. While all of the presented frameworks are equipped with interesting features, no one really satisfies completely. The concluding section of this chapter will summarize the advantages and drawbacks of these frameworks to give a guideline for the design of a new framework.

The four presented systems are not a complete list of all known frameworks or applications covering the P2P/ NAT traversal topic. They are chosen to introduce to specific aspects which should be improved by a new framework. These aspects concern the supported protocols for NAT traversal, the mechanisms used for NAT traversal (and their performance), if the mechanisms are available as a library for new application development and the complexity of using the framework.

3.1 NATTrav

NATTrav [9] is a framework that offers the necessary services to traverse NAT devices with TCP-based hole-punching. So called connection brokers facilitate establishing connections between two peers. These connection brokers are replicated to improve availability and scalability, but they are not part of the actual P2P network. Uniform Resource Identifiers (URIs) are used to identify and locate peers for connection establishment.

The NATTrav framework was presented in a paper from J.L. Eppinger in 2005. While he gives a detailed description of the mechanisms used in that library, he did not publish

the Java source-code itself or at least the paper gives no indication for it. Therefore the library could not be used for evaluation purposes.

However, NATTrav lacks the support for UDP traversal, which might be required by multimedia-streaming applications or for other purposes as well. The connection brokers not being part of the actual P2P network require an extra party to run and maintain these nodes. And even if they are replicated they will still be a small group compared to a global running P2P application. For this reason it is desirable to recruit this kind of nodes within the group of common nodes.

3.2 STUNT

Simple Traversal of UDP Through NATs and TCP too (STUNT [27]) is a framework that uses an elaborate NAT traversal mechanism for TCP and is implemented in Java. STUNT uses two different peers that are not NATed to anticipate the mapping rule for ports of the NAT device. This can be seen as some kind of triangulation of the network address translation behavior that provides more accurate results than other mechanisms.

On the other hand, STUNT has no built-in P2P mechanisms yet. In the current version it uses hard-coded IP addresses of the two responders and this must be seen as a single-point-of-failure. In addition, the directory server—where every client has to be registered—has to be the same for all nodes.

The library is based on a paper of Guha and Francis [11]. For this paper they wrote a client that tested various TCP traversal techniques *in the wild*. For this matter they asked the internet community to download and use this test client and to provide the scientists with information about their network configuration. In this self-selected user group, their clients could connect to other NATed clients in more than 85% on average, using the algorithm implemented in STUNT.

While this framework clearly uses one of the most reliable and elaborate TCP hole-punching protocols available at the moment, it lacks direct applicability to P2P applications. Also the framework does not have support for UDP traversal yet¹.

¹This is going to change soon. The developer has implemented this feature and it is already available for beta-tests.

3.3 JXTA

JXTA [2] is an open-source P2P framework introduced by Sun Microsystems in 2001. It is basically a set of XML based protocols and allows to develop applications that can be run on virtually any Java-enabled device, starting with cell-phones up to main-fraines, in order to allow decentralized communication.

JXTA uses relaying to allow NATed peers to connect to the JXTA P2P network. Peers behind a NAT device—in JXTA called edge peers—are connected to reachable nodes called relay nodes. These relay nodes forward all messages addressed to the edge peers registered with them. While relaying is a very reliable NAT traversal technique, the network performance suffers heavily if the relay nodes' bandwidth is exhausted by all the relayed message traffic. The relay nodes can become bottlenecks very easily for all edge peers registered with them. In addition, if these relay nodes are recruited out of common nodes, an end-user might not be willing to offer his hole bandwidth for the benefit of others to his own disadvantage.

The performance problems that come with relaying are one problem, JXTA's complexity is another one. JXTA is a very powerful framework and it offers many features to the developer. But this richness comes with a price; it is time consuming for a developer to understand and integrate it into his own P2P application.

For the developer of such a P2P application, a NAT traversal framework should be more simple to apply and it should only use NAT relaying as a last resort.

3.4 Skype

Skype [29] is a very successful P2P network mostly used for voice communications. Since it is closed-source and most of the communication is encrypted, most publicly available technical information about it is reverse-engineered. To do so, Baset und Schulzrinne [4] ran various experiments using network monitoring tools . They gained deep insight into the Skype protocol and because Skype obviously solved the NAT traversal problem it should be mentioned here.

The Skype overlay network consists of two different types of nodes, ordinary nodes and super nodes. Super nodes are ordinary nodes which serve the network better than ordinary nodes in regard to bandwidth, CPU resources, memory and availability. The super nodes are the end points of ordinary hosts in the overlay network. Because they become super

nodes if they are behind NAT devices, they can act as some kind of rendezvous server for other peers in the network for "penetrating" the NAT devices.

Skype uses a third type of node: the credential server. It keeps usernames unique and certifies the peers' public keys used for the encrypted point-to-point connection. Even if this introduces a single point of failure to the otherwise distributed network, it probably is the only solution for the application provider to remain in control over the network. Control over the network is important for Skype, since this company offers additional paid services for calling to traditional phones and depends economically on having full administrative control over the network.

While the NAT traversal mechanisms built into Skype seem to work very reliably, they are not available (i.e. as a free library) to P2P developers and therefore no option to them. That the work at hand is aiming in solving the NAT traversal issue universally and making the results available to the public can be clearly seen as an advantage over a closed-source application like Skype.

3.5 Summary

	NATTrav [9]	STUNT [27]	JXTA [2]	Skype [29]
Type	P2P-Lib.	H.-Pnch. F.work	P2P Framew.	P2P VoIP App
Open-Source	(√)	√	√	-
Program. Lang.	Java	Java/ C++	Java/ C++	?
Protocols	TCP	TCP/ UDP	TCP	TCP/ UDP
Traversal	H.-Punch.	H.-Punch.	Relay	H.-Punch/Relay
Rendezv. Nodes	Replicated	Single-Point	Replicated	unNATed, bandw.

Table 3.1: Four existing NAT traversal frameworks in comparison

This chapter has presented four different existing approaches that can help developers of P2P applications to overcome the NAT traversal problem. Their most important features are listed in table 3.1. However, everyone of them lacks support in one or several aspects in regard to the thesis objectives made in section 1.3.

NATTrav and JXTA only support TCP for data transport through NAT devices, but applications streaming multimedia directly from one peer to another usually also require UDP. STUNT has implemented excellent mechanisms for NAT traversal, but it uses single-points-of-failures—which are not distributed—for pinpointing the NAT-device behavior and for looking up peers. Being fault-tolerant on single nodes is one important feature of highly distributed systems that STUNT cannot cover at the moment. Finally, although

Skype seems to have solutions for many NAT- and firewall-related problems, these concepts are not available to the public, since Skype protects its internal mechanisms with much effort from interested eyes.

This leads to new demands for features which the new framework will have to satisfy:

- Support for *both* UDP *and* TCP NAT traversal. Both protocols are necessary to support a large variety of applications.
- No relaying as main NAT traversal technique. Relaying introduces bottlenecks to a network system and should therefore be avoided.
- Available and extendable for everyone, especially for P2P developers. Since this framework aims to be a *toolbox* for all kind of P2P applications, it should be published as Open-Source.
- Simple programming interface. For instance, the complexity of JXTA can keep developers from using it.

Chapter 4

Framework Design

This chapter addresses the design of the P2P NAT traversal framework according to the thesis task description and the deduced conclusions from the previous chapters.

4.1 Requirements

In the following sections the constraints of the NAT traversal system will be described. They either originate in the task description of the diploma thesis or evolved from the previously-described related work. If the NAT traversal system is regarded as a black box, then the first-described high level requirements address the constraints important for everybody using this black box. The low-level requirements are important for everything that happens inside of it.

4.1.1 High Level Requirements

The high-level requirements describe the architecture constraints from the user point of view. In the context of this framework, the term "user" is synonymous for application developer, e.g. the developer who wants to use the framework. In single cases, it also addresses the end user who intends to run an application built on top of the framework, but these cases will be mentioned explicitly.

Configuration-Free Setup

This requirement mostly targets the need of the end user. In a lot of today's P2P based applications, the user has to configure his NAT box for port forwarding in order to use

the full application functionality. This requests that he has to be aware of being behind a NAT box or a firewall at all. This is not self-evident, since broadband (ADSL) modems often already incorporate this functionality, but the user is not necessarily aware of using it. Or said differently, the user did not buy a NAT box but rather a broadband modem and is therefore not aware of it.

However, even if the user knows that he is using the NAT functionality (e.g. the application could have told him), for an average home-user it can still be a long way to set up port forwarding for a specific application: first, the access to the NAT configuration interface has to be known. For a web interface, the IP and the access credentials to the NAT box have to be known. When the user has finally gained access to the interface, the proper page for those settings has to be found. In order to make port forwarding work properly, the ports to forward for a specific application have to be known, as well as the IP of the targeted machine in the LAN. Getting the IP of your own machine is not a trivial task either.

But a configuration-free setup will not only help the average home user. Usually employees in an enterprise network do not have administrator access to the networking equipment and could not set up port forwarding, even if they knew how to do it.

Simple Application Programming Interface

The interface of the framework to the outside world should be as simple as possible. The application developer should not have to care about all the mechanics under the hood. The application on top will feed the framework with the necessary information (e.g. about existing nodes) and will be notified when the framework is ready to proceed. The application should have access to the gained information about the network environment. Further, it will tell the system to establish a connection to a given node in the network, either based on UDP or TCP.

From the view of the application developer, the NAT traversal system has to behave like a black box. This will help to keep the system expandable and ready for future changes.

Application Decoupling

The framework should be built in a way that it allows the decoupling of the framework from the application. Network operations might take some time until they are finished and the application using the framework should not block until that task is done. That also introduces the advantage that even for communications failure, the application remains

controllable. It also will introduce some additional design problems: a call-back mechanism has to be used to notify the application about success or failure of the requested task.

Licensing

The resulting code has to be put under GNU General Public License ([10]) in order to make it available to the P2P developer community. All external code used (e.g. libraries) has to suit this constraint as well.

4.1.2 Low Level Requirements

The low level requirements are the constraints that the NAT traversal system has to meet *under the hood*. They might only partially be interesting for its users, but are the corner-stones for its developers.

Programming Language

As requested by the task description for this diploma thesis, Java should be used for developing the prototype of the traversal system. This makes sense, since a large number of the previously described scientific P2P networks are also written in Java and they therefore easily could use the traversal framework in the future.

Because the old input/ output library of Java (*java.io*) performed badly for a large number of connections, the new input/ output package (*java.nio*) should be used. This leads to the minimum Java version 1.4, because the new input/ output system was introduced by SUN Microsystems with this version.

Messaging

Many communication protocols have been developed over the years. They can differ in many aspects: efficiency, structure, readability (e.g. for humans) or complexity. The communication protocol designed and used in the NAT traversal framework should be structured by the extended markup language (XML). This helps reading and creating messages for debugging with common tools but also leads to a significant message overhead, compared to pure bit-length encoded protocols. It is possible, however, to reduce the overhead by using message compression.

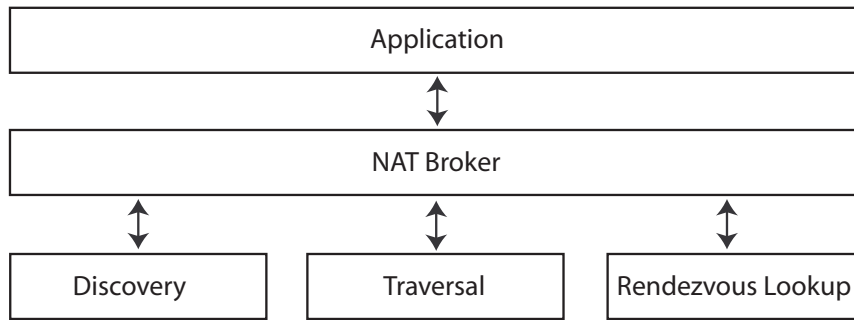


Figure 4.1: The framework's architecture from a high-level view

Networking

As already described in the high level requirements, the NAT traversal framework has to support establishing direct (P2P-) connections between to NATed hosts for both UDP and TCP. Since the NAT traversal problem is relatively reliably solved for UDP, some applications try to abuse UDP as a carrier for TCP. They implement typical TCP features into UDP, such as packet-order-control or retransmission in case of lost packets. But because TCP has been optimized for these features over the last two decades in software as well as in hardware, it is not desirable to put TCP piggyback on UDP and only pure endpoint-to-endpoint TCP connections will satisfy performance aspects properly.

4.2 Architecture

This section will describe the frameworks architecture from a high level point-of-view, evolved from the previously described requirements. The NAT traversal framework, in the previous chapter seen as a black box, will now be separated into smaller components that have to offer specific services to the outside world or for each other. Figure 4.1 gives a first coarse-grained overview to this architecture.

It is important to understand that even if all peers in a NAT-aware P2P network come with the same functionality, not all of them will actually offer the same services. It depends largely on the network environment to what roles they are acting in. Therefore the next section defines these roles, before they can be used in the subsequent section to describe the components in the framework.

4.2.1 Peer Roles

The peers participating in the NAT traversal framework will play one or several roles, depending on their networking environment. Joining the network, a peer will start with the safest role. While communicating with other peers, it discovers the networking environment and decides what other roles it can play to contribute to the P2P network. As the term *contribute* already suggests, some roles request more resources of a peer than others do. Later, the question has to be answered as to how incentives can be offered for *contributing*, in order to avoid free-riders.

Discovering Peer

When a peer first joins a P2P network it does not know anything about its networking environment. In order to function properly, it first has to detect how it is connected to the public internet and under what conditions. Here are some of the questions it has to solve during the first phase while joining the P2P network:

- Is there one or several network address translation devices between me and the rest of the P2P network?
- What is the IP address where other peers can reach me?
- Are some of the ports I need for communication blocked by a firewall?
- What method for NAT traversal can I use, depending on the limits of the NAT device?
- Is the NAT device UPnP enabled and does it support the functionality for programmatic port configuration?
- Is communication limited to either UDP or TCP?

Discovery Responder

A peer acting in this role has to help other peers joining the network to discover their environment. Some discovery problems can not be solved by an application on its own; it needs a node that is known to be available in the public address realm (and which is not NATed). A joining node will send requests to discovery responders, asking for

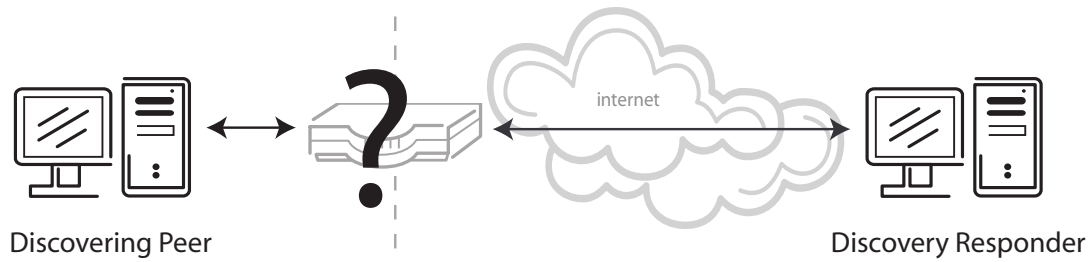


Figure 4.2: The Discovery-Related Roles

assistance. While every peer in the beginning of its lifetime will act as a previously-described Discovering Peer, not every one will become a Discovery Responder. Only peers available from the internet realm can offer this service to others.

Figure 4.2 depicts the discovering peer and the discovery responder role.

Rendezvous Peer

After a peer has found out everything necessary about the networking environment, it knows if it is reachable for other peers in the P2P network. The hole-punching technique requires for every NATed peer at least one peer in the public internet realm. This peer acts as a proxy for all peers registered with it. Anyone intending to establish a direct connection to a NATed peer has to find one of the responsible proxies for it. This proxy role will here be called *Rendezvous Peer*, since it is some kind of meeting point. Only peers that are not behind a NAT device can act in this role.

Attached Peer

The role for peers registered with one or several Rendezvous Peers will be called *Attached Peer*. An Attached Peer usually is behind a NAT device. To be more independent from node-failure, an attached peer should be registered with more than one Rendezvous Peer. Every Rendezvous Peer is just a normal peer which, due to its environment, got a special task. If an attached peer was only registered with one Rendezvous Peer at a time, failure of this node would disconnect the peer from the P2P network.

Initiating Peer

This role is used during establishing a direct connection. The *Initiating Peer* is the one that intends to establish a connection with another peer. Every kind of peer can act in

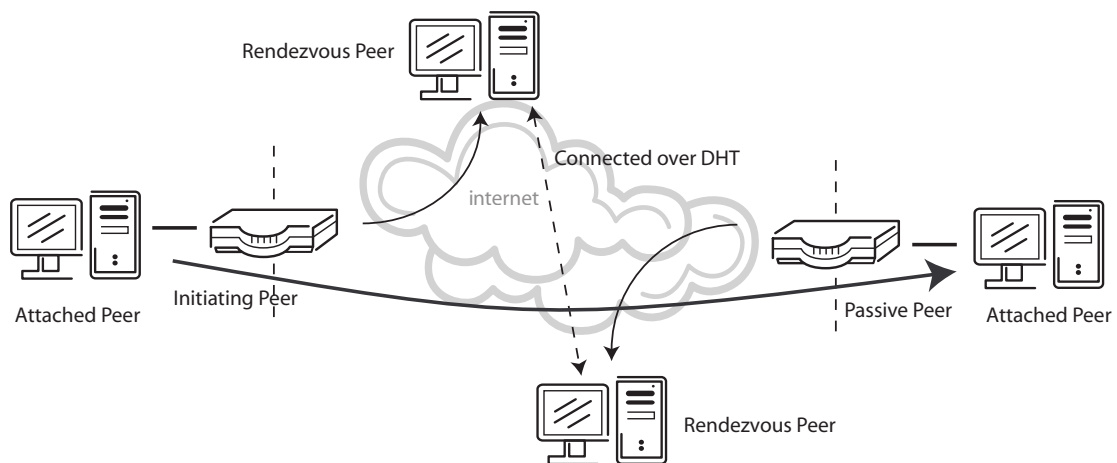


Figure 4.3: The different roles for NAT traversal

this role, independent of whether it is NATed or not. The initiating peer will first have to find the rendezvous peer responsible for the attached peer with whom a connection should be established. This rendezvous peer will then manage the NAT traversal process for the initiating and the attached peer.

Passive Peer

In the same context as for the initiating peer, a *Passive Peer* is the role of the passive endpoint to whom an initiating peer wants to get a connection. Every peer can act as a passive peer, but, of course, its behavior will be the most interesting and elaborate for a NATed peer.

Figure 4.3 illustrates the relation of the different roles to their location in the network environment.

Summary

The last sections have presented the different roles a peer can act in, depending on the network environment they are running in. Table 4.1 shows this in a condensed form; some of the roles can only be *played* by peers behind NAT devices, others only by peers directly available from the internet.

	NATed	not NATed
Discovering Peer	✓	✓
Discovery Responder	-	✓
Rendezvous Peer	-	✓
Attached Peer	✓	-
Initiating Peer	✓	✓
PassivePeer	✓	✓

Table 4.1: Possible locations for the roles

4.2.2 Components

Having the different roles defined which a peer can take, we now can proceed to the definition of the single components of the framework. This section will address the responsibilities and the interaction between the components and not their behavior within. Figure 4.4 depicts this architecture in coarse-grained form.

NAT Broker

The first attempt to design this framework was to use the layer pattern comparable to the ISO/ OSI network stack of protocols. In this pattern, every layer only relies on the services and interfaces of the one below it. This has the advantage of leading to a clean extendable design, since every layer can be altered or exchanged if it keeps the contracts it made for the layer above. During the design phase, it turned out that the layer pattern was only partially applicable to the framework. There was even more need for one managing component, supervising the work of the other components but still hiding their functionalities from the framework user (i.e. application developer).

This led to calling this strong central component a broker, the *NAT Broker*. It takes requests from the application built on top of the framework and forwards them to the proper components within the framework. After those components have finished their task, the broker will be notified and then decides whether the application should be notified too, or if another task has to be executed instead.

Discovery Component

A peer joining the network has to first collect information about the surrounding networking environment before it can decide in which roles it can act. While some discovery tasks can be done without the help of another peer (e.g. UPnP device discovery), others

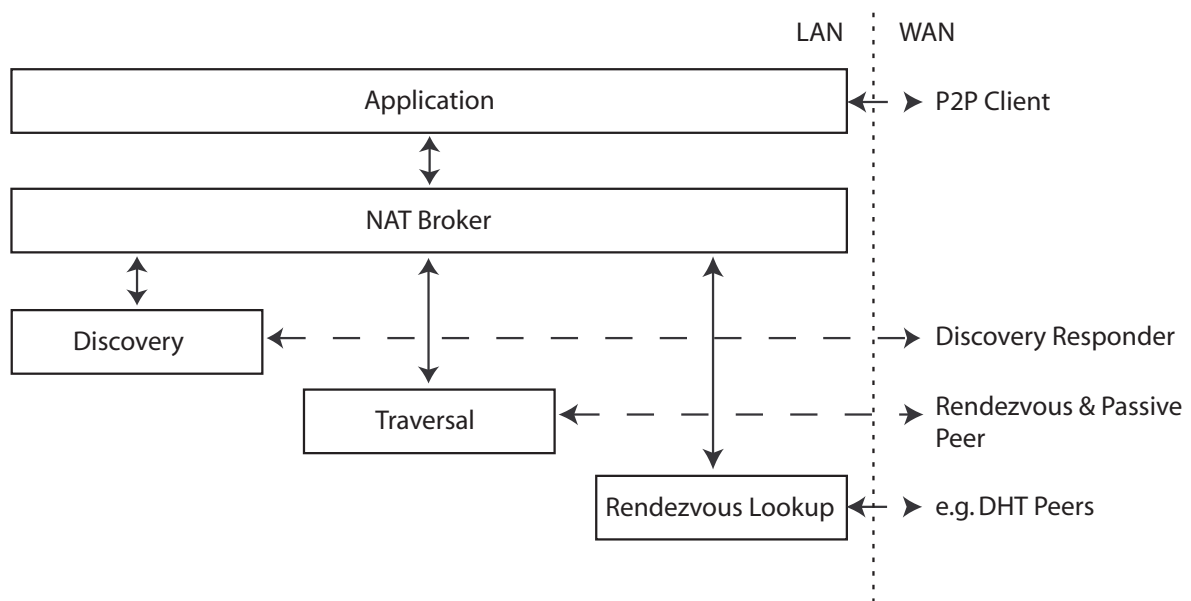


Figure 4.4: The single components and their interactions among each other and with other peers

need the assistance of peers outside their own local sub-net. How can the joining peer find these assistance peers, if not even connected to the P2P network? The answer is simple, yet not completely satisfying: it gets a list of possible peers from the application. But where does the application using the framework get that list from? Starting for the first time, the application will use a list of peers that are highly available and this list will be deployed together with the application and be replaced with more accurate peers during the application's runtime.

The component will contain both the server and the client part for the bilateral environment discovery, because every peer theoretically can play both roles. Discovery tasks requiring another peer are

- Detecting NAT devices on the path to the public internet
- Detecting the translation scheme of the NAT device
- Call-back tests, to see if port forwarding is activated or firewall devices are present
- Traversal tests, to see if an eventually present NAT device is capable of hole punching, for UDP and TCP
- To measure times UDP and TCP connections are kept as entries in the NATs translation list

One possible problem to keep in mind is that the testing might already influence the later behavior of the NAT device. While some tests can not avoid using the same ports as the application, testing on other ports should be preferred whenever possible.

This Component will have to implement the previously described *Discovery Responder* role as well as the *Discovering Peer* role.

Traversal Component

The third main component after the NAT broker and the environment discovery is the NAT Traversal Component. It has to offer the functionality for establishing direct (unrelayed) connections between two NATed peers.

Considering all requirements and the insights gained from comparing related work, hole-punching seems to be the right choice for the main NAT traversal technique. It allows direct—not relayed—connections between two NATed peers and promises a high success rate, at least for UDP-based connections.

While hole-punching will be the main traversal technique, extending the network with other techniques such as UPnP-based configuration should already be considered during the component design. This component will offer at least two methods to the NAT broker: *connectUDP(nodeID, rendezvAddr)* and *connectTCP(nodeID, rendezvAddr)*. When the NAT broker calls one of these two methods, a connection using the specified transport protocols will be established.

As the method's signature shows, the traversal component needs additional information where it can find a rendezvous peer for the peer it wishes to connect to. This information will be obtained by the subsequently described *rendezvous lookup component*.

This traversal component will implement the functionality for the rendezvous role, the attached peer role, the initiating peer role as well as for the passive and active peer role.

Rendezvous Lookup Component

The *Rendezvous Lookup Component* has to provide the traversal component with the IP address of the responsible rendezvous peer. This information can be obtained in various ways. For a basic attempt, a distributed hash table, managed by the NAT broker, will be used for this. While it is clear that this will not be the best solution regarding performance, it will prove the feasibility of the basic concepts. After the prototype is running with this

attempt, it will still be possible to improve the basic attempt with a more sophisticated one.

What could this sophisticated attempt look like? If structured P2P overlays can be used, the position of the rendezvous peers relative to their attached peers will be determined after the role they are playing. Usually in structured overlays, the node ID is calculated more or less randomly, to get a uniform distribution over the ID space and to protect the overlay against malicious nodes. The idea is now to make non-NATed peers responsible for a given ID range, using a function that maps that range to the ID of the rendezvous peer. Every other peer looking up a given ID in that range would first contact the responsible rendezvous peer which he even might find in a local table.

Chat Application

To prove the applicability of the framework, a simple chat application should be written, which uses the framework's public API. Chat applications do not cause heavy network traffic and for a real-world P2P-based solution, relaying the chat data might be a more accurate approach. But since it is supposed to be a proof-of-concept application, the two communication peers have to be able to establish a direct connection between each other. The chat application should give the user the possibility to enter an identification string (e.g. an email URI) of the opposite peer with whom a chat session should be established. In addition, the user can choose if the chat session should be UDP- or TCP-based. For UDP-based sessions, no reliability mechanisms have to be implemented. If a UDP packet—with a chat message as payload—gets lost, no retransmission is necessary, since this is clearly the domain of TCP.

4.3 Alternative Concepts

The design of a complex system is rarely a straight path to the ultimate solution. In case of the NAT traversal framework, other architectures have been taken into consideration. This chapter is going to describe those alternative design attempts, since they might be valuable for other frameworks and also for the one at hand in the future.

4.3.1 Locating the Rendezvous Peer

During the design process of the framework, some elements could be identified early as fixed points necessary in all possible solutions. It became clear that whenever two NATed

peers needed to establish a direct connection, the help of a third party would be needed. How to find this third party was not quite that clear and still leaves a lot of room for improvement.

The finally applied concept uses a distributed hash table for this task. While it is easily applicable it does not perform very well for all kinds of P2P problems. In some cases the chosen attempt will double the lookup traffic over the network. Other solutions might perform better but they would no longer be universal. Nonetheless, those ideas should be presented here.

Brute-Force

Instead of asking how a peer could find the proper rendezvous node, the question could also be altered to where the rendezvous peer should be located to be found easily. One of the first ideas that came up was to locate the rendezvous peer according to all entries in the routing table. That means that if a peer has a routing table with a hundred entries to other nodes, it would keep up connections with all of them and play either attached peer or rendezvous peer according to its network environment.

Some (if not most) P2P routing algorithms use iterative message routing in contrast to recursive routing. With iterative message routing, the original sender of the message keeps in control of every hop to the target, since every node in between tells the sender which node should be contacted in order to get closer to the destination. With recursive routing, the message just gets forwarded from node to node until the message reaches the destination.

Combining the idea of keeping open connections to all nodes in the routing table with recursive message routing would allow even NATed peers to be available and helpful for the message transport. But what are the costs for this approach? A brief example will illustrate this: because the Pastry routing algorithm has a node-ID space of 2^{128} and the size of its routing table is of the order $O(\log(n))$, we will assume that the table contains 100 nodes. Some NAT devices keep UDP connection entries in their translation table only for a few seconds, let us assume 5 seconds. Avoiding those entries to time out, one single client would have to send more than 1200 messages a minute in addition to the common maintenance messages. This large amount of overhead traffic was the reason why this approach was not chosen for the prototype implementation.

Good Neighborhood

Instead of making all nodes responsible as rendezvous peers for each other, a dedicated set of nodes could be selected for this task. In some routing algorithms, each peer keeps a list with its neighbors in the ID space to facilitate the last hop(s) of a message to its destination. Instead of maintaining connections to all nodes in the routing table, only the neighbors could be selected. A peer being directly available from the internet could act as a rendezvous peer for its neighbors.

In real life, this could be compared to a good neighborhood of houses, where everyone is at work except one. The postman knows about this and instead of taking certified mail back to the office, he brings all letters to the only one at home. Since the one at home knows the phone number at work of everyone in his street, he tells them to stop by after work to pick up the letters. This example also makes clear what every approach involving rendezvous peers is built upon: trust and personal sacrifice. The postman and the neighbors need to trust the one at home to notify the respective neighbors about mail and he might not even get an incentive for his work.

4.3.2 Layer Design

While figure 4.4 almost looks like the result of a layer design, there is one major difference, however. The discovery-, traversal- and lookup-component depend on the design of the next higher layer and other components on the same layer. This hurts the basic design principle in a layered architecture. There, every layer only knows about services offered by the layer below and nothing else. Contracts and dependencies are only made in one direction and this keeps the system simple to exchange single layers in it.

A pure layer architecture was considered because of the previously described advantages. The only way to realize it would have been to put the discovery-, traversal- and lookup-component into one big container, which would have offered many completely different services to the broker-layer above. But the advantages of a layer design would have been outweighed by the disadvantages of one single component containing all services at once and so the framework was designed with a central broker component.

4.4 Summary

This chapter has first defined the high- and low-level requirements for the NAT traversal framework, evolved from the thesis objectives and demands that were made by comparing

already existing frameworks. As a tool to understand all necessary situations of a peer in the framework roles were defined. Every role describes the services and tasks a peer will offer and execute according to its knowledge about the network environment and requests made from the outside.

This was followed by a coarse-grained description of the components used in the framework. These components help to discover the network environment of a peer, to locate a specific peer in the network and to establish a direct connection to that peer. A broker hides details about these components from the application developer and manages the workflow between them.

The concluding sections described alternative design principles that were considered during the design phase but could not be selected for the ultimate design.

Chapter 5

Implementation

This chapter describes the implementation of the framework's prototype, the development process and the end product. While it will not cover every single implementation detail, it focuses particularly on important decisions that had been made and problems faced during the development phase.

5.1 Used Libraries

Not every part in a software development project like this can and should be written from scratch. Existing libraries should be used in order to reduce development time and to increase reliability of the whole system. The following subsections explain why libraries were chosen, what their advantages are and give a short introduction in using them. Figure 5.1 depicts where the libraries have been used.

5.1.1 XStream - XML Object Serialization

Since the thesis description requested a XML-based communication protocol, a library supporting this task had to be found. The range of possible solutions goes from a simple XML parser to a relatively complex and sophisticated object-serialization tool. After evaluating a few different libraries ([17],[15]) by doing simple Hello-World-like tests, XStream [34] was chosen among them as the most suitable and powerful solution.

XStream turned out to be a very impressive and yet handy tool for replacing the native Java object-serialization, which has some serious drawbacks in dealing with different versions of the serialized objects. This means that a message serialized and sent over the

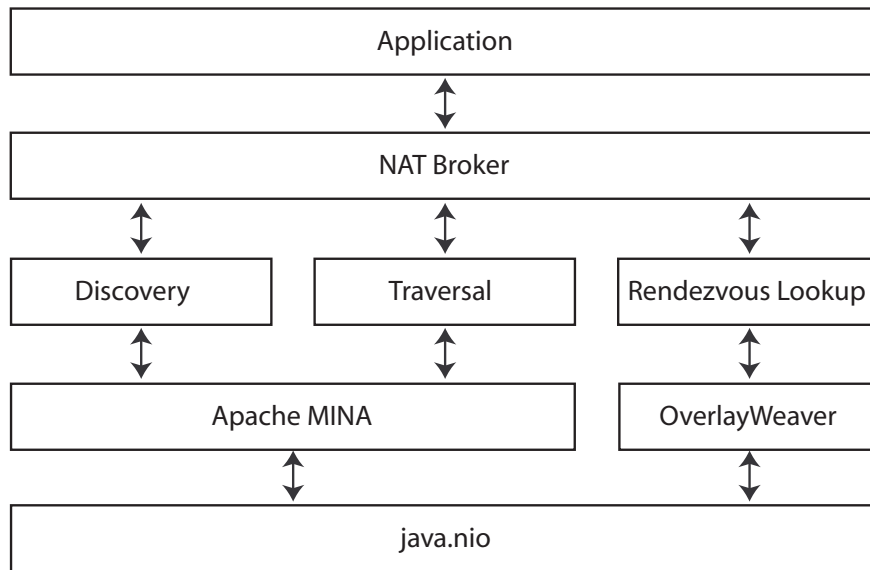


Figure 5.1: The relationship of the components and the used libraries

internet might not be decoded on the receiver side, if both sender and receiver are using different versions of that message class. In addition, native Java object serialization is not interoperable with components of other languages. This would require all communication participants to be written in Java as well and this dependency might break in the future.

Here are the main reasons why XStream was chosen:

- Very easy to use: in only two lines of code, even a complex object can be serialized into a String containing the object description in XML
- Object version independent: small changes, for instance in serialized message objects at the sender side, still keep them readable for an outdated receiver environment (in contrast to the native Java serialization)
- No modification to objects required: the used class inspection technique allows even private and protected object fields to be serialized. However, the library does not serialize fields marked as *transitive*, which can be used to prevent them from being converted to XML.
- Easier debugging: if a certain behavior has to be tested, objects can be read, created and edited easily with simple tools such as a *notepad* or using *telnet* for communications.
- Licensing: XStream is available under the BSD license, which suits the requirement of the thesis prototype

Listing 5.1 shows how to use XStream:

Listing 5.1: Initialization and simple serialization with an XStream object

```
1 XStream xstream = new XStream();  
2 String xml = xstream.toXML(objectToSerialize);
```

5.1.2 Apache MINA

Apache MINA [18] is the library used for network communications in the prototype written for this thesis. To understand why MINA was chosen for this task, a short introduction to Java networking is given in the subsequent paragraphs.

From blocking over non-blocking networking...

Until Java version 1.4, developers of communication applications had to use the stream-oriented package *java.io* for networking. While a lot of simple applications could be written easily with this set of classes, they did not scale well for applications that had to maintain a lot of connections, as it is the case with servers. Since the networking classes would block the applications while waiting for incoming data, most writers of server applications would tend to open a new thread for every new connection/ session it had to handle. But every new thread introduces a overhead to the available system resources and the performance of server applications with hundreds of open connections would suffer severely.

In Java 1.4.0, the New I/O (NIO) was introduced to the Java community and along with it the possibility to handle high performance networking in a much more elegant manner. In *java.nio* a single thread is used to handle all connections. Applications using *java.nio* can set communication to non-blocking-mode, which means that it does not have to wait until a specific job is done. The application first subscribes to certain actions and gets notified if those events occur. This decouples the application from networking and, for instance, allows user interaction to a GUI without opening (and furthermore handling) a new thread every time while sending or receiving data.

Another important change in SUN's networking concept of Java is how the raw data is understood. While in the old networking packet data was always handled as stream, in the new I/O buffers are used. Buffers have a better performance than streams and they also can decouple read and write system-calls from the application.

But *java.nio* is not all good, it also has some drawbacks. *Java.nio* makes it quite hard to separate the business logic from the pure networking task. It is then very complicated to design a clean architecture.

...to a higher Abstraction Level

Since a lot of networking developers had problems to keep their code clean while using *java.nio*, in June 2004 Trustin Lee published a networking framework he called *Netty2*. At that time it was one of the first networking application frameworks for Java that was event-based and it therefore attracted a lot of developers. After a some months, Trustin joined the Apache Directory project and *Netty2* was rewritten and renamed into *MINA*.

MINA reduces the complexity of *java.nio* for the developer and still offers a richness of options. *MINA* encapsulates the *java.nio* classes on a higher abstraction level and does a lot of boring repetitive tasks for the developer. A very powerful feature of *MINA* is the filter chain. In this chain the developer can add different encoders and decoders (codecs) for data encryption (built-in SSL support), or message serialization/ de-serialization. Together with the previously described XStream library, it was possible to write a xml message codec within a few lines of code.

API

The corner-stone of using *MINA* for networking is the class *IoHandlerAdapter*. It does not matter if a server or a client, an UDP- or a TCP- based communication protocol should be written, every time this class has to be overwritten. After a connection to a MINA application has been established and data has been received, the method *messageReceived(IoSession, Object)* gets called. In this method the received messages can be read and actions according to their content can be selected. While *Object* holds the actual message, *IoSession* contains all information necessary about the connection: sender and receiver IP address, idle time, protocol type etc.

In the NAT traversal framework, all applications using it have to extend the *IoHandlerAdapter* to implement their specific protocol behavior. This is convenient since it allows the application developer to maintain full control over the peer-to-peer connection.

5.1.3 Overlay Weaver

Overlay Weaver [19] is a construction framework which supports algorithm designers as well as application developers. It was chosen for the implementation at hand, because it

has some persuasive features:

- Well structured: the decomposition of the different layers of a DHT helps to exchange or change single parts of it
- Already has built-in different routing algorithms: Chord, Kademlia, Pastry and Tapestry. Since the routing algorithms only have to care about how messages are forwarded, they were implemented within only a few hundred lines of code. This helps to change the behavior of it to support NAT traversal.
- Supports iterative and recursive routing:
- Licensing: Overlay Weaver is published under the Apache License, Version 2.0, which complies with the requested GNU Public License

Other similar libraries for this task have been evaluated as well, but all of them have some important drawbacks compared to the Overlay Weaver Toolkit. *FreePastry* ([6]), for instance, is a very powerful framework and a number of applications have been built on top of it, such as ePost [1], a fully-distributed serverless email system, or PeerMart [13], an auction-based market for P2P services. But *FreePastry* has grown over time into a complex and hard-to-understand structure. The comments in the code did not help much to understand the components responsibilities and this would have made it hard to change parts of the framework.

5.2 Other Evaluated Libraries

This section describes libraries that have been evaluated but could not be used. Even if they did not fit entirely into the framework, they are still worth being mentioned and the reason why they could not be used will be explained here.

5.2.1 STUNT

Simple Traversal of UDP Through NATs and TCP too (STUNT, [27]) is a library that extends the STUN protocol to include TCP functionality. This library offers one of the most elaborate TCP hole punching procedures available at the moment. The author has tested it with a wide range of NAT devices and it seems to be very reliable (64% for TCP hole punching).

At the time STUNT was evaluated for the NAT traversal framework, it had some drawbacks, however. According to the requirements, the framework has to support NAT traversal for both UDP and TCP. But STUNT did not support UDP hole-punching yet, and implementing it into the library seemed to be too complicated to manage. In the meantime, the author has added this feature to the library and has made it available to interested testers. It will soon reach a stable state and might therefore be considered for future work on the framework.

But lacking UDP support is not the only negative aspect of the STUNT library. The code is not yet documented very well and extending it would have been a hard and complicated task. The choice for the framework design was either to extend the library with many more necessary features (such as NAT device discovery) or to implement the TCP traversal functionality from scratch according to the described behavior of STUNT.

5.2.2 UPnPLib

UPnPLib is a Java library published under the Apache license that can be used to discover UPnP-enabled devices on the network, to interact with the found devices and to keep track of their states [33]. For the matter of NAT traversal, UPnP is interesting since it allows an application to configure port-forwarding on the gateway device without further user assistance. On the other hand, UPnP is disabled on most end-user devices by default, because it can introduce severe security issues to the network.

The UPnPLib has a simple and yet powerful API and tests could be written very easily with its documentation. The test application could discover the UPnP-enabled Internet Gateway Device (IGD) in the test-environment without knowing its IP address. The application also returned a list with the IGDs capabilities, but failed to setup port-forwarding on the device. The test was repeated with three different devices, a Linksys WRT54GC with original firmware, a Linksys WRT54G Version 3.1 with original firmware and a Linksys WRT54G Version 1.1 with custom firmware. Although these tests are clearly not representative, it was decided to drop the UPnP feature for the first version of the prototype due to missing test equipment, in addition to security concerns about UPnP.

5.3 Implementation of the Components

The following sections will describe how the framework's components have been implemented and what problems had to be solved during the development phase.

5.3.1 NATContinuation Interface

The *NATContinuation* is an interface used repeatedly throughout the framework. This mechanism helps to decouple single components from each other. The continuation will be passed as a parameter to the object that has to do the processing. As soon as the object has finished, it will call either one of the two methods in listing 5.2. To make this work, every class that needs to be called back will have to implement the two methods of the NATContinuation interface.

Listing 5.2: The NATContinuation interface

```
1 public interface NATContinuation {  
2     public void receiveResult(Object result);  
3     public void receiveException(Exception result);  
4 }
```

The *receiveResult* method will be called when the given task has finished successfully, *receiveException* when something unexpected has happened. The parameters will either contain the result of the calculation or an exception with a description of the failure.

The continuation can be seen as a one-use-only variation of the observer/ observable pattern. In networking applications like the NAT traversal framework, requests over the internet can take quite some time or even fail, and components must not block while they are waiting for a response. Especially in a P2P application, where every node is server and client at the same time, the nodes have to be able to handle incoming requests while waiting for a response.

5.3.2 NATBroker

The NATBroker class is the public interface to the traversal framework. It hides all implementation details from the application developer. It is also the manager for the other components in the framework and handles their life-cycle. Listing 5.3 shows how the chat application initializes the NATBroker.

Listing 5.3: Initialization of the NATBroker from the external chat application

```
1 NATBroker broker=new NATBroker(localURI, bootstrapAddr, bootstrapURI);  
2 broker.addProtocol(ChatMessage.class, new ChatProtocolHandler(localURI));  
3 .  
4 broker.datagramConnect(remoteURI, natContinuation, initialMessage);
```

The constructor takes three parameters: the identification of the local node, a list with the socket addresses of possible bootstrap nodes and their node identification names. At

least one of the bootstrap nodes in the list has to be online and reachable, otherwise joining the network will fail.

Line 2 in the listing adds the default protocol handler, which in this case handles incoming chat messages. If the application receives a message of the type *ChatMessage*, the *ChatProtocolHandler* class will be used for processing. After the local node has successfully joined the network, the chat application tries to establish a UDP connection with a node *remoteURI* and sends an *initialMessage* of the type *ChatMessage*. If the NAT traversal to the other node is necessary and successful, the counterpart will know that a chat session should be established, thanks to the *ChatMessage* that has been sent.

The second parameter in the *datagramConnect* call is a reference to the chat application. The NAT broker will use the continuation to notify the chat application about successful connection establishment to the intended peer.

5.3.3 Discovery Component

First the NAT broker has to collect data about the networking environment it is operating in. The discovery component does this by using one of the bootstrap nodes—from the public address realm—as a responder. Several network tests can be implemented; in the current version the discovery responder can handle three types of tests and their messages:

- *BINDING_REQUEST*: named after the corresponding message in the STUN protocol; asks the responder to return the public IP address of the requesting peer. It can then compare its private address with the address it got from the responder. If the two addresses are identical, the peer can presume that there is no NAT device on the way to the internet
- *CALLBACK_REQUEST*: comparing the public and the private address is not sufficient to detect all problems. First, even if the two addresses are different, it still could be reachable from the public internet, e.g. if port forwarding was activated on the NAT gateway. Second, the private and the public address being identical does not prove that the peer is reachable. A firewall could block incoming requests, but does not necessarily need to work as a NAT device. For this reason the call-back test tries to find out if the peer is reachable from the public address realm.
- *TCP_TRAVERSAL_REQUEST*: the used TCP hole-punching technique does not work together with all known models of NAT devices. To find out if the present device supports hole-punching for TCP, this message starts an appropriate test. However, the significance of this test might be questioned, since it can only simulate

one part of the *worst case scenario* with two NATed peers. Because the test is executed with the help of a peer that is most likely not NATed, it might succeed here, but might still fail between two NATed peers.

One problem that had to be solved for all three tests was the time-out handling. How long does it take to be sure that a specific test has failed? The messages from a joining-peer to a bootstrap-peer can travel from a few hops up to many hops if the peers are far away from each other. There also has to be a retry-mechanism if the communication was UDP-based, since UDP messages can get lost without further notification to the sender.

Also the decision to decouple components from each other required additional mechanisms. Since one component can start a process in another component without waiting for completion, an external monitor has to watch for time-out events and notify the appropriate component that a request has failed or did time-out.

All data about the networking environment is collected in data structure called *NATDevice*. It is passed back to the NATBroker when all tests are finished. The NATBroker can then apply this information to initiate the appropriate next steps.

5.3.4 Traversal Component

The traversal component is responsible for all tasks that are directly related to the establishment of a direct connection between two peers. Again, there is a set of messages that two traversal components can exchange. They can be separated into maintenance messages, UDP- and TCP-hole-punching messages. For each request-type there is also an response-type:

Maintenance Messages

- *REGISTER_REQUEST*: the first step of a NATed peer is to register with its rendezvous server. The registration has to be periodically refreshed, in order to keep the entry in the NAT device's translation table alive and to notify the rendezvous server that the attached peer is still alive.
- *LOOKUP_REQUEST*: this message is sent when an attached peer needs to find the responsible rendezvous server for another given peer. In the current implementation, a rendezvous peer receiving such a message will do a look-up in the DHT it is part of and send the answer back to the attached peer for further use.

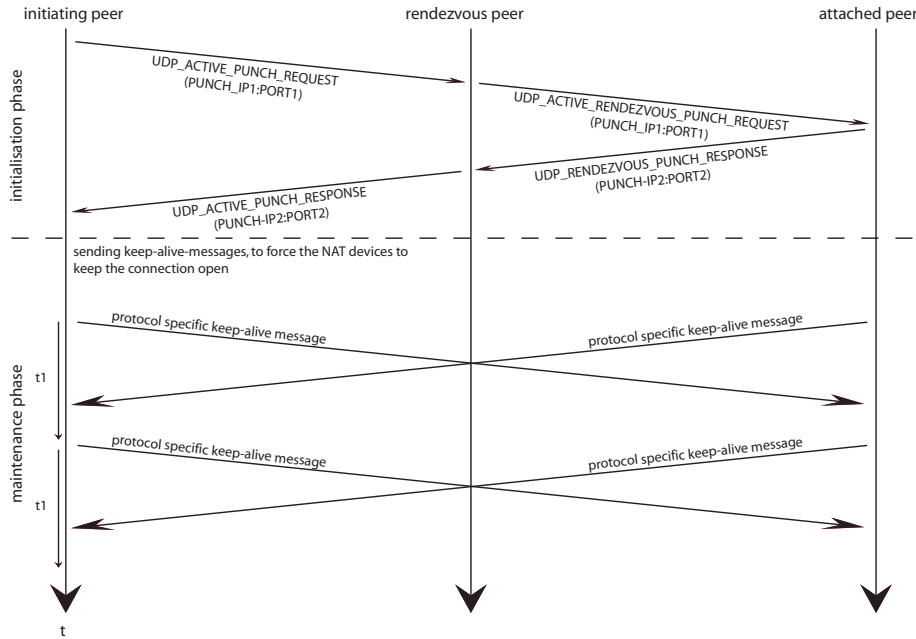


Figure 5.2: The communication protocol used for UDP traversal

UDP Hole-Punching

- *UDP_ACTIVE_PUNCH_REQUEST*: A initiating peer sends this message to the appropriate rendezvous peer it intends to establish a UDP connection with. The rendezvous peer checks in its lists if the given peer really is attached to it and forwards the message, changing it into a *UDP_RENDEZVOUS_PUNCH_REQUEST* message. The message already contains the public IP and port of the initiating peer that should be used for UDP hole-punching. The rendezvous peer has to keep track of this initiation session, since it later has to forward the response of the attached peer back to the proper initiating peer. For this reason, every initiation message has a session ID.
- *UDP_RENDEZVOUS_PUNCH_REQUEST*: this is the message that is forwarded from the rendezvous peer to its attached peer, in case a third peer wants to establish a direct connection. Like the previously described message, it contains the target hole-punching socket address. An attached peer that receives such a message starts sending UDP packets to the given socket address and simultaneously sends a response to its rendezvous peer, containing its own punching address. The rendezvous peer then forwards the message to the initiating peer, which then can start with hole-punching as well. Figure 5.2 illustrates the sequence of messages used for UDP hole-punching.

TCP Hole-Punching

While UDP hole-punching is relatively simple but still more reliable, TCP hole-punching is much more complicated and many factors influence its success. The chosen approach is fairly simple and might be replaced with more sophisticated ones, as described in [11]. On the other hand, it works very well with a specific set of NAT devices, even if it is not clear why some devices work and others do not. TCP—in contrast to UDP—is a stateful, connection-oriented protocol, which usually also requires the NAT device to keep track of the states of a given connection. While some NAT implementations do not care about failed connection sessions, others do and this poses the main problem for this simplistic approach.

TCP traversal is asymmetric, in contrast to UDP traversal. Both participating peers play different roles in one connection attempt. Because success and failure of TCP hole-punching largely depends on the two—or more—NAT devices involved, the traversal component will switch the two roles after a failure and try again. The active peer is the one that actually tries to connect, while the passive peer just punches the hole through the NAT device and then listens for incoming connections.

Punching a hole works as follows: the passive peer opens a connection to the public punching address of the active peer. Ideally it just sends a SYN packet through its own NAT, but not as far as the opposite NAT (see figure 5.3 to understand the TCP Three-Phase-Handshake). If the SYN packet reaches the opposite NAT, it might respond with a RST (reset) message. The near NAT device might interpret the RST signal as what it is: a failure notification. Depending on the NAT's implementation, the according NAT table entry might be deleted and the hole-punching attempt fails.

If no RST is sent (for whatever reason), then an entry for this connection remains in the NAT table and the hole-punching was successful. Depending on the NAT implementation of the local gateway, an incoming request from the appropriate source will be interpreted as a response to the initial hole-punching connection.

Available NAT implementations hold many pitfalls in different behavior: some send RST, others do not. Some interpret even SYN ACKS from the wrong side of the NAT device as legitimate responses others do not. The variety of different behavior can only be compensated with a variety of traversal mechanisms.

Now the messages used for hole-punching will be explained. Again, for every *REQUEST*-type there is a *RESPOND*-type:

- *TCP_PASSIVE_PUNCH_REQUEST*: the initiating peer always tries the simpler passive mode first. It sends this message to the responsible rendezvous peer

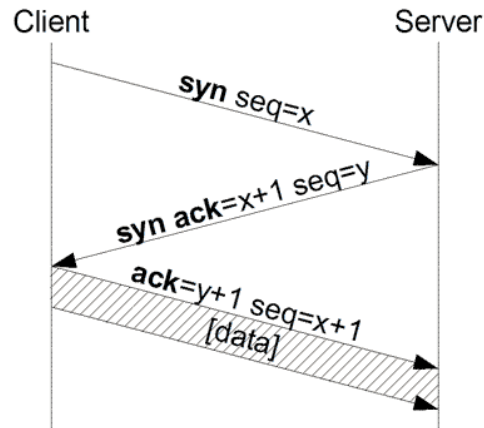


Figure 5.3: TCP's Three-Phase-Handshake

with the ID of the target peer in the header and its own public punch address as payload. It will request the target peer to act in passive mode, i.e. to punch a hole and wait for incoming connections.

- *TCP_PASSIVE_RENDEZVOUS_PUNCH_REQUEST*: after the rendezvous peer gets a passive punch request, it changes the type and forwards it to the appropriate attached peer. As soon as the attached peer gets this message, it punches a hole to the given address, listens for incoming connections and responds to the rendezvous peer with a message containing the own hole-punch-address.
- *TCP_ACTIVE_PUNCH_REQUEST*: this message is sent from the initiating peer to the responsible rendezvous peer to request an active-mode hole-punch attempt. This means that this time the initiating peer will be passive after hole-punching and the attached peer will try to punch through that hole. The initiating peer has switched to active mode, since passive mode did not work. Active mode requires more messages than passive mode, because at the moment the initiating peer starts active mode, it does not yet know the other end-point socket address for hole-punching. Therefore it requests in a first step the public punching address from the attached peer. When it gets the response it punches the hole and sends another message that the attached peer can now start the active connection establishment.
- *TCP_ACTIVE_RENDEZVOUS_PUNCH_REQUEST*: this message is received by the attached peer from the rendezvous peer. As previously described, it chooses a port which the initiating peer can punch its hole to and sends this information back over the rendezvous peer to the initiating peer.
- *TCP_ACTIVE_INIT_REQUEST*: the rendezvous peer receives this message after the initiating peer has finally punched a hole and listens on it for incoming connections.

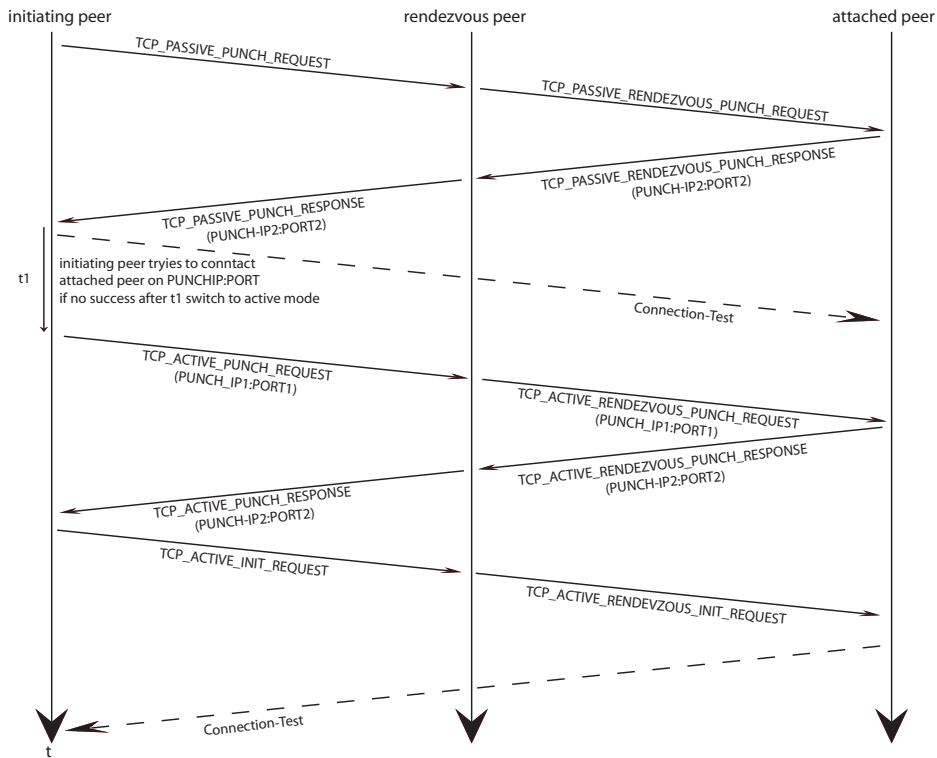


Figure 5.4: The communication protocol used for TCP traversal

- `TCP_ACTIVE_RENDEZVOUS_INIT_REQUEST`: the attached peer receiving this message can assume that the initiating peer has established the punched hole and is listening for incoming connections. The attached peer—now in active mode—tries to open an ordinary direct TCP connection to the initiating peer.

Figure 5.4 illustrates and summarizes the message-flow for TCP hole-punching.

5.3.5 Rendezvous Lookup Component

The task of this component is to locate the responsible rendezvous peer for a given DHT- or node-key. While this can be done in many different ways, for this universal prototype a DHT, distributed over the rendezvous peers, was chosen.

Thanks to the OverlayWeaver library, it was fairly simple to implement this rendezvous lookup DHT. Every time a node chooses a rendezvous peer and attaches to it, this information will be put into the DHT and will be available to all rendezvous peers. If attached peer A needs to know where it can find peer B, it sends a lookup-request to its own rendezvous peer. The rendezvous peer then does a `dht_get("B")` and gets a list with IP addresses of B's possible rendezvous peers. The rendezvous peer then returns this list

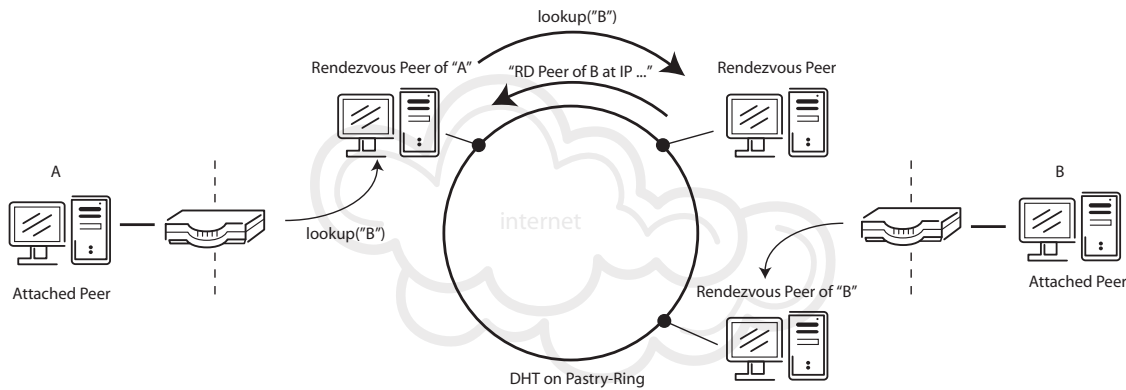


Figure 5.5: The participants in a rendezvous peer lookup process

back to A and it can start to contact those peers on the list. While some of the elements on the list might be outdated and not respond anymore, others will hopefully still be available. See figure 5.5 for an illustration of the described lookup example.

The example with two NATed peers A and B is the worst case. But the mechanism also works with less extreme cases, for instance if A and B are already rendezvous peers and publicly available from the internet. Rendezvous peers just put their own socket address into the DHT. Any peer looking for them will get the right address directly and connection establishment can proceed.

While this DHT-based lookup was rather simple to implement, its performance is not very good. A first-level DHT is used to make a second-level DHT work. Lookups in the second-level ring cause lookups in the first-level ring. On the other hand, it is the only universal solution that came up during the design process. All other solutions would have required changes in the source routing stack of the P2P network, and those changes would have been different for every routing algorithm.

5.3.6 Chat Application

The chat application is not really part of the framework, it is rather more an example of how to use it and to test its functionality. Therefore this section will also be a guide for using the framework for application developers.

The *ChatProtocolHandler* is the central class in the chat application. It basically implements the interface *MessageHandler*. This interface is required by the MINA library to handle different types of incoming messages. It is part of a demultiplexer. Multiplexing in this context means that several protocols can be working on the same port. Which protocol will actually handle an incoming message depends on the type of message. For

instance, one *MessageHandler* implementation can handle *String*-messages, while another one only responds to *Integer*-type messages. This makes connection establishment for a specific (application layer) protocol relatively easy and keeps the framework extendible.

Since the type of the message-object itself triggers the appropriate protocol, developers are encouraged to define new message classes for every protocol. For this reason the class *ChatMessage* was written. It transports the chat message itself but its type helps the framework to decide which protocol will have to handle the message. Every application using the framework has to define its protocols during initialization. On line 2 in listing 5.4, the chat application adds the own message handler *ChatProtocolHandler* to the NAT broker. After the rest of the initialization, the broker will always use the *ChatProtocolHandler* whenever it receives messages of the type *ChatMessage*.

Listing 5.4: Initialization of the NAT broker from within the chat application

```

1 broker=new NATBroker(localURI, rendezvousPeerAddress, rendezvousPeerURI);
2 broker.addProtocol(ChatMessage.class, new ChatProtocolHandler(localURI));
3 chatGUI=new ChatGUI(broker);

```

If the application needs to support additional protocols, the developer just has to write another implementation for the *MessageHandler*- and the *NATPushingProtocol* interfaces and add it, together with the message type, to the broker object.

If the protocol is UDP-based, the developer might consider writing a keep-alive mechanism. This could be necessary, since the translation entry in the NAT device might be deleted after some traffic idle-time. Keep-alive mechanism means that either one of the two endpoints has to periodically send (empty) messages to the other side, just to reset the timer for NAT devices in between. For TCP-based connections, this will not be necessary.

The *ChatTask* class in listing 5.5 is such a keep-alive mechanism and has been implemented as a nested class in the *ChatProtocolHandler*. It extends the *TimerTask* class of the *java.util* package to reuse its functionality. The *ChatTask* will be executed periodically and will send a message to the other endpoint if a session between them exists already.

Listing 5.5: Keep-Alive mechanism for UDP based protocols

```

1 class ChatTask extends TimerTask{
2
3     ChatMessage message;
4
5     public ChatTask(){
6         message=new ChatMessage(ChatMessage.Type.
           CHAT_KEEP_ALIVE_REQUEST, null);

```

```
7      }
8
9      public void run() {
10         try {
11             if (outgoingSession == null) {
12                 if (acceptor != null) {
13                     outgoingSession = acceptor.newSession
14                         (remoteSocketAddress,
15                         localSocketAddress);
16                 }
17             } else {
18                 outgoingSession.write(message);
19             }
20         } catch (IllegalArgumentException e) {}
21     }
```

5.4 Summary

This chapter described the fine-grained design of the NAT traversal framework and its dependencies and contracts made to the outside world. The first sections went over the libraries used in the framework and why they were chosen. This first part was succeeded by a description of libraries that might be interesting for future work on the framework but did not entirely meet the requirements. The last part presented the design and implementation of the single components and how they interact with each other locally and over the network.

Chapter 6

Evaluation

The two preceding chapters described design and implementation of the NAT traversal framework. This chapter will now focus on evaluating the results made so far in regard to various aspects. These aspects are related to clean-room testing and the results received by it and to tests placed in a real-world-environment. The subsequent section compares the initial made requirements with the final result and is followed by an description of good or weak aspects in the prototype implementation.

6.1 Testing

Every application or library under development has to be tested in order to find bugs or design flaws as early as possible. Testing can always be only as good as the environment where the tests are made, but choosing an appropriate setting is not always a trivial task. This weights even heavier for applications which need to make assumptions about the environment they will run in.

For the design of the NAT traversal framework, such assumptions had to be made, although it was clear that the expected real-world environment would show a large variety of behavior. This was a serious dilemma, since, on one hand, the framework has to operate in as many environments as possible and, on the other hand, tests with a representative quantity of devices would not have been feasible.

For this reason, the tests during development time were made with the testing environment as depicted in the subsequent sections.

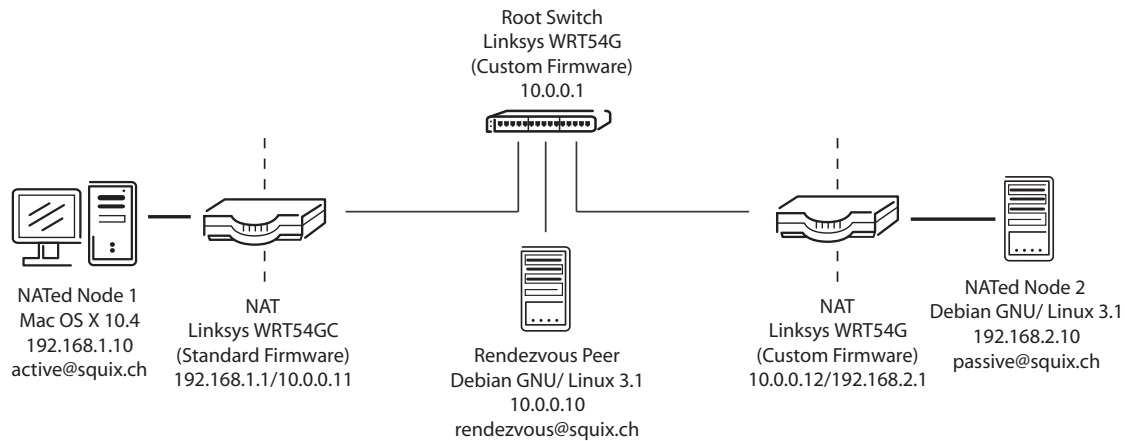


Figure 6.1: Test Setup : three nodes, two of them behind a NAT device

6.1.1 Test Setup

The first setup for testing simulated the real internet in a local network. It consisted of six devices, as illustrated by figure 6.1.

- **Rendezvous Node:** attached directly to the *root* network switch, running on Debian GNU/ Linux 3.1. This node is behaving like a node directly reachable from the public internet, and has always to be started first, because it has to work as a bootstrap node for the other nodes and the DHT overlay
- **NATed Node 1:** attached to the root network switch over a Linux-based NAT device (WRT54G with custom firmware), running on Mac OS X (10.4). Since this node has a graphical front-end, it is usually used for testing the GUI of the Chat Application as well.
- **NATed Node 2:** like the other NATed node attached to the root network switch using a Linux-based NAT device. This node is running on Debian GNU/ Linux 3.1 and can only be accessed over a SSH shell. Because on this node the application is running in text mode a GUI cannot be opened and this node just works as a message responder for chat-testing.

6.1.2 Test Run

This section describes how the framework was tested during and after development. It also gives a brief insight how the framework is expected to work for a flawless run.

1. Starting the rendezvous peer: since this peer acts as a *root*-node to the rest of the network it always has to be started first. In this state of the network startup no other node can help this peer to discover any present NAT devices. Like every peer, this peer gets a list with IP-addresses of possible bootstrap peers. For testing, those IP-addresses are just dummies and will never respond. After unsuccessfully trying to connect to all those dummy-addresses, it assumes to be publicly available and acts from now on as a rendezvous peer. Since it assumes to be a rendezvous peer it also mounts the DHT, which later will be used to lookup the responsible rendezvous peer for a given node ID. As soon as the DHT is mounted, the rendezvous peer publishes itself as the responsible rendezvous peer for its node ID.
2. Starting the two NATed peers: now the two peers behind the NAT devices can be started. Like the rendezvous peer they get a list of IP-addresses of possible bootstrap nodes. Some of them will not respond as well, but one address belongs to the previously started rendezvous peer. As a first step the starting peers will use the rendezvous peer as a discovery responder. This means, that they will send requests to the rendezvous peer and the rendezvous peer will execute those tests. These tests help to find out if there is a NAT device present and what the public IP address might be (please refer to section 5.3.3 for details). When the discovery tests have finished, the peers know about the NAT device on their way to the rendezvous peer and they assume to be NATed.
3. Testing the NAT traversal: as soon as the discovery tests have finished, the peers can register with the rendezvous peer. They are now available to connection establishment requests from other peers. Their presence will be published by the rendezvous peer in the DHT, in order to be available for lookups of other rendezvous peers. On peers with a graphical interface the chat application has started the GUI in the mean time. The test user has now to enter the node-ID of the peer it wants to establish a direct connection to. For the chat application node names with URI-like identifiers have been chosen (see figure 6.1 for the specific URIs). Before starting the actual traversal sequence, the test user has to choose whether the test should be run using a UDP- or a TCP-connection. The framework tries now in the background to establish a direct connection using hole-punching. To do so one of the two participating peers sends unsolicited packets to the other endpoint, which will be dropped by the NAT device. The outgoing connection has produced translation entries on the local NAT device, which is actually the *hole* for incoming requests. The other peer can now send packets in the other direction through the hole. If this first attempt fails for TCP the roles of hole-punching are switched and the other peer starts with hole-punching.

4. Testing the connection: if the hole punching has been successful, a new GUI will open where the user can send chat messages to the other endpoint of the connection. Since the other endpoint was not able to open a chat GUI after a successful hole punching, it just mirrors the received messages back to the sender. In a successful test the own sent message is received from the remote endpoint and printed to the GUI as a new message from the connected endpoint. There is a special case in UDP-based connections: it is possible that UDP-based chat messages sent over the internet get lost, since no reliability mechanisms are built into this protocol. This does not necessarily mean that sending messages between the two chat clients does not work. In this case, multiple messages should be sent and only if none of them is mirrored back to the sender the test run failed.

During all test runs the applications log their states to the console where they can be evaluated by the tester. Message flows are logged as well which helps to identify the exact state of the different nodes in the moment when problems occurred.

6.1.3 Qualitative Test Results

While the previous two sections described the test environment and how the tests were done, this section describes the results in a qualitative manner. Within the scope of this work a qualitative evaluation seemed to be reasonable since quantification tests with many subsequent runs would suffer from test runs tampering each other. This originates in the fact that NAT devices keep connection states in their translation table for a given time. Subsequent test runs can only produce coherent results, after involved NAT devices return to their initial states.

UDP Hole-Punching

UDP-based hole-punching can be seen as the simpler of the two mechanisms, since the protocol itself is stateless. Since there are no special data packets for initializing, maintaining and destroying connections, every UDP packet looks the same to the NAT device; it is just a carrier for data from the local socket address 1 to the remote socket address 2. This also facilitates the hole punching process. And then, UDP-based hole-punching has been successfully used for several years in protocols like SIP.

For this reasons it is not surprising that UDP-hole punching with the framework succeeds in the broad majority of all test runs. The only failures happened in rare occasions when two subsequent test-runs randomly selected the same hole-punching port. This leads to

failure because the applications assumes that the NAT device chooses a specific public port according to the translation rule. If two subsequent test runs choose the same local port but with a different destination port, the NAT device applies a different translation rule, since the port is already used by the previous run.

These failures have been reduced with a list of *already used ports*. The application will not choose twice the same port during its lifetime, but this list does currently not survive the stop of the test application and the failures can still occur after repeatedly restarting it.

TCP Hole-Punching

TCP is compared to UDP a much more complicated protocol. It is connection-oriented and every connection can have different states. Timing and low-level network options play for TCP-hole-punching a far more important role than for UDP. Following the higher complexity of TCP, the NAT devices show more diverse behavior for this protocol than for UDP. Some of them forward connections only within tight constraints, following the TCP specifications. Others are more docile—not to say naïve—if it comes to treating TCP connections.

With this background, TCP hole-punching must currently be seen as a best-effort technique. If it succeeds, every single detail worked out. If it does not succeed something else has to be tried out until the mechanism runs out of options. The current implementation has just two options, but there are more that can be implemented in the future. The two options are to try the asymmetrical connection establishment process from both involved sides. First from one side, if that fails from the other one.

Now understanding the complexity of TCP hole-punching, it is comprehensible that its success-rate is not as high as the one for UDP. Even with a relatively homogenous networking equipment it succeeds in at most two-thirds of all test runs. This clearly leaves enough space for future improvement, probably by adding more *options* to the protocol, to proceed if one attempt fails.

6.1.4 Deployment

In order to deploy the code to the involved test nodes in a fast and accurate manner an ANT-script was written. After compiling the Java source-code the newly built binaries were packed automatically into a JAR-archive and then copied to the remote nodes over an SSH/ SCP session. For not slowing down the deployment process, the external libraries

were not included into the main JAR. They were only copied manually once to the remote nodes.

6.2 Real World Tests

Especially because all available NAT devices can behave in so many different ways, a large-scale test would be necessary to further improve the framework. Simulations on a large cluster-platform such as PlanetLab [20] would not help in this matter, since the behavior of the NAT device is usually proprietary and not directly accessible to the public. Therefore a simulation could not cover the large variety of NAT behavior occurring *in the wild*.

Besides testing the framework's NAT-accuracy, a simulation could give a measurement for its efficiency, especially when tested together with an application using the framework. In the current implementation, an important component that might require further investigation is the rendezvous lookup mechanism. It currently uses a distributed hash table to locate a rendezvous peer responsible for a given node ID. The external library used for this component is called OverlayWeaver [19]. Tests about OverlayWeaver's performance have been made by its Japanese developer, but the test documentation is not available in english yet.

How could a real world test look like? Besides testing the network performance the most interesting measurement would be the rate of successful established direct connections between two NATed peers. A test application using the framework could be written which would be deployed to a possibly large number of test clients. Each node in this large-scale test-setup would get a specific ID from a given list of ID's. A running test application would then try to establish a direct connection, in random intervals and to random peers. The success or failure of those attempts for all nodes would then be logged and sent to a central server, where it could be analyzed and interpreted. The complexity of such a test would have exceeded the possibilities of this diploma thesis, but might be an interesting task for further investigation. Inspiration for NAT related real-world tests can be found in [11].

6.3 Validation and Verification

This section compares the resulting framework with the requirements demanded in chapter 4.1. This evaluation will briefly describe if and how the requirements have been met.

6.3.1 High-Level Requirements

- **Configuration-Free Setup:** This requirement stressed the demand for P2P applications running without complicated configuration and can be seen as a very central one. The prototype of the framework shows that this need can be satisfied; direct connection between two arbitrary NATed peers can be established using the framework without configuring the gateway device. However, it is possible, but the reliability of the traversal mechanism can be further improved.
- **Simple Application Programming:** even the most elaborate framework is of no use if it cannot be applied to solve actual problems. The example chat application could be written just by implementing two classes: a simple message class and a message-handler class for the actual communication mechanism. To set this in contrast to GUI programming: the graphical interface of the chat application required four additional classes only for the GUI.
- **Application Decoupling:** the idea behind this requirement was to protect the actual application from blocking while framework-internal operations are waiting to complete. This could be realized using the callback mechanism described in section 5.3.1. The application on top of the framework gives a reference to itself while calling a possibly blocking method. If the process in the method has finished, it calls the application with the result.

6.3.2 Low-Level Requirements

- **Programming Language:** the requirement was to use at least Java version 1.4. During the implementation several features of the newer Java 1.5 seemed to be very helpful and this version was chosen for implementation. The new features of version 1.5 that were used are *Generics*, *Typesafe Enumerations* and *Static imports*.
- **Messaging:** all messages that go over network are XML conform, as requested by the requirements. This turned out to be very helpful during the debugging process, since all sent or received messages could be logged to the console. This facilitated to find errors in the code. Simple messages could also be written with *telnet* in order to test the functionality of the receiver instead of writing a separate Java application for it.
- **Networking:** as required, NAT traversal for both UDP and TCP has been implemented. No piggyback TCP-on-UDP protocol had to be used

6.4 Prototype: Pros & Cons

This section points out important aspects in the framework design and shows its weaknesses and strengths, such as security concerns and reliability in general.

6.4.1 Security

Although the framework is well prepared to allow secure communication between two peers, there are no security features implemented yet. This section will describe possible security problems present in the current implementation of the framework. This description is supposed to facilitate further development on the framework in regard to security.

Data Transmission

At the moment, the framework does not encrypt any data going over the wire. MINA—the external networking library used by the framework—supports SSL with only a few lines of additional code. But encryption only makes sense with an appropriate key-/credentials-management. And in addition, security in data transmission does not only mean that the communication can not be eavesdropped, but also that all participants are who they claim to be (authentication) and that they have the right to participate (authorization).

Questions about security in data transmission lead to more questions. Who should possibly be able to participate in a P2P network built on the framework? Who could possibly want to take advantage in weaknesses in security?

Possible Attacks

In the current version, the framework and peers running it are not protected against malicious nodes in any way. Since any security mechanisms like data encryption or authentication are missing, malicious nodes can easily infiltrate the network on which the framework is relying on. For instance, malicious nodes can try to become rendezvous servers for NATed nodes which would be the perfect position for a man-in-the-middle-attack. Or even worse, a malicious node could lie in the role of a discovery responder about the networking environment of the discovering peer. This would even push unNATed peers into a defensive role and put them on a tight leash to the malicious node.

Since rendezvous servers are using a DHT to look for the responsible rendezvous peer, a malicious node could just *spam* this distributed directory listing with wrong entries. Doing this, they could attract connection requests for peers to them, even if they are not responsible.

The list of possible attacks could be further extended. But these examples just tried to point out that the framework is not safe against deliberate disturbance yet.

6.4.2 Reliability

In the clean test-room environment—without malicious nodes—the framework has proved to work reliably. A single node failing does usually not disturb the rest of the nodes, assumed it is not the only node of its kind. But the current version is only a prototype and has to be seen as such. Building a more sophisticated application than the chat program on top of the framework would most possibly confront with new demands or problems.

Taking malicious nodes into account, it also has to be mentioned that the reliability of the framework can easily be tampered with deliberately injected messages. This problem will be solved as soon as the previously described security mechanisms will be implemented.

6.5 Summary

This chapter has evaluated this thesis under several different aspects. First, the environment used to test the framework's functionality has been described, followed by the description of a common test run with a brief description of the processes running in the background and the results expected for the given phase in the run. This was followed by a qualitative evaluation of the actual NAT traversal mechanisms, for both UDP and TCP separately.

Since these clean-room tests only can cover a given aspect of testing, the need for real-world tests has been stated too. Real-world tests would give an interesting insight in the framework's behavior on a large variety of devices, but within the scope of this work it was not possible to cover this topic.

The subsequent section compared the initial requirements with the actual results. This was separated again in high- and low-level requirements and showed that all requirements have been met.

The concluding sections evaluated the framework under the aspects of security and reliability and showed that the framework might need further investigation in these areas.

Chapter 7

Summary

This work has addressed the problems that can emerge when Peer-To-Peer applications are running in a networking environment together with Network Address Translation devices. In a first part it has introduced the terms and concepts important to the P2P field, such as network topologies and distributed hash tables. It has explained how NAT works and has grouped the possible behaviors of NAT devices into four groups. Finally, it has described different possibilities for NAT traversal, such as port forwarding, UPnP, application layer gateways and variations of hole-punching.

With this background, a second part has shown what work has already been done in solving the NAT traversal problem related to P2P networks. Four different approaches—NATTrav, STUNT, JXTA and Skype—have been described with their advantages and disadvantages. Aspects open for improvement present in those frameworks were identified and helped to evolve a list of requirements that a new framework has to meet, together with requirements that originated in the task description for this diploma thesis.

These requirements have been described in more detail in a third part, split up in high-level requirements which treated the system like a black-box and low level-requirements which have set the constraints within the framework. Built upon these limits a coarse-grained framework architecture was developed and described, separated in single framework components and in roles those components eventually will act in. On the path to this ultimate design other possibilities had to be considered but could not be chosen at the end. These alternatives have been covered here as well.

A fourth part described the fine-grained architecture of the framework and explained which external programming libraries had been used and why they were chosen at all. Other libraries showed interesting features and were presented as well, although they could not be used in the framework. Insights to the internal behavior of the framework's

components and a description of the developed communication protocols used by them concluded this part.

The last main part of this thesis evaluated the accomplished work under several different aspects. The environment in which the framework had been tested in, was described as well as the tests themselves and their results. As necessary for every quality management process, the resulting product had been compared to the initial specifications and requirements. The description of specific advantages and disadvantages of the framework covered security and reliability and concluded the evaluation and showed where the framework offers space for improvement and where it already has reached a satisfying state.

7.1 Conclusions

This work has presented a new approach to the NAT problem that many of today's P2P applications have to face. The developed framework improves other similar systems in several aspects: it supports *both* UDP- and TCP-based NAT-traversal, does not rely on single-point-of-failures, does not use the relay technique which introduces bottlenecks to data transfer and is open-sourced and available to everyone accepting the terms of the GNU public license. In addition, a simple chat application using this framework has showed that its integration into other applications can be done with only two extra classes, due to its small and simple programming interface.

However, how good the used NAT traversal techniques actually work together with the large variety of available and deployed NAT devices in the real world could not be investigated, since this would have been enough work for yet another diploma thesis.

7.2 Future Work

Although the framework does not use the most elaborate TCP hole-punching algorithm available it provides already good results. Still, the integration of more sophisticated algorithms like the one used in STUNT [27] could further improve its reliability. Since the first evaluation of this library, it has been further improved and might now meet the necessary requirements to be used by the framework at hand.

This framework was designed to have a universal architecture in order to be usable by many different P2P networks. This universality came at the costs of performance. In the future, the framework could be altered according to the alternative concepts presented

in section 4.3. These changes could improve the performance, although the framework would lose its universal applicability.

As stated in 6.4.1, this framework has no implemented security mechanisms yet. Although the framework is well prepared for implementing security mechanisms such as encrypted data transfer, this topic must not be neglected and needs further work.

Bibliography

- [1] ePost, a serverless email system, based on P2P technology, link: <http://www.epostmail.org/>, last visited: 10.5.2006.
- [2] JXTA, Sun Microsystems, Java-based P2P Framework, link: <http://www.jxta.org>, last visited: 7.05.2006, 12 2005.
- [3] P Backx, T Wauters, B Dhoedt, and P Demeester. A comparison of peer-to-peer architectures. *Eurescom Summit*, 2002.
- [4] SA Baset and H Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *Columbia University, New York, NY, Tech. Rep. CUCS-039-04, . . .*, 2004.
- [5] CH Ding, S Nutanong, and R Buyya. P2P Networks for Content Sharing. *Arxiv preprint cs.DC/0402018*, 2004.
- [6] P Druschel, E Engineer, R Gil, J Hoyer, YC Hu, and S Freepastry, open-source implementation of the P2P routing algorithm Pastry, link: <http://freepastry.org/FreePastry/>, last visited: 11.5.2006.
- [7] eEye. December 20, 2001 - eEye® Digital Security Announces Major Vulnerabilities in Default Installations of Windows XP and Certain Installations of Windows ME and 98, link: <http://www.eeye.com/html/press/PR20011220.html>, last visited: 15.4.2006, 2001.
- [8] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), Mai 1994. Obsoleted by RFC 3022.
- [9] JL Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. *reports-archive.adm.cs.cmu.edu*, 2005.
- [10] GNU.org. GNU General Public License, link: <http://www.gnu.org/copyleft/gpl.html>, last visited: 20.04.2006.

- [11] S Guha and P Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. *guha.cc*, 2005.
- [12] F Harrell, Y Hu, G Wang, and H Xia. Survey of locating & routing in peer-to-peer systems. *cse.ucsd.edu*.
- [13] D Hausheer and B Stiller. Peermart: the technology for a distributed auction-based market for peer-to-peer services. *Communications, 2005. ICC 2005. 2005 IEEE International ...*, 2005.
- [14] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. RFC 3027 (Informational), January 2001.
- [15] JOX. Java Object XML, serializes Java objects into XML, link: <http://www.wutka.com/jox.html>, last visited: 10.5.2006.
- [16] J Kubiawicz, D Bindel, Y Chen, and S Czerwinski. OceanStore: An Architecture for Global-Scale Persistent Storage. *ACM SIGPLAN NOTICES*, 2000.
- [17] P. LeHégaret. KOML, Koala Object Markup Language, link: <http://koala.ilog.fr/XML/serialization/>, last visited: 10.5.2006.
- [18] MINA. (Multipurpose Infrastructure for Network Applications), link: <http://directory.apache.org/subprojects/mina/>, last visited: 17.04.2006.
- [19] OverlayWeaver. Overlay construction toolkit, link: <http://overlayweaver.sf.net/>, last visited: 17.04.2006, 2006.
- [20] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services, link: <http://www.planet-lab.org>, last visited: 6.05.2006.
- [21] C Plaxton, R Rajaraman, and A Richa. Accessing nearby copies of replicated objects in a distributed system. *Proceedings of the Symposium of Parallel Algorithms and ...*, 1997.
- [22] S Ratnasamy, P Francis, M Handley, R Karp, and S A scalable content-addressable network. *Proceedings of the ACM Special Interest Group on Data Communications Conference, San Diego, NY, 161-172*, 2001.
- [23] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), Juni 2002. Updated by RFCs 3265, 3853, 4320.

- [24] J Rosenberg and J Weinberger. Traversal using relay nat (turn) draft-rosenbergmidcom-turn-01, work in progress. *Internet Engineering Task Force July*, 2004.
- [25] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), März 2003.
- [26] A Rowstron and P Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems ...*, 2001.
- [27] Paul Francis Saikat Guha. STUNT, Simple Traversal of UDP Through NATs and TCP too, link: <http://nutss.gforge.cis.cornell.edu/stunt.php>, last visited 25.04.2006.
- [28] C Schmidt and M Parashar. Peer to peer information storage and discovery systems. *forthcoming*, 2004.
- [29] Skype. Closed source, P2P based Voice-over-IP application, link: <http://skype.com>, last visited 28.04.2006.
- [30] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), Januar 2001.
- [31] I Stoica, R Morris, D Karger, MF Kaashoek, and H Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 2001.
- [32] UPnP. Universal plug 'n' play group, link: <http://www.upnp.org>, last visited: 21.11.2005.
- [33] UPnPLib. Java library for programmatically configure UPnP enabled device, link: <http://www.sbbi.net/site/upnp/index.html>, last visited: 28.04.2006.
- [34] XStream. Simple library to serialize objects to XML and back again, link: <http://xstream.codehaus.org/>, last visited: 17.04.2006, 2006.
- [35] BY Zhao, J Kubiawicz, and AD Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. *Computer*, 2001.

Abbreviations

ADSL	Asymmetric Digital Subscriber Line
ALG	Application Layer Gateway
DHT	Distributed Hash Table
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
ID	Identification (Number)
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
MINA	Multipurpose Infrastructure for Network Applications
NAT	Network Address Translation
P2P	Peer-To-Peer
RD	Rendezvous (Peer)
SCP	Secure Copy
SIP	Session Initiation Protocol
SSH	Secure Shell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VoIP	Voice-over-IP
WAN	Wide Area Network
XML	Extensible Markup Language

List of Figures

2.1	P2P network topologies according to [5]: a) Centralized, b) Ring, c) Hierarchical, d) Decentralized, e) Hybrid Centralized/ Decentralized	8
2.2	Full Cone NAT	13
2.3	Restricted Cone NAT	13
2.4	Symmetric NAT	14
2.5	Iterative Message Routing: Peer A has to contact unknown peers until it reaches destination D	19
4.1	The framework's architecture from a high-level view	30
4.2	The Discovery-Related Roles	32
4.3	The different roles for NAT traversal	33
4.4	The single components and their interactions among each other and with other peers	35
5.1	The relationship of the components and the used libraries	42
5.2	The communication protocol used for UDP traversal	50
5.3	TCP's Three-Phase-Handshake	52
5.4	The communication protocol used for TCP traversal	53
5.5	The participants in a rendezvous peer lookup process	54
6.1	Test Setup : three nodes, two of them behind a NAT device	58

List of Tables

2.1	Comparison of the five previously described DHT algorithms according to [12]. N stands for the number of nodes in the network, d for the number of dimensions in CAN	11
2.2	The four different NAT types summarized with their internal translation entries according to the examples in the previous figures.	15
3.1	Four existing NAT traversal frameworks in comparison	24
4.1	Possible locations for the roles	34

Listings

5.1	Initialization and simple serialization with an XStream object	43
5.2	The NATContinuation interface	47
5.3	Initialization of the NATBroker from the external chat application	47
5.4	Initialization of the NAT broker from within the chat application	55
5.5	Keep-Alive mechanism for UDP based protocols	55

Appendix A

Installation Guidelines

This manual describes how to install and use the NAT traversal framework. For instructions of how to use it in your application, please refer to the example chat application and its description in section 5.3.6.

A.1 Requirements

The following requirements have to be met on all nodes in order to be able to run the example chat application:

- Java 1.5 installed
- At least three nodes necessary, two behind a NAT device, one (the rendezvous peer) accessible on ports 4320, 4321 and 4322. Configuration with port forwarding possible in order to simulate a unNATed node. The NAT devices for the other two nodes do not need to be configured in any way.
- All nodes have to be connected somehow, either over the internet or in a local simulation environment (see figure 6.1 for an example).
- On Unix-like systems it might be necessary to configure the *hosts* file. Java sometimes binds listening ports to the wrong interface if the *hosts* file is not configured appropriately in the *localhost* section.
- Dependencies for external libraries: nlog4j (version 1.2.22), mina (version 0.9.1) xstream (version 1.1.3 and xpp3 (version 1.1.3.4) and OverlayWeaver (version 0.1). They are already included in the *fat* JAR-file

A.2 Running the example chat application

Under any circumstances, the rendezvous peer has to be started first, since it helps all other peers to discover their network environment. The rendezvous peer will try to contact the IP addresses passed to it on the command line. Since no other peer is running in this moment, no other peer will respond and the rendezvous peer will just *assume* that it is not NATed. After the rendezvous peer has completely started up, all other peers can be started as well, simultaneously or sequentially.

To start a peer on a system with Graphical User Interface use

```
1 java -jar P2PChatClient.jar
```

on the command line or just double-click the P2PChatClient.jar file. On headless machines please use

```
1 java -cp P2PChatClient.jar net.natlib.NATBroker localURI bootstrapIP:port  
bootstrapURI
```

The parameter *localURI* is the identifier for the local node, *remoteIP : port* gives the socket-address of the bootstrap-node and *bootstrapURI* the bootstrap node's identifier, e.g. *bootstrap@squix.ch*.

A.3 Using the framework

To use the framework in a new application the library-jar file has to be added as external library to the project (e.g. in an IDE like eclipse) or be located according to the classpath parameter.

For details of how to use the framework please refer to section 5.3.6 and to the JavaDoc documentation on the CD.

Appendix B

Contents of the CD

The CD contains:

- Source-Code of the framework
- JavaDoc documentation
- JAR-File with the compiled classes of the framework
- Runnable JAR-File *P2PChatClient.jar* with the example chat application
- This diploma thesis as PDF-file
- The Latex-sources of this thesis and PDF-files of the used illustrations
- Abstracts of this thesis in German and English