Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Engineering and
Networks Laboratory

Firewall and NAT Traversal for Peer-to-Peer Storage Nodes

(SA-2006-23)

by Marc Müller (muemarc@ee.ethz.ch)
July 2006

Supervisors: Marcel Baur (marcel.baur@tik.ee.ethz.ch), Germano Caronni
Professor: Prof. Bernhard Plattner

Abstract

Celeste is a peer-to-peer storage system and designed to work in the public routed internet. Thus, peers behind Firewall or NAT are not supported. So, what is the motivation for enabling peers behind Firewall or NAT? The motivation is, that there will be a large amount of nodes in private or corporate networks. So, it is important to enclose this relative big part of users.
This work compares existing solutions for the stated problem. Based on these, a Firewall and NAT traversal concept for Celeste is worked out. The solution uses a method called "TCP hole punching" for establishing a connection between two peers behind NAT. Due to the variety of different NAT implementations in the market, this method does not work always. A backup mechanism ensures connectivity. In any case where a direct connection fails, a relayed connection through another peer from the network is used.
An evaluation version of the developed concept was implemented in java. Based on preliminary experiments with the implementation, we can conclude, that the presented concept works. But no thorough tests have been done in the lab and must be done in the future.

Zusammenfassung

Celeste ist ein peer-to-peer Speichersystem und entworfen um im Internet mit öffentlichen Adressen zu funktionieren. Dadurch werden keine Celesteknoten hinter einer Firewall oder NAT unterstützt. Was ist die Motivation Knoten zu unterstützen, welche sich hinter einer Firewall oder NAT befinden? Die Motivation ist, dass es sehr viele Knoten in Privat- oder Firmennetzwerken sein werden. Es ist wichtig diesen relativ grossen Anteil an Knoten mit einzubeziehen.
Diese Arbeit vergleicht bestehende Lösungen für das genannte Problem. Basierend auf diesen wird ein Firewall und NAT traversal Konzept erarbeitet. Die Lösung verwendet, für das Herstellen einer direkten Verbindung zwischen zwei Knoten hinter NAT, eine Methode welche sich "TCP hole punching" nennt. Auf Grund der Vielfalt verschiedener Implementationen von NAT Geräten auf dem Markt, funktioniert diese Methode nicht immer. Ein Backup Mechanismus garantiert Konnektivität. In jedem Fall, wo eine direkte Verbindung fehlschlägt, wird eine, über einen anderen Knoten des Netzwerkes, weitergeleitete Verbindung benutzt.
Eine Testversion des entwickelten Konzepts wurde in Java implementiert. Basierend auf ersten Versuchen mit der Implementation, kann man folgern, dass das präsentierte Konzept funktioniert. Allerdings wurden keine gründlichen Versuche unter Laborbedingungen durchgeführt und müssen zu einem späteren Zeitpunkt noch gemacht werden.

Table of Contents 1

1. Introduction

1.1 Situation

Celeste is a system for distributed secure storage on top of an untrusted peer-to-peer network. Files and directories are redundantly stored on untrusted peer-to-peer nodes. Peer-to-peer nodes communicate using the TCP protocol. For more details visit reference [1], a white paper about celeste.
The current system does not support peer-to-peer nodes using network address translation (NAT) or being behind a firewall. NAT devices usually reject incoming TCP connections and thus break many existing IP applications. Firewalls can block traffic on particular ports.
For a peer-to-peer system to work efficient, the majority of the peers should not be behind a NAT or firewall device. Hence, the goal of this work is to identify, evaluate and implement concepts to enable TCP communication for a minority of clients behind a NAT or firewall device. There exist some possible solutions for this problem, each with its own advantages and drawbacks.

1.2 Approach

First, a storage node must be able to detect the presence of a NAT or firewall device between itself and the public internet. Second, as some traversal techniques need to know the NAT type, it is useful to learn the type of NAT method applied by this device.

The work consists of several parts:

1. For NAT traversal, a basic knowledge of different NAT types and behaviour is necessary. We explain NAT and various NAT/peer scenarios in chapter 2.
2. We then review existing possible solutions for NAT traversal. Among the evaluated techniques are STUNT, NATBlaster, P2PNAT, TURN, uPNP and MIDCOM.
3. We evaluate the existing solutions according to a set of requirements: A good solution for Celeste must work completely distributed, must be compatible with the system of each Celeste peer and must establish a connection for almost every network situation. Each solution has some restrictions that the environment has to meet for a proper functionality.
4. Once an adequate concept has been chosen, a prototype is implemented in Java as a proof-of-concept and starting point for a later implementation in Celeste.
5. The last part is to check the correct function of the prototype, to evaluate its performance and to report the results.

2. Background

This chapter gives a brief review of TCP/IP and UDP/IP networks. One focus is put on describing the different NAT types and their behaviour. This background knowledge is essential to develop a NAT traversal technique for Celeste.

2.1 IP Packet and Address

IP Packet:

The Internet protocol (IP) builds the network layer of the OSI reference model [13]. An IP packet is the basic unit to send information to any host in the internet. Each packet consists of a header and a data payload. The parts of the IPv4 header that are relevant for this work are:

- Source Address – IP address of the sender of this packet.
- Destination Address – IP address of the receiver of this packet.
- TTL – Time to live indicates over how many hops, i.e. how many nodes, this packet has to be carried on until it has to be discarded.
- Protocol – Type of protocol used for the packet in the payload (e.g. TCP or UDP).

The payload contains arbitrary data , for example a TCP or UDP packet, with a specified maximum length.

IP Address:

Each host or device in the internet has its own unique IP address, that identifies it for IP packets. An IP address consists of four octets. Important in the context of NAT is, that the IP address space is split into publicly routed addresses and so-called private addresses. Private IP addresses are not routed in the public internet, thus hosts with a private address are not directly reachable by hosts from the public internet (the only possibility to reach them is via NAT). The motivation for private addresses is that today, the IP address space is too small to supply a unique IP address for each host. Furthermore, security can be improved in a private network by hiding hosts from the public internet. Figure 1 shows special IP address ranges. In figure 1 the CIDR address notation is used. Classless Interdomain Routing (CIDR) uses subnetting instead of the IP classes. So, a CIDR address consists of an IP address and a subnet mask.

Address Space Name	CIDR Address Notation	IP address range
Private IP addresses	10.0.0.0 / 8	10.x.x.x
	172.16.0.0 / 12	172.16.x.x - 172.31.x.x
	192.168.0.0 /16	192.168.x.x
Link Local IP addresses	169.254.0.0 / 16	169.254.x.x
Loopback IP addresses	127.0.0.0 / 8	127.x.x.x
Multicast IP addresses	224.0.0.0 / 3	224.x.x.x - 239.x.x.x
Reserved IP addresses	240.0.0.0 / 4	240.x.x.x - 255.x.x.x

Figure 1: Special IP address ranges

2.2 UDP and TCP

User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) are two implementations of the transport layer in an IP network stack. These protocols associate a packet, received from the internet, with an application on the receiver host. Association is done using port numbers. A port is a logical address represented as integer in the range between 0 and 65535. Each packet has a source port (specifies the process at the sender) and a destination port (specifies the process at the receiver).

UDP – User Datagram Protocol

The UDP protocol is described in RFC 768 [2]. UDP is a very simple and connectionless protocol, which does not guarantees successful packet reception at the receiver. The receiver does not acknowledge a received packet. A UDP packet consists of a packet header with source / destination port pair and a data payload.

TCP – Transmission Control Protocol

The TCP protocol is described in RFC 793 [3]. TCP is a connection-oriented protocol. Before data can be transmitted, a connection is established using a three-way-handshake [3]. The receiver has to acknowledge received packets, otherwise packets are retransmitted to guarantee delivery. A TCP packet consists of a header with a source / destination port pair, a sequence number (to maintain the right order and detect missing packets), flags and a data payload. The SYN flag indicates a new connection request from the sending party. The ACK flag is used for confirmation of packet reception. The FIN flag indicates the end of a connection. Figure 2 shows a typical TCP connection where party A connects to party B)

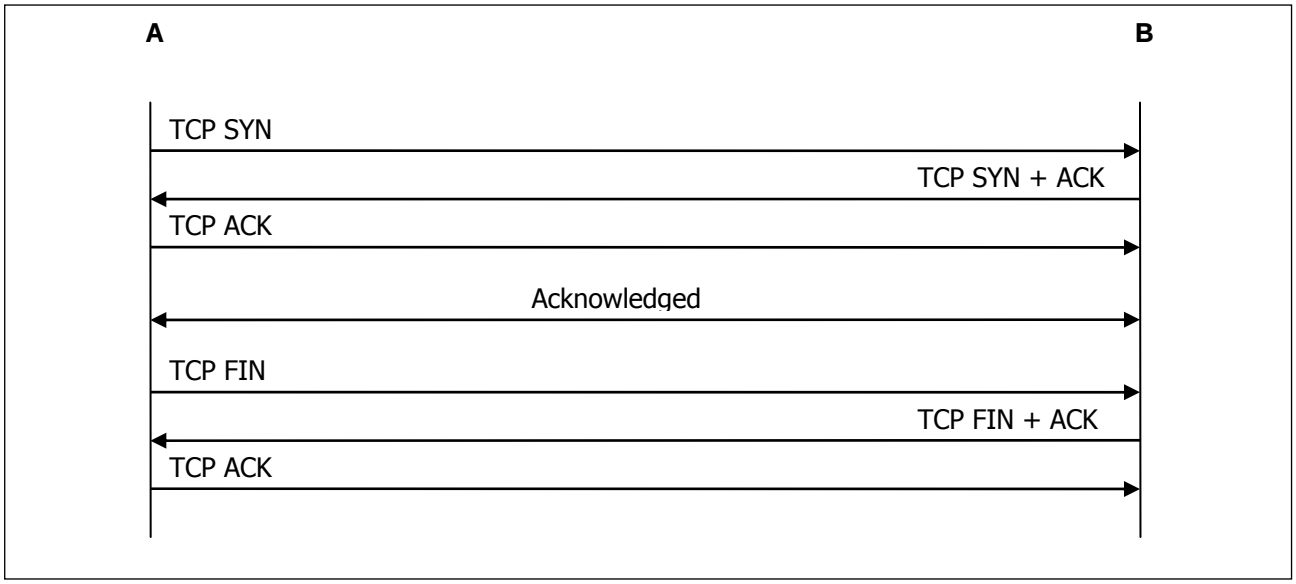

Figure 2: Typical packet sequence of a TCP connection

2.3 NAT and Firewall

2.3.1 NAT – Network Address Translation

Network address translation is the rewriting of source and/or destination of an IP packet when it passes through a router or firewall. NAT is most commonly used to connect a private network to the public internet. Since there is only a limited amount of public IP addresses, NAT allows for a multiple number of hosts to access the internet using only the single public IP address of the NAT device.
A NAT device changes the source/destination address and port of an IP packet (TCP or UDP), according to some rules. These rules depend on the actual implementation of the NAT device. Common rules are explained later.

Figure 3 illustrates a NAT process. The notion [A:a → X:x] represents a packet with source A:a and destination X:x. A NAT device has two network interfaces, one of which is assigned a public (e.g IP: M) and the other one a private IP address (e.g IP: L). If a host from the private network (with IP: A and Port: a) wants to send a packet to a public IP address (e.g. IP: X and Port: x), it sends it to the NAT device. The NAT device rewrites the source address of this packet to its own IP address M and a previously unused port m and forwards it to the specified destination address and port X:x. In addition, the NAT device creates a binding, which defines that whenever a packet is received on M:m it has to be forwarded to A:a (NAT changes the destination address and port to A:a). A binding expires if a TCP connection is finished (after successful TCP FIN handshake) or a defined connection idle time is reached.

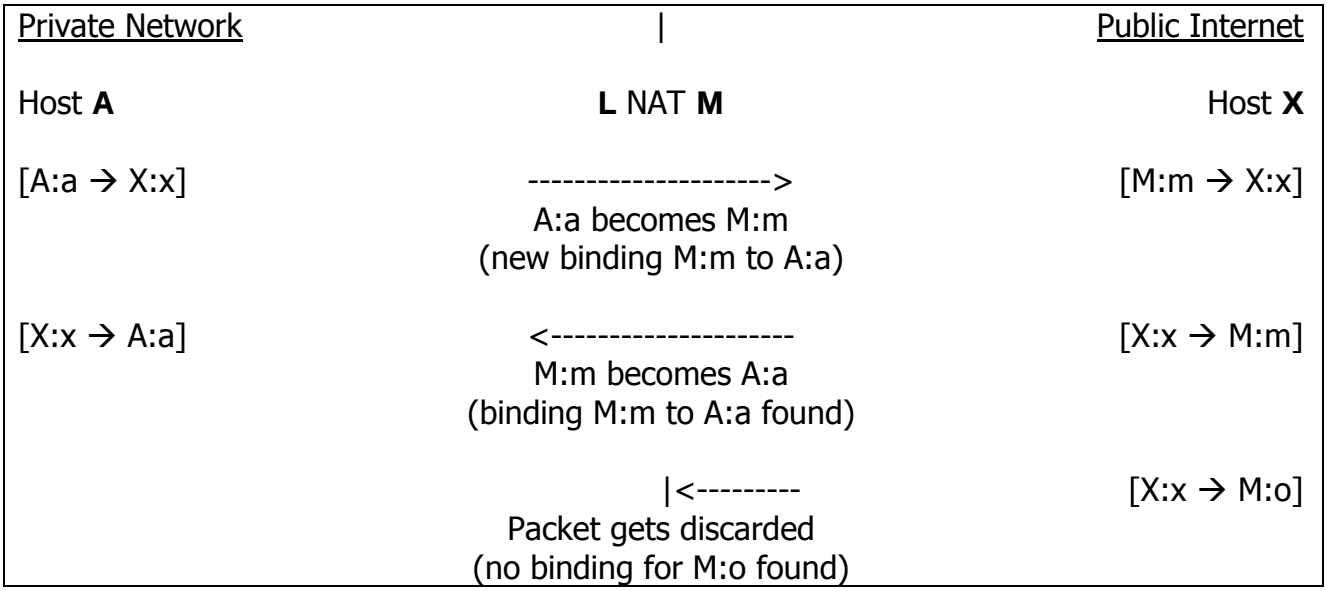

Figure 3: A typical NAT process: Host A (A:a) with private address connects to Host X:x with public address.

2.3.2 Firewall

A firewall is a device with several (at least two) network interfaces. A firewall controls the packet flow between all connected networks to improve security in the own network. A firewall is typically installed between a (corporate or home) network with of several workstations (having public IPs) and the public.
Generally, a firewall can either allow or deny a packet to pass through, depending on configurable rules. Since one can set arbitrary rules, the behaviour of a firewall depends

entirely on the rules. For instance, a firewall can be configured to block all incoming TCP connection requests (SYN flag set). A further scenario is that only packets with particular well-known ports are allowed to pass through the device: to allow only web browsing, HTTP traffic (identified by port 80) could be granted while FTP (port 21) could be denied.

Packet inspection
As mentioned before, a firewall has to decide, based on rules, if a packet must be forwarded or discarded. One method is to inspect the packet's source and destination IP and port. The rules define which combinations of source and destination addresses are allowed. It is common practice to allow only certain packets from well-known applications (identified by their UDP/TCP port number) to pass the firewall. The simplest way to do this, is to filter packets depending on their destination port number. The firewall administrator defines to block all ports by default and allows only special ports. Figure 4 shows a short list of well known ports.

Service	Port
FTP (File transfer)	21, 23
SSH (Secure Shell)	22
SMTP (Mailserver)	25
HTTP (Webserver)	80
POP3 (Mailserver)	110
HTTPS (Webserver with SSL)	443

Figure 4: Some well known services and their ports

Stateful packet inspection (SPI)
SPI is a method to extend stateless packet inspection explained above. Packets are not only inspected for source and destination addresses. In addition, the state of a connection is tracked by the firewall. A common case is that connections can only be established from inside (corporate or home network) to outside (public internet). So, packets from outside can pass to inside only, if an appropriate outgoing connection already exists. From this it follows that a stateful firewall can control the direction in which connection establishing is allowed. In contrast, a stateless firewall always has to allow both directions to pass through, otherwise two-way communication would not be possible.

2.4 NAT types with UDP traffic

There are different implementations for NAT devices handling UDP traffic. The difference lies in the filtering method for incoming packets from the public network. Cone behaviour on the one hand means that for any request from the same internal IP and port, the same external IP and port is used. Symmetric behaviour on the other hand means that for each request, which differs either in source or destination address and port, a new separate external port is mapped. See [11] for illustration.

Full Cone NAT
All requests from A:a are mapped to the same public address M:m. Once a mapping exists, each host from the public network can send packets to A:a using the mapped address M:m.
→ Input filtering for binding M:m: Source [AnyIP:anyPort]

Restricted Cone NAT

All requests from A:a are mapped to the same public address M:m. Once a mapping exists, only the hosts (same IP) which have previously received a packet from A:a, can send packets to A using the mapped address M:m.
→ Input filtering for binding M:m: Source [X:anyPort]
(where X is a list of allowed IPs)

Port Restricted Cone NAT
All requests from A:a are mapped to the same public address M:m. Once a mapping exists, only the hosts (same IP and port) which have received a packet from A:a before, can send packets to A:a using the mapped address M:m.
→ Input filtering for binding M:m: Source [X:x]
(where X:x is a list of pairs 'IP address:port number')

Symmetric NAT
Each request from A:a to a specific address X:x is mapped on its own public address M:m (where port number m is assigned for each request to a different IP or port). Once a mapping exists, only the host that has received a packet from A:a can send packets to A:a using M:m
→ Input filtering for binding M:m: Source [X:x]
(where X:x is a single pair 'IP address:port number')

2.5 NAT types with TCP traffic

There are different implementations for NAT devices handling TCP traffic. The differences lie mainly in the 'Endpoint Filtering' and 'Port Mapping' methods, which will be described in the following two sections.

2.5.1 Endpoint Filtering

If a NAT device receives a packet and a mapping exists on the receiving port, then the packet has to pass through the endpoint filter. The endpoint filter decides based on the origin of a packet, if it is to be forwarded or discarded. The following scenario helps to describe the different filtering methods:
First, A:a sends a packet to X:x. The NAT device creates a new external mapping M:m to A:a. Depending on the filter type, various hosts can now send packets to A:a using the address M:m.

Address and Port-independent Filtering
All packets from any host in the internet sent to M:m get forwarded to A:a.
→ Input filter rule: AnyIP:anyPort

Address-dependent Filtering
Packets from a particular host X sent to M:m get forwarded to A:a, independent of the source port number. All packets with a different source address are discarded.
→ Input filter rule: X:anyPort

Address and Port-dependent Filtering
Packets originating from host X with port number x sent to M:m get forwarded to A:a. All packets with a different source address or a different port are discarded.
→ Input filter rule: X:x

2.5.1 Port Mapping

A NAT chooses an external mapping for each TCP connection based on the source and destination IP and ports. Some NAT devices reuse existing mappings under some conditions while others allocate new mappings every time. For different internal addresses and/or source ports, always a new external port number is used.

Address and Port-independent Binding
For each new TCP connection originating from the same internal IP and source port, the device maps the same external port, regardless what the destination IP and destination port is.

Address and Port-dependent Binding
For each new TCP connection originating from the same internal source IP and source port, the device maps the same external port, but only if the destination IP and destination port are the same.
In most cases, where the destination IP or port differs, the port number assigned to the new mapping is obtained by increasing the last assigned port number by one. Where this is not possible, a new unused port has to be chosen.

Connection-dependent Binding
For each new TCP connection, the device maps a new external port, regardless what the source or destination IP and the port is.
In most cases, the port number assigned to the new mapping is obtained by increasing the last assigned port number by one. Where this is not possible, a new base port has to be chosen.

Remark: Some NAT devices select ports for new mappings randomly, regardless what the last assigned port number was. But this behaviour is seldom observed.

Port Prediction
Using some test connections and the knowledge of the port mapping type, a client (private net) and a server (internet) can predict the next external port, that is being assigned to a new connection. Explanation: A client connects from A:a to a server. Then the server tells the client which source port it sees (this corresponds to the external NAT port). Now one can predict, knowing the port mapping type, what the next external port will be, after the client closes its connection to the server and reconnects from the same address A:a to another host.

2.6 NAT Scenarios

In this section, we introduce different scenarios with two peers and either zero, one or two NAT devices. For NAT traversal, one has to analyze all possible configurations. These scenarios are later used and will be referenced to as 'Scenario x'.

Scenario I

Both clients A and B are directly connected to the internet with a public address. A and B can always connect to each other. As no NAT device is present, no network address translation is performed. This scenario is depicted in figure 5.

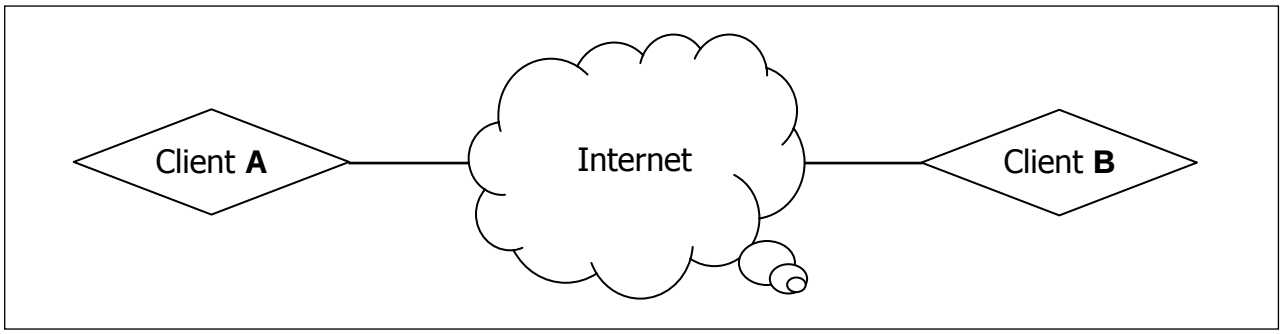

Figure 5: Scenario I: Two peers, no NAT device

Scenario II

Client B is directly connected to the internet with a public address. Client A is behind a NAT device. A can connect to B but not vice versa. This scenario is depicted in figure 6.

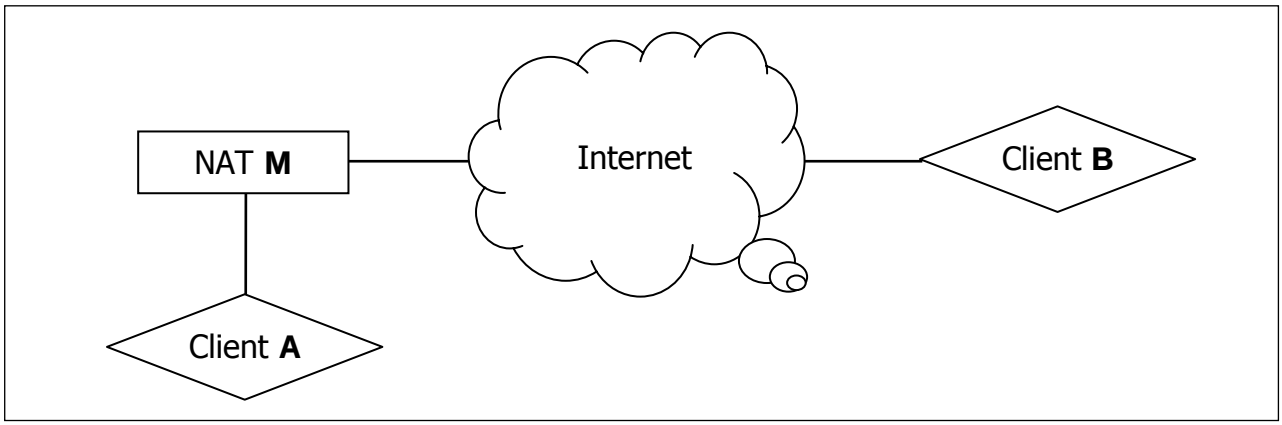

Figure 6: Scenario II: Two peers, one NAT device

Scenario III

Both clients are behind different NAT devices, which connect them to the internet. Neither A can connect to B nor B can connect to A. This scenario is depicted in figure 7.

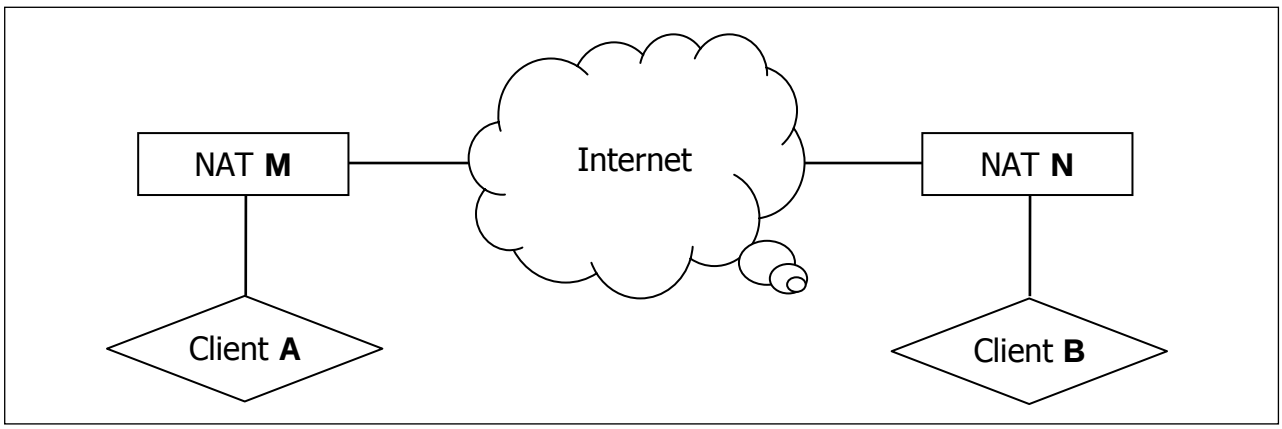

Figure 7: Scenario III: Two peers, two NAT devices

Scenario IV
Both clients are behind the same NAT device, which connects them to the internet. A and B can only connect to each other if they know their private addresses. This scenario is depicted in figure 8.

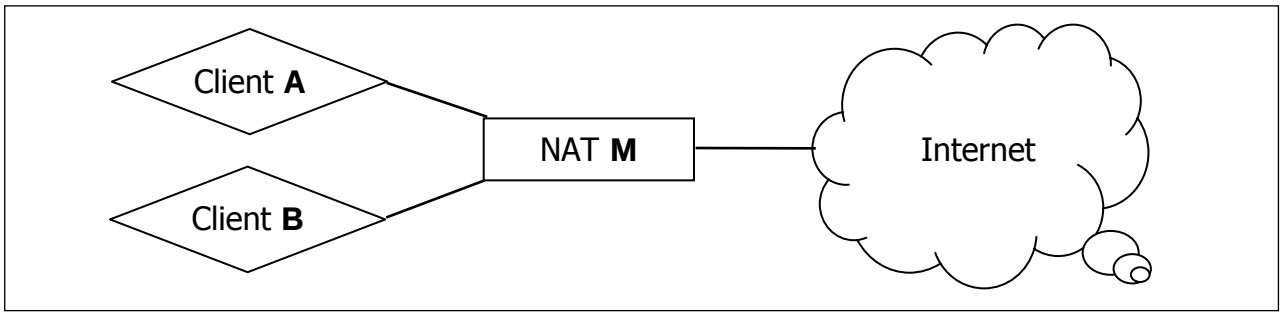

Figure 8: Scenario IV: Two peers behind the same NAT device

2.7 Observed NAT types on the market

Guha et al. analyzed about 100 commercial available NAT devices in [14]. They tested the NAT devices for 'Endpoint Filtering', 'Port Mapping' and 'TCP state tracking' behaviour. In addition, the market shares of each NAT device has been taken into account. As a consequence, they are able to estimate the percentages of NAT types present in the internet. These numbers, shown in Figures 9-11 below, are a good basis to assess in how many percentages of all cases a NAT traversal solution will work.

Endpoint Filtering

Type	Devices
Address and Port-independent	5.8%
Address-dependent	12.3%
Address and Port-dependent	81.9%

Figure 9: Estimated rate of endpoint filtering types in the internet, according to [14]

Port Mapping

Type	Devices
Address and Port-independent	70.1%
Address and Port-dependent	23.5%
Connection-dependent	3.9%
Others	2.5%

Figure 10: Estimated rate of port mapping types in the internet, according to [14]

TCP State Tracking

NAT devices track the TCP connection state whereby some devices filter uncommon packet sequences and eventually expire a binding.

Packet Sequence	Filtered
SYN-out SYN+ACK-in	0%
SYN-out SYN-in	13.6%
SYN-out ICMP-in SYN+ACK-in	6.9%
SYN-out ICMP-in SYN-in	22.4%
SYN-out RST-in SYN+ACK-in	25.6%
SYN-out RST-in SYN-in	28.1%

Figure 11: Percentage of devices that filter a certain packet sequence, according to [14]

3. Related Work

3.1 Existing NAT traversal Solutions for TCP

For the following NAT traversal approaches we consider the network scenarios described in section 2.6. Additionally, we assume that there exists at least one Server S with a publicly routable IP address, i.e. with a direct connection to the internet. Additionally, we assume that there exists a Proxy P which represents an out-of-bound channel that enables indirect communication from A to B.

The NAT traversal solutions in this chapter intend to solve the NAT traversal problem for the Scenario III depicted in figure 2.6.3. Scenario III is the most difficult case, since without help from a third party, neither A nor B will be capable to connect to the other party. Scenarios I and II are not subject of these solutions.

For Scenario I no special treatment is required. Clients A and B can communicate without restrictions using public addresses.

Scenario II is easier to solve, because client A always can connect to B using the standard TCP implementation. But client B cannot simply connect to client A, hence an out-of-bound channel between A and B is required. This channel can be either a common proxy or a network both clients can use to exchange messages through. So client B can inform client A of his connection need and then client A connects to B.

In case of Scenario IV, communication would be easy if client A and B know each others private IP addresses. If not, solutions for Scenario III can work, but the NAT device has to support hairpin translation. Hairpin translation means that, outgoing packets from client A destined for client B’s external address have to be redirected by the NAT device back to B’s internal address.

3.1.1 STUNT – Simple Traversal of UDP through NATs and TCP too

Guha et al present the STUNT protocol in [4]. STUNT helps to establish a direct TCP connection between two NATed hosts.
Assuming that client A wants to establish a TCP connection to client B, client A sends the intention to connect to client B using proxy P. Both STUNT clients now connect to a STUNT server S (which can be either the same server for both clients or two different servers) and perform a port prediction for their connection to each other. Using proxy P again, A tells B his predicted transport address M:m and vice versa N:n for B. Both A and B now send a TCP SYN packet to each other (A using destination N:n, B using destination M:m) with a TTL small enough that the packet does not reach the other party's NAT, but big enough to pass their own NAT and create the required port mapping. While sending those SYN packets, clients A and B have to listen on their own network interface with a raw socket for the outgoing TCP packet containing the SYN flag and read the sequence number of it. This sequence number must be sent to S. S then has to spoof a SYN+ACK packet (using a raw socket) with destination M:m and source N:n for A and vice versa for B (using the correct sequence numbers). Due to this spoofed SYN+ACK packets, A and B can now complete the three-way-handshake with a direct ACK to each other and the TCP connection is established. Figure 12 illustrates the TCP handshake using the STUNT protocol. To keep the diagram simple, the complete procedure is not depicted.

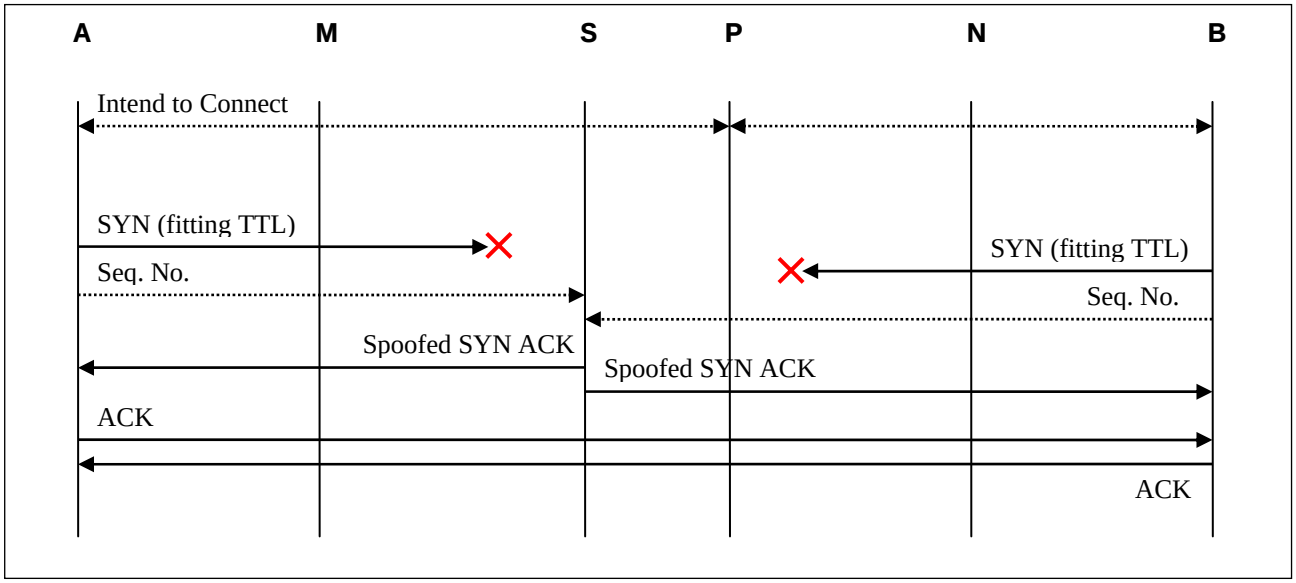

Figure 12: TCP handshake using the STUNT protocol

3.1.2 NATBlaster

Biggadike et al present the NATBlaster protocol in [5]. NATBlaster is similar to STUNT, but the server does not need to spoof a SYN+ACK packet.
NATBlaster helps to establish a direct TCP connection between two NATed hosts.
Assuming that client A wants to establish a TCP connection to client B, client A sends the intention to connect to client B using an out-of-bound channel. Both NATBlaster clients now connect to a NATBlaster server S (which can be either the same server for both or two different servers) and perform a port prediction for their connection to each other. Using again an out-of-bound channel, A tells B his predicted transport address M:m and vice versa N:n for B. Both A and B now send a TCP SYN packet to each other (A using destination N:n, B using destination M:m) with a TTL small enough that the packet does not reach the other party's NAT, but big enough to pass their own NAT device and create the required port mapping. While sending those SYN packets, clients A and B have to

listen on their own network interface with a raw socket for the outgoing TCP packet containing the SYN flag and read the sequence number of it. This sequence number has to be interchanged using an out-of-bound channel. A and B can now generate a SYN+ACK packet with the correct sequence number and send it to each other. As soon as the OS's TCP stack of A or B receives the SYN+ACK, it can complete the three-way-handshake with a direct ACK to the other party and the TCP connection is established. Figure 13 illustrates the TCP handshake using the NATBlaster protocol. To keep the diagram simple, not the complete procedure is depicted.

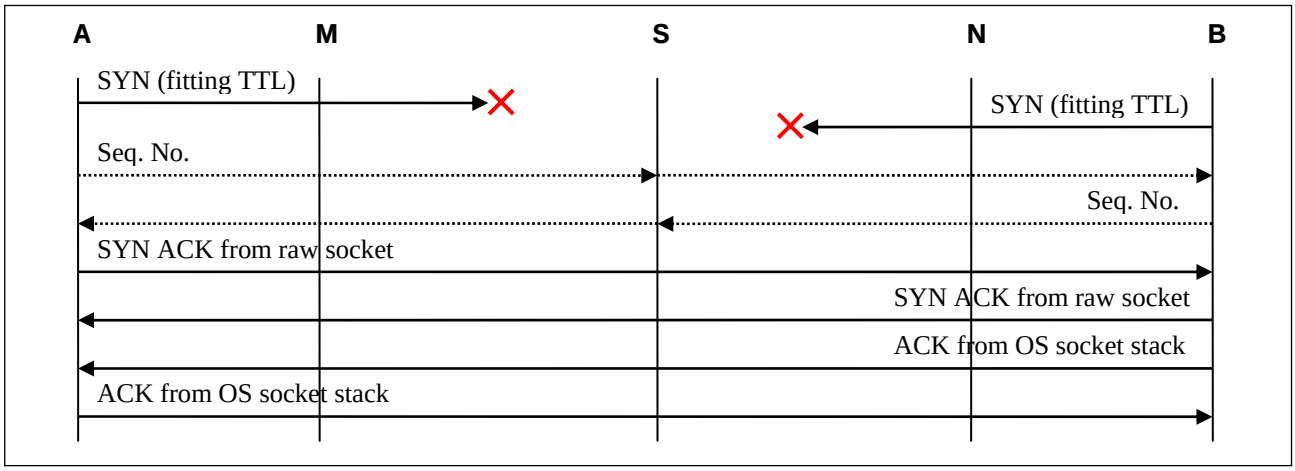

Figure 13: TCP handshake using the NATBlaster protocol

3.1.3 Peer-to-Peer NAT

Ford et al. present the P2PNAT protocol in [6]. P2PNAT helps to establish a direct TCP connection between two NATed hosts.

P2PNAT uses TCP "hole punching", i.e. outgoing TCP packets punch a hole in its own NAT. Client A wants to establish a TCP connection to B. Suppose that clients A and B already have a connection with a well-known rendezvous server S. Then the following steps are performed:

1. Client A uses its active TCP session with S to ask S for help connecting to B.
2. S replies to A with B's public and private TCP endpoints, and at the same time sends A's public and private endpoints to B.
3. From *the same local TCP ports* that A and B used to register with S, A and B each asynchronously make outgoing connection attempts to the other's public and private endpoints as reported by S, while simultaneously listening for incoming connections on their respective local TCP ports.
4. A and B wait for outgoing connection attempts to succeed, and/or for incoming connections to appear. If one of the outgoing connection attempts fails due to a network error such as "connection reset" or "host unreachable", the host simply re-tries that connection attempt after a short delay (e.g., one second), up to an application-defined maximum timeout period.
5. When a TCP connection is made, the hosts authenticate each other to verify that they connected to the intended host. If authentication fails, the clients close that connection and continue waiting for others to succeed. The clients use the first successfully authenticated TCP stream resulting from this process.

Figure 14 shows a possible scenario for these connection attempts.

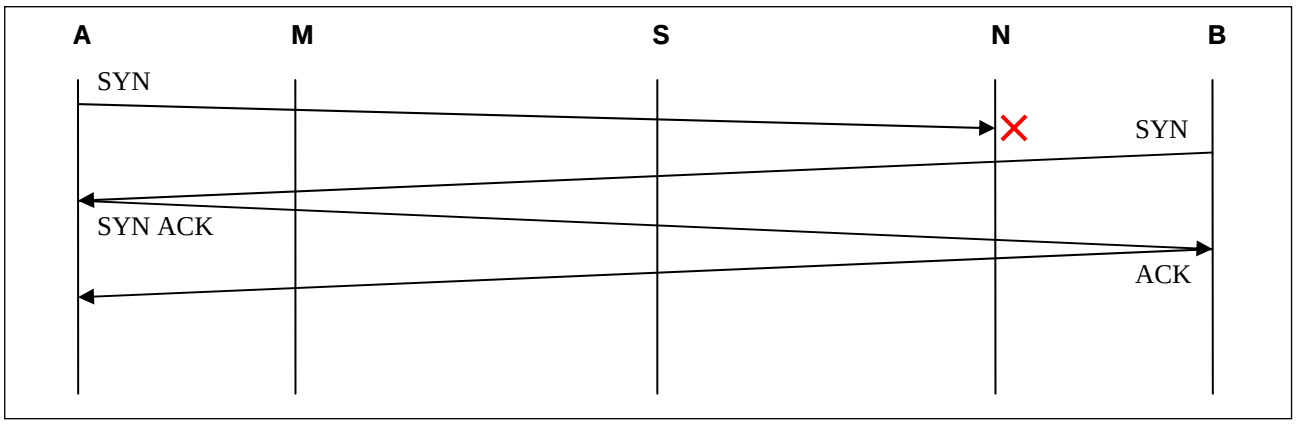

Figure 14: TCP handshake using the P2PNAT protocol

3.1.4 TURN – Traversal Using Relay NAT

J. Rosenberg presents the TURN protocol in [7]. TURN uses a relay server S, which provides transport addresses for clients using NAT (e.g. A and B). S is required to have a public IP address and must not be firewalled, so that all hosts can connect to it (whether they are using NAT or not). S is a kind of proxy which forwards traffic. If A wants to be reachable from the public internet, then A:a sends a bind request to S. As a consequence, S binds a local port (for example S:s) for A and listens for incoming connections. As soon as a host B connects to S:s, the TURN server S accepts the connection and forwards all traffic from S:s to A:a and all traffic from A:a to S:s. Once the binding S:s receives a connection, S:s becomes locked and no other host can connect to it. A new binding request is needed for further connections.

3.1.5 uPNP and MIDCOM

Details about uPNP can be found in the uPNP Forum [8]. For more information about MIDCOM see Middlebox communication architecture and framework [9].
These two approaches use another mechanism to traverse NAT devices. uPNP and MIDCOM are protocols to configure NAT devices, which have to support these protocols. A client A can configure his NAT device M, so that M provides a public transport address for A:a, say M:m. Whenever traffic arrives at M:m it gets automatically forwarded to A:a. → Static port forwarding. The result is that each NATed host is always reachable for an intended service. This is a really good and simple approach, but a minority of NAT devices support these protocols. No more servers are needed to traverse NAT devices and traversing becomes really reliable, but just a few manufacturer deliver NAT devices with uPNP or MIDCOM support. This is because uPNP, MIDCOM and similar mechanisms break the security function of NATs and firewalls.

3.2 Firewall and NAT traversal in other Applications

This chapter briefly reviews how other peer-to-peer applications solve the problem of firewall and NAT traversal.

3.2.1 Skype

Skype does not explain their techniques in detail, but they provide general information about firewall and NAT handling in [12].
First of all, Skype prefers UDP traffic because of smaller latencies and overhead. Skype makes no use of servers except a login server. For server-similar tasks a small amount of user nodes is elected (called supernodes).

Establishing connections in Skype works as follows:
If a user A wants to connect to user B, then first user A tries a direct connection to B. However, if B is protected by a firewall or NAT, then A informs the so called 'global index' (decentralized user list maintained by supernodes), which instructs B to try to connect direct to A. If none of this connection attempts succeeded due to firewalls protecting A and B, then another non-firewalled Skype user is chosen to act as relay for this connection. In case of two NATed hosts, Skype tries a method called UDP "hole punching" and probably eludes to a relay host, if direct UDP connection does not work.

3.2.2 eDonkey

Although eDonkey is called a peer-to-peer filesharing system, eDonkey is not a pure peer-to-peer system, since a lot of servers are invoked to coordinate the peers. When a client registers at a server, the client is assigned an ID. This ID is created by the server and depends on the client's IP address, if the host is reachable from the internet. In this case the ID is a HighID, which means greater than 16777217. If the client is firewalled or NATed, then the ID is a LowID and has nothing to do with the client's IP address. Clients with a HighID can communicate with all other clients, but a client with a LowID can only exchange data with HighID clients. By this means the problem of two NATed clients wanting to connect to each other never occurs. Besides, eDonkey servers prompt LowID users to connect to HighID users, so the scenario II is also no problem.
eDonkey solves the problem of NAT traversal by avoiding NAT traversal. The network only provides good performance if there are enough HighID users.

3.3 Classification of existing Solutions

As described in section 2.5, there are three different types of approaches for NAT traversal. We classify the NAT traversal solutions in three classes as follows:
In the first type (e.g. STUNT, NATBlaster, P2PNAT), a server procures a direct TCP connection between two hosts behind a NAT. We subsequently refer to these approaches as "Agent-type", because the server acts as an agent.
The second type (e.g. TURN) describes a relay server, that offers a public IP address and forwards all traffic for a client. We will refer to this type as "Relay-type".
Finally, there are approaches which require configurable NAT devices (e.g. uPNP, MIDCOM). This type subsequently will be called "Preconfiguration-type".
Each type and approach has its strengths and weaknesses, which we will discuss in the following paragraphs.

3.3.1 Agent-Type

This type is widespread and works with most NAT devices. Since a server is needed, some modifications would be necessary for this type to work in a pure peer-to-peer network. There are a few restrictions, which NAT devices have to obey, for each implementation to work. For example, P2PNAT works only with NAT devices that have cone behavior and accept uncommon packet sequences, such as outgoing SYN followed by incoming SYN. STUNT and NATBlaster do not work if a NAT device changes sequence numbers in TCP packets (some NAT devices add an offset to outgoing sequence numbers and subtract it from incoming sequence numbers). Another problem is that some routers and firewalls filter packets with spoofed IP addresses, which troubles the STUNT approach. Most ISPs' router check the source address of their clients packets to avoid spoofed IP packets in the network (also known as ingress and egress filters).

NATBlaster and STUNT make use of raw sockets and have to set a fitting TTL. But the use of raw sockets and the need to set the TTL in TCP packets can cause troubles or is not possible at all on some operating systems. For example, sending TCP data over a raw socket has been disabled under Windows XP SP2 (see [10]). Further such operations need superuser privileges on Windows and Unix.

3.3.2 Relay-Type

The big advantage of this class is that it will enable communication between hosts behind a NAT device in probably almost all cases. No special requirements for NAT devices, operating systems or user privileges are needed. TURN uses the NAT how it was designed to be used (client <--> server model). Moreover, an arbitrary client can communicate, using a standard TCP socket implementation, with a host behind a NAT device only by knowing his transport address.
The drawback is that each connection between two hosts behind a NAT uses bandwidth from the server. Since a server is needed, some modifications would be necessary for this type to work as a fully distributed system. Peers have to perform server tasks and the relay traffic has to be well distributed over all peers.

3.3.3 Preconfiguration-Type

This would be the most sophisticated solution, because there is no need for servers. Each host behind a NAT appears as a host with a public address. The problem of this solution is, that only a few devices support it. Further it only allows configuring the NAT devices of its own network, which makes it unusable when there is for example another NAT device in the ISP's network. So this solution drops out early.

4. Concept

4.1 Evaluation of existing solutions in the context of Celeste

As solution for Celeste one could imagine the Agent or Relay type. But since Celeste is a completely distributed peer-to-peer system, no dedicated server for NAT traversal purposes would be possible. For this reason, peers with a public address must necessarilly act as servers.

Considering network scenario III and STUNT or NATBlaster, one can traverse most of the NAT devices available in the market.
But if a peer should act as an agent-type server, there are some requirements the peer's computer system has to meet: Peers have to learn a NAT device's port mapping behaviour. The server uses two network interfaces, which most of the peers probably won't have. Also the operating system must allow sending TCP packets over a raw socket. Furthermore, the user running Celeste must have superuser privileges to access raw sockets. Hence, NAT traversal would work only for a subset of all peers behind a NAT device. Additionally, only a fraction of the peers could play the role of a server.
P2PNAT in contrast to STUNT or NATBlaster can traverse only a fraction of available NAT devices. This is because no port prediction is proposed. But the mechanism for TCP handshake is much easier and there are no special needs to the peer's computer system. This solution works with the standard TCP stack implementation of all operating systems (this means that no raw sockets or IP address spoofing are used).
→ An agent-type approach turns out to be not optimal, but TCP "hole punching" could be used to cover some special cases (e.g. where two NAT devices, each with independent port mapping, are present).

In the relay-type NAT traversal, each peer could act as a relay server, because there is no special requirement for the operating system of a peer (a standard TCP socket suffices for implementation). Assuming that most of the peers are not behind a NAT device, the relay traffic could easily be shared over the whole peer-to-peer network. This avoids that a single peer has to sacrifice too much of his bandwidth to act as relay server. In Celeste, most of the traffic does not make his way directly to the destination, as a packet is forwarded via other peers to its final destination. One could imagine the relay server to be a neighbour node in the peer-to-peer network.

After comparing the different methods, the best solution for NAT traversal in Celeste seems to be an approach similar to TURN, because TURN has no special requirements and provides connectivity for most of the peers. STUNT and NATBlaster are not realizable on many systems and not easy implemented in Java, because Java supports no raw sockets nor IP address spoofing by default.
In addition, a kind of P2PNAT (TCP "hole punching") can be used to cover a common case, where both NAT devices use 'independent port mapping'. Further port prediction can be introduced to enhance connectivity with TCP "hole punching". So, a direct TCP connection is used whenever it is possible with simple TCP "hole punching" or else TCP relaying is used. This decreases the bandwidth used in the peer-to-peer network to relay traffic.

Figure 15 summarizes the comparison.

Solution	Advantage(s)	Disadvantage(s)
Agent-Type		
- STUNT	· Good connectivity for many different NAT devices	· Superuser privileges required · Raw sockets required · IP address spoofing
- NATBlaster	· Good connectivity for many different NAT devices	· Superuser privileges required · Raw sockets required
- P2PNAT	· Simple approach · No superuser privileges required	· Average connectivity · Can only traverse a subset of available NAT devices, since no port prediction is suggested
Relay-type		
- TURN	· Provides connectivity in almost all cases · No superuser privileges required	·Consumes bandwidth from the peer which acts as server
Preconfiguration Type		
- uPNP	· Would provide connectivity in almost all cases	· Only a minority of NAT devices is supported.
- MIDCOM	· Would provide connectivity in almost all cases	· Only a minority of NAT devices is supported.

Figure 15: Evaluation summary of existing solutions for NAT device traversal

4.2 General Concept

4.2.1 NAT Traversal

My solution for NAT traversal for Peer-to-Peer Storage Nodes looks similar to TURN combined with TCP "hole punching". The main difference is that the dedicated relay servers are replaced by the peers with a public address (later referenced as servers).
The servers provide a 'public address:port' tuple for each NATed or firewalled peer, where "firewalled" means not reachable with a TCP connection from the internet. Each NATed host publishes itself as the tuple assigned by a server.
The following sections give an overview how my solution handles the scenarios described in section 2.6.

Scenario I

For a connection between A and B no traversal is needed. A direct TCP connection is always used. Both A and B indicate at the beginning of the connection, that they are neither a relay nor NATed. After the initialization phase, they are ready to transmit data.

Scenario II

If A wants to connect to B no traversal is needed. A indicates at the beginning of the connection that A is NATed. B indicates that it is not a relay, which means that A has reached the desired peer and it is ready to receive data.
If B wants to connect to A, it is not possible with a direct connection. But B knows the address of the relay server from A (e.g server S). Hence, B connects to S and tells S, that B is not behind a NAT. Thereon S indicates that it is a relay server and instructs B to wait for an incoming connection from A. As next, S instructs A to connect to B. The result is the desired connection from B to A.

Scenario III

For this scenario the following assumptions apply:

1. Client A is connected to a server S. S sees packets from A with source address M:m.
2. If client B connects to relay S, then S sees packets from B with source address N:n.

If B wants to connect to A, then NAT traversal is needed. As in scenario II, B knows the address of the relay server from A (e.g. server S). Hence, B connects to S and tells S, that B is behind a NAT. Thereon S indicates that it is a relay server and instructs B to disconnect from S and to send a packet with SYN-flag to A using M:m or M:m+1 as destination (depending on A's NAT port mapping type). B must use the same source port, which is used to connect to S (e.g. port b). This SYN "punches a hole" in NAT N, which enables A to connect to B. After B has sent a packet with SYN-flag to A, B has to listen on port b for an incoming connection. In the meantime, S instructs A to disconnect from S and to connect to B using N:n or N:n+1 as destination (depending on B's NAT port mapping type). A must use the same source port, it used to connect to S (e.g. port a). If NAT device M does not filter 'SYN-out SYN-in', then NAT N receives a packet with SYN-flag on port n or n+1 from M:m or M:m+1 and forwards it to B:b → B accepts the connection and completes TCP handshake. Figure 16 shows the TCP handshake using this simple TCP "hole punching". The estimated success rate for this "hole punching" connection can be calculated based on the statistics in section 2.7. 97.5% of all NAT devices have a predictable port mapping behaviour. 13.6% of the devices filter a 'SYN-out- SYN-in' and 22.4% a 'SYN-out ICMP-in SYN-in' sequence (some NAT devices generate an ICMP error when receiving a TCP SYN on a unmapped port). So, this result in a percentage connectivity between 84.24% and 75.66%. TCP relay is used where this connection fails.

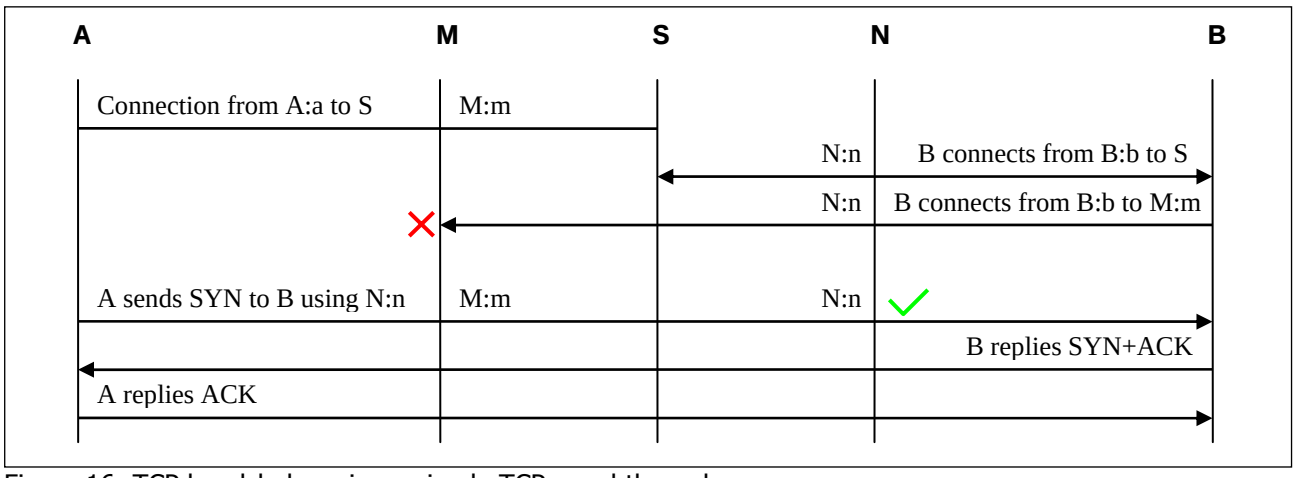

Figure 16: TCP handshake using a simple TCP punchthrough

As described, this solution only works, if both NAT devices use 'Address and Port independent Port Mapping', because no port prediction is used. Further NAT N must allow the uncommon packet sequence 'Outgoing SYN, incoming SYN' (a minority of NAT devices filters this sequence). After B sent a packet with SYN-flag, B waits for a connection over a defined time. If no connection arrives, then B reconnects to S and demands a relayed connection, which guarantees an indirect connection to A.

Scenario IV
If B wants to connect to A, then NAT traversal could be needed. As in scenario II, B knows the address of the relay server from A (e.g. server S). Hence, B connects to S and tells S, that B is behind a NAT. S recognizes that both peers A and B have the same IP and informs B of A's private address. B can connect to the private address of A. If B fails to connect to A's private address, B can request a relayed connection to A, which guarantees an indirect connection to A.

4.2.2 Firewall traversal

Firewall traversal is a more basic problem than NAT traversal, because sometimes firewalls are not meant to be traversed. Each network administrator has its own security policies, according to which he configures his firewall. The result can be very restricted firewall rules, which trouble peer-to-peer networks a lot. For instance, the firewall is configured to block all incoming connections and allow outgoing connections on port number 80 (HTTP), 25/110 (SMTP and POP3 for eMail) and perhaps a small number of other well known application port numbers. If Celeste port numbers are blocked by the firewall, there is no possibility for a Celeste peer to connect to other clients or receive connections from other clients. To solve this problem, we look at two cases separately.

The first case is port restriction, where only a few port numbers of well known applications are not blocked. One can imagine two approaches to traverse most of port restricted firewalls:

1. Each Celeste peer listens to a port assigned to another well known application (e.g. port number 80, 443 or 25). We can assume that most of the firewalls are configured to accept connections on these ports. As a consequence, this approach would traverse most of the firewalls.
2. Each Celeste peer, behind a port restricted firewall, communicates through a relay server, which listens on a port number that is assigned to a well known application, with the other peers.

Though these two solutions would work, there are some drawbacks. A peer can only listen to well known port numbers, if the user, running Celeste, has super user privileges. Further Celeste could come into conflict with a webserver, mailserver or a similar applications (using its specified port) running on this machine. If a peer cannot listen to well known ports, he probably cannot receive connections from peers behind a port restricted firewall. Additionally, such a "trick", as using another application's port number, compromises the network administrator's security policy.
The most sophisticated solution would be to enable the Celeste ports at the firewall. So, the Celeste user would have to inform the network administrator of his wish to use Celeste.

The second case is a stateful firewall that only allows outgoing connections. Imagine the NAT devices described in scenarios I-IV to be such firewalls. To solve this case, we can simply use the same methods as for NAT devices. Since we have the same problem, that only outgoing connections are possible. Besides detecting a NAT device, it is important to check for a stateful firewall too.

4.3 Detailed Procedures

In this chapter, a detailed description of the procedures, necessary for this firewall and NAT traversal approach, is given. The program that runs on every Celeste peer is called "NAT and Firewall Traversal Client" or short NFTClient. Additionally, all peers with a public address run a program called "NAT and Firewall Traversal Server" or short NFTServer.

4.3.1 Bootstrap

At start-up, Celeste has to provide a list of IP address of a Celeste node with a public address (each public Celeste node also runs a NFTServer). Those must be the nodes in the Celeste network with the nearest NodeID, compared to the own NodeID, sorted from the nearest to the farthest NodeID. Celeste should provide about ten Nodes, to have a sufficient big choice, but not having to test too much hosts. The first address (if not reachable the next address is taken) is used to test if the peer is behind a NAT/Firewall or not. To perform this test, a *discoverRequest* is sent to the NFTServer over TCP. The NFTServer must answer with a *discoverAnswer*, which contains the source IP and source port of the *discoverRequest* packet. This way, the NFTClient can compare the IP address and source port received in the *discoverAnswer* to its own. If it is the same, the NFTClient is not behind a NAT, else the NFTClient is behind a NAT.

1. In the first case the NFTClient is not behind a NAT and so, the NFTClient has to check for a firewall device. Therefore the NFTClient sends a *discoverRequest* containing a port number. The server has to connect to this port number on the IP address the request came from. The NFTClient listens on this port for a connection from the server. If the NFTClient received a connection from the server, the client can assume not to be behind a stateful firewall. If a NFTClient is not behind a NAT nor firewall, then it has to start a NFTServer. And Celeste uses the NFTClient to obtain sockets to other Celeste peers with NFTClient. If no connection arrives from the server, the NFTClient can assume to be behind a stateful firewall and use the same procedures, as if the client was behind a NAT, described in 3.

2. In the second case the NFTClient is behind a NAT or firewall and a subsequent *discoverRequest* to he next server from the list is made. This subsequent request is used to determine the port mapping of the NAT device. The port number received in the *discoverAnswer* from the second server is compared to the one from the first server. If it is the same, the NAT device uses "address-independent port mapping". If the port is one higher, the NATdevice uses "address and port-dependent" or "connection-dependent port mapping". Which of the two does not matter for out approach. This knowledge is used later to predict ports for NAT traversal. If the port is neither the same nor one higher, then the port mapping is declared as non predictable.

3. When the NFTClient is behind a NAT device, the NFTClient does not start a NFTServer. Instead, it has to choose a NFTServer that provides it with a public address and relay capability. The NFTClient evaluates the connection performance to all the peers on its node list. Possible metrics for performance are:

    1. How many other NFTClients use this NFTServer (more clients may but need not indicate worse performance)
    2. Distance between own NodeID and the NFTServer's NodeID
    3. Latency (the shorter the packet round trip time is, the better the estimated performance)
    4. Available Bandwidth (no simple method to measure)
    5. Number of network hops between client and server (Either count hops or measure latency)

Metric 2 will result in a good load balancing of relay traffic over the whole Celeste network, if we assume, that peers behind NAT are evenly distributed over NodeIDs. To avoid choosing a peer that already serves several other peers, metric 1 should be combined with 2. Additionally, the latency has to be considered, to avoid choosing a peer at the other end of the world. This would result in slow connections. Metric 4 and 5 are more difficult to measure and are not taken into account.
After a fitting NFTServer has been chosen, the NFTClient sends a *relayRequest* to it. This request also contains the type of port mapping the client's NAT device uses. The server has to accept the request, if the server has enough resources (e.g. if the maximum amount of relay connections is not reached, this maximum amount as to be determined in practice). The NFTServer answers with a *relayAnswer* containing the public address (TCP port), on which the NFTServer accepts connections for the NFTClient, and another TCP port, which the client must use to connect to the server.

Now the Celeste peer can announce itself in the Celeste network to be reachable with the received public address.

4.3.2 NFTClient Behaviour

The NFTClient provides functions to accept incoming connections and to establish outgoing connections. These functions have different behaviour depending on the presence of a NAT device. To accept an incoming connection, acceptSocket() is called. To request an outgoing connection, requestSocket(DestIP, DestPort) is called.
Further there is a function to discover the presence of a NAT or firewall device and one to measure the performance/load of a NFTServer.

First, the behaviour for a NFTClient not behind a NAT is described:

requestSocket
The NFTClient generates a new socket to the DestIP:DestPort tuple and sends a message: "I am not behind a NAT".
Scenario I: If the destination is another NFTClient, then it answers: "I am not behind a NAT" (Scenario I). So, the connection is established and the 'requestSocket' returns the socket to celeste.

Scenario II: If the destination is a NFTRelay, then it answers: "I am a NFTRelay for [IP Address:Port]" (Scenario II). The NFTClient has to close this socket and listens on the same port, it used to connect to the server, for an incoming connection from [IP Address:Port]. In the meantime the NFTRelay instructs its client (the one that is NATed) to connect to the NFTClient. When the connection arrives, the 'requestSocket' returns the socket to Celeste.

acceptSocket
The NFTClient waits on a defined serverSocket for an incoming connection (Java: serverSocket.accept()). When a new connection has arrived, the NFTClient waits for a message. This message can either be "I am not behind a NAT" (Scenario I) or "I am behind a NAT" (Scenario II). No matter which case happens, the NFTClient answers with "I am not behind a NAT" and the 'acceptSocket' returns the socket to Celeste.

Second, the behaviour for a NFTClient behind a NAT is described:

requestSocket
The NFTClient generates a new socket to the DestIP:DestPort tuple and sends a message: "I am behind a NAT". This message also contains the type of port mapping of the sender's NAT. So, the server can predict the next port for the subsequent connection).

Scenario II: If the destination is another NFTClient, then it answers: "I am not behind a NAT" (Scenario II). So, the connection is established and the 'requestSocket' returns the socket to celeste.
Scenario III: If the destination is a NFTRelay, then it answers: "I am a NFTRelay for [IP Address:Port]". The NFTClient has to close this socket and it has to connect to the received [IP Address:Port] from the same port it used to connect to the server. This will fail, but punch a hole in its own NAT device. After failure, the NFTClient has to listen on the same port for an incoming connection. In the meantime the NFTRelay instructs its client (the other NATed party) to connect to the NFTClient (again, the other NATed party has to use the same port, it used to connect to the NFTRelay).

When a connection arrives at the NFTClient from the other NATed party, then the 'requestSocket' returns the socket to Celeste. Figure 16 illustrates the TCP handshake for this case.
If this connection fails (the NFTClient waits for 10 seconds for an incoming connection), then the NFTClient sends a "Relayed Connection Request" to the NFTRelay and the 'requestSocket' returns the socket to Celeste.

acceptSocket
To accept an incoming connection, the NFTClient establishes a connection to its NFTRelay. This signalizes the NFTRelay to accept an incoming connection from outside. As soon as the NFTRelay received a connection from outside, it sends a message to the NFTClient: "Connect to (NATed) [IP Address:Port]" (Scenario II or III). Whereon the NFTClient closes its connection to the NFTRelay and connects from the same port, it used to connect to the server, to [IP Address:Port]. The NFTClient waits for 1 second. If the connection is not accepted, it retries the connection and waits for another 5 seconds. In the meantime the server instructs the other party to wait for an incoming connection and eventually punch a hole in its NAT. When this connection succeeds, then the 'acceptSocket' returns the socket

to Celeste. Figure 16 illustrates the TCP handshake for this case. When this connection fails, the NFTClient reconnects to the server and waits for a new message. Direct connection through NAT failed in this case. The other party has to request a relayed connection.
If the NFTClient receives the message "Relayed Connection", then the 'acceptSocket' returns the socket to Celeste. In this case the NFTRelay has received a "Relayed Connection Request" and creates a forwarding to the NFTClient.

4.3.3 NFTServer Behaviour

The NFTServer answers any kind of NFTClient requests. It can start relay services for NFTClients and those relay services can create TCP connection forwards. The NFTServer is a thread that a NFTClient starts if it is reachable from the internet (not behind a NAT nor firewalled).

NFTServer Thread
The NFTServer thread waits for incoming NFTClient requests. The following requests can occur:

discoverRequest
The NFTServer answers this request with a *discoverAnswer*, which contains the IP address and port the request came from.

echoRequest
The NFTServer immediately answers this request with a *echoAnswer*, which contains the number from the *echoRequest* and the number of relay threads that exist on this NFTServer.

relayRequest
The NFTServer answers this request with a *relayAnswer*, which contains a public-port number and a client-port number, if the server is not full and the *relayRequest* gets accepted. If the maximum amount of relay threads is reached, the *relayRequest* gets denied and the server answers with an empty *relayAnswer*.
When the relayRequest is accepted, the server creates a new NFTRelay Thread, which is bound to the peer that requested the relay. The NFT listens on the public-port for incoming connections and communicates with the NFTClient through the client-port.

NFTRelay Thread
To describe the behaviour of this thread, the following arrangements are made:
- The NFTClient who requested this NFTRelay is referred to as "Owner".
- The NFTClient who requests a connection to the Owner is referred to as "PeerX".

The NFTRelay listens on two ports, the client-port for connections to the Owner and the public-port for connections to any PeerX.
The NFTRelay thread does the following steps:
1. Wait for a connection on the client-socket (the Owner must connect to this socket to receive a connection).
2. Accept a connection on the public-socket (any PeerX, that wants a connection to the Owner, can connect to this socket).

3. Wait for a message from PeerX (timeout is 5 seconds, after that go to 2.).
   If the message is "I am not behind a NAT" go to step 4.
   If the message is "I am behind a NAT" go to step 4.
   If the message is "Request Relayed Connection" go to step 5.
4. Send a message to PeerX: "I am a NFTServer for [IP Address:Port]" (where [IP Address:Port] is the tuple from the Owner Connection with port prediction applied)
   Send a message to the Owner: "Connect to [IP Address:Port]" (where [IP Address:Port] is the tuple from PeerX with port prediction applied)
   Goto step 1.
5. Send a message to the Owner: "Relayed Connection".
   Create a Forward from the Owner to PeerX and vice versa.
   Goto step 1.

4.4 Summary

This short section summarizes the NFT solutions for different scenarios. Figure 17 shows the connectivity for different scenarios. A description of the cases can be found below the figure.

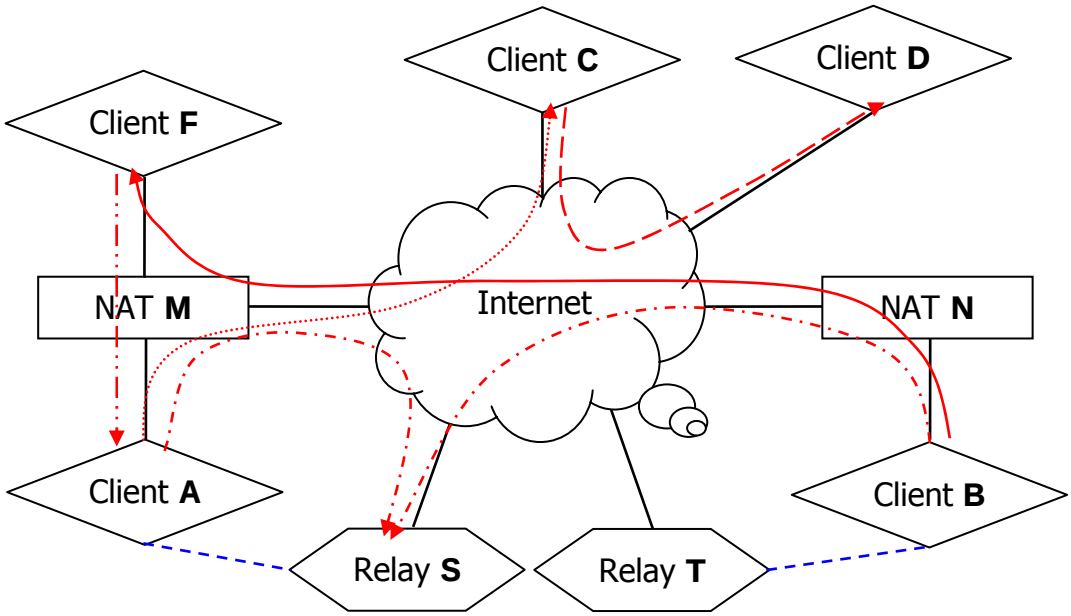

Figure 17: Connectivity for different Scenarios.

Scenario I: Client C and D use a simple direct connection. → – – –

Scenario II: Client A can directly connect to C and C gets a connection to A with the help of client A's relay server S. → ·····

Scenario III: If possible, a relay server procures a direct connection (such as between B and F). → ———
If a direct connection is not possible, then traffic is relayed by a relay server (such as between A and B). → –·–·–

Scenario IV: If a relay server recognizes that two clients are behind the same NAT device (i.e. if they have the same IP address), then the relay server instructs a client to try to connect to the other client's private address. → –··–
If this fails, the client can request a relayed connection.

5. Implementation

For testing purposes, I implemented a simple version of this NAT and firewall traversal approach in Java (JDK version 1.5). The implementation does not suffice the full specification described in the concept, but serves to test the general functionality of this NAT traversal solution (especially Scenario III).

Implemented features

My implementation supports the following features from the concept:

- NAT device and TCP port mapping discover.
- Server selection using the latency and relay load as metric.
- NFTRelay thread is implemented and supports TCP "hole punching" NAT traversal and relayed connections.
- NFTForward is implemented and supports duplex communication.
- The NFTClient supports outgoing connections to NATed and public hosts, and relayed connections. Incoming connections are only supported for NATed clients from NATed clients (since this is the interesting case to test).

Source Code

The source code consists of several files for the client and the server. Figure 18 shows a table of these files and a summary of their content.

Server Files	**Description**
NFTServer.java	Interface to start and stop the NFTServerThread. Includes a main procedure for execution.
NFTServerThread.java	Actual NFTServer funcionality is implemented in this file.
TCPDiscover.java	Additional thread that answers discoverRequests for TCP. This thread is started by the NFTServerThread and is part of the NFTServer.
NFTRelay.java	NFTRelay thread funcionality is implemented in this file.
NFTForward.java	NFTForward thread funcionality is implemented in this file.
NFTPacket.java	Used by NFTClient, NFTServer and NFTRelay for handling communication messages between client and server.
IPAddress.java	Used by all other classes. This class handles IP addresses and the conversion of them into different types.
Client Files	
NFTClient.java	NFTClient funcionality is implemented in this file.
NFTPacket.java	Used by NFTClient, NFTServer and NFTRelay for handling communication messages between client and server.
IPAddress.java	Used by all other classes. This class handles IP addresses and the conversion of them into different types.

Figure 18: Table of NFT files with short description

5.1 UML Class Diagram

Figure 19 illustrates the basic class structure of my NAT and Firewall Traversal implementation in an UML-like class diagram.

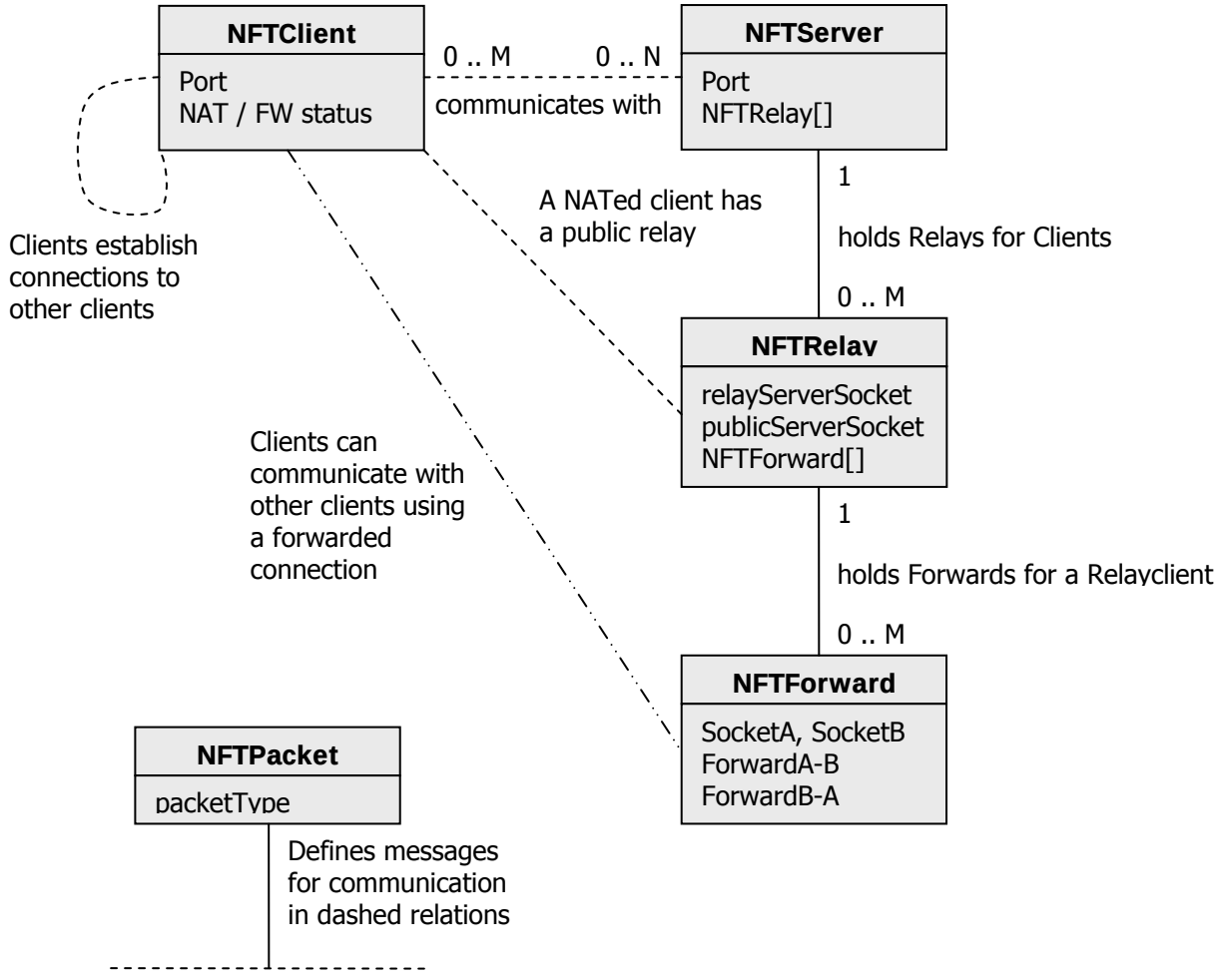

Figure 19: UML-like class diagram of my NFT implementation.

5.2 Messages

Communication messages are defined in the NFTPacket class. This class parses incoming packets and generates outgoing packets. The messages used in the concept are defined in this section in figure 20. The first character in the message format always indicates the type of message

Message used to...	**Message type**	**Format**
Discover NAT or firewall devices	discoverRequest	A[port];
Discover answer to detect NAT / FW	discoverAnswer	B[IP]:[port]
Request an echo containing server load	echoRequest	C[hostNo];
Send back echo containing server load	echoAnswer	D[hostNo]:[metric];
Request a relay server.	relayRequest	E[portMap];
Answer on a relay server request	relayAnswer	F[hostNo]:[portA],[portB];
Non-NATed NFTClient requests a connection	nonNatedConReq	0;
1. Non-NATed NFTClient answers this to a (non)NatedConReq. 2. NFTRelay answers this to its owner for a nonNatedConReq from peerX.	nonNatedConAns	1[IP]:[port];
NATed NFTClient requests a connection	natedConReq	2[portMap];
1. NFTRelay answers this to peerX for a (non)NatedConReq. 2. NFTRelay answers this to its owner for a natedConReq from peerX	natedConAns	3[IP]:[port];
A NATed peerX requests a relayed connection from a NFTRelay to its owner.	relayedConReq	8;
A NFTRelay answers this to its owner and peerX for a relayedConReq from peerX.	relayedConAns	9[IP]:[port];

Figure 20: Table of messages used for communication between NFT processes.

5.3 Implementation Issues

This section summarizes some issues that help to successful implement this NAT and firewall traversal approach.

- In some situations, it is required to reuse the same socket (local IP and port) in close succession. After closing a socket, normally it is locked for a defined time. To avoid this locking 'SocketX.setReuseAddress(true)' has to be set. Further SocketX.setSoLinger(true, 0) is needed, that a SocketX.close() call immediately terminates the connection.
- The server services are organized in three different classes (as visible in figure 18). Each class represents a thread. The NFTServer thread answers general requests and spawns an NFTRelay thread for each succesful relayRequest. The NFTRelay thread serves a single peer, the owner of this relay. The NFTRelay procures direct connections between the owner and a peerX. Further, the NFTRelay spawns NFTForward threads for each successful relayedConReq. The NFTForward thread does TCP packet forwarding for a single forward job (duplex → two-way-communication).
- In this implementation no security aspects are taken into account, since it is a pure test implementation. But a NFTServer and NFTRelay must be save from malicious clients, since a peer running a server provides own resources (such as bandwidth, sockets and CPU-time). The consequence is, that only trusted and authentificated peers are allowed to get a NFTRelay service (and only one per peer). Farther the owner of a NFTRelay service must authenticate itself when connecting to its NFTRelay.

6. Evaluation and Results

6.1 Test Setup

To test my implementation, I arranged the following setup: I used three workstations at home and two Tardis machines from the ETH Zurich. The exact test setup used for evaluation is depicted in figure 21.

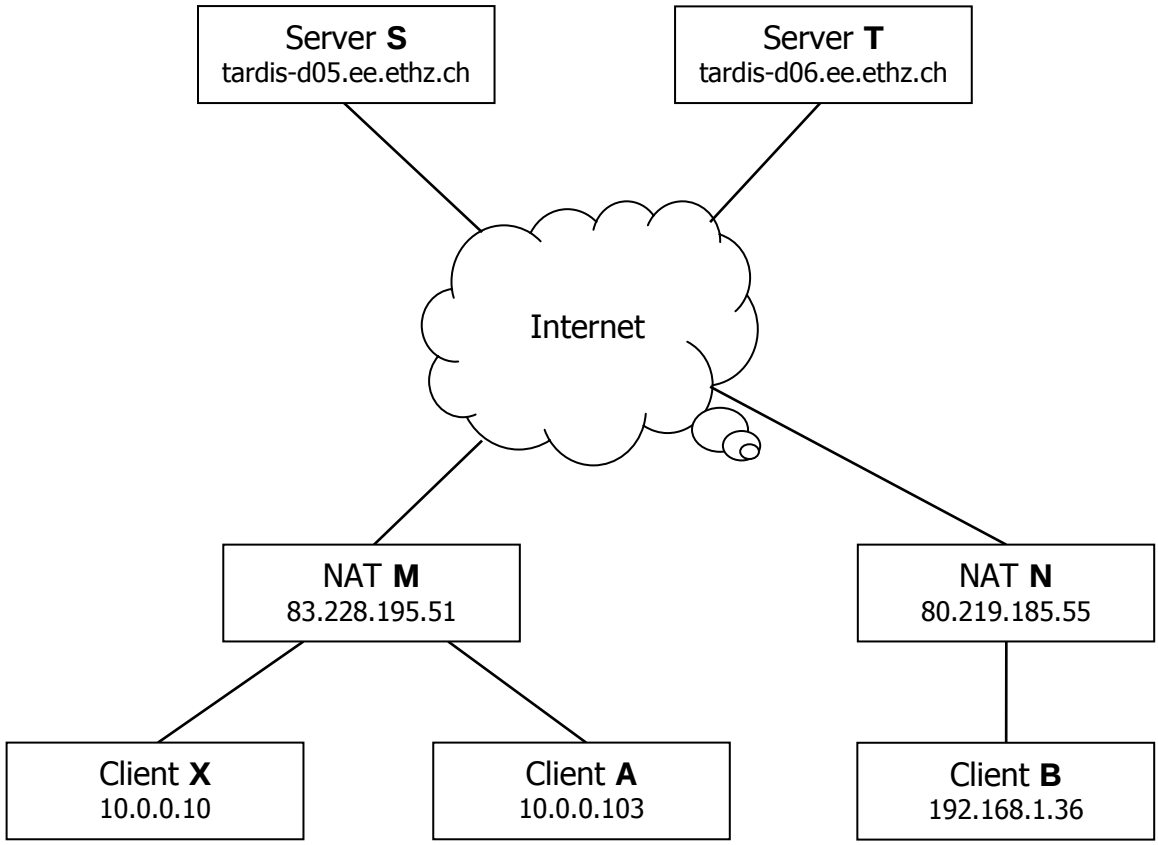

Figure 21: Network test setup used for evaluation.

The nodes of the network depicted in figure 21 have the following functions:

- Client A represents an NFTClient that waits for incoming connections.
- Client B represents an NFTClient that wants to connect to client A.
- Client X is used to control both servers through a SSH session.
- Server S runs an NFTServer service.
- Server T runs an NFTServer service.
- NAT M ZyXEL Prestige 642R-13 ADSL Router with NAT functionality.
- NAT N ZyXEL P335 ADSL Router with NAT and Firewall functionality.

Test procedure:

1. Compiling source files:
   "javac NFTServer.java" in the server source directory.
   "javac NFTClient.java" in the client source directory.

2. Start NFTServer on server S and T:
   "javac NFTServer" in the server source directory.
3. Start NFTClient on client A:
   "javac NFTClient accept NAT" in the client source directory.
   The NFTClient class has a main method implemented which used for testing. The arguments 'accept NAT' cause the client to initialize, request a relayServer and then wait for incoming connections.
4. Start NFTClient on client B:
   "javac NFTClient request NAT" in the client source directory.
   The NFTClient class has a main method implemented which used for testing. The arguments 'request NAT' cause the client to initialize, discover NAT port mapping. And then prompt for an IP address and port number of a NFTClient to which this client connects. The public address that Client A received of its NFTRelay has to entered here.

This goal of this procedure is to establish a connection between A and B, if possible a direct conection is used. When a direct connection attemp, using NAT traversal, fails, then a relayed connection is used. After connecting, a short test message is sent and confirmed.

6.2 Test Results

1. Evaluating NFTServers

The NFTClient class starts with a list containing the two servers S, T and some unreachable hosts. Client A successfully measures server metrics and chooses the server with the shorter latency. In this test client B also evaluates the NFTServers but does not request a relay.

2. Discovering NAT and TCP port mapping

Both, client A and B, detect the presence of a NAT device correctly. Also, the NAT port mapping of both NAT devices is determined. For each new connection from the same local socket, the NAT devices assign a port number that is one higher than the last assigned port number. The conclusion of this is that both devices use "connection-dependent" port mapping.

3. NFTRelay service request

Client A requests an NFTRelay service from the selected NFTServer and thereon receives a public address (IP address and port number) from the NFTServer. After A connected to its NFTRelay, A is prepared to receive connections.

4. Direct connection between client A and B

After entering client A's public address at client B, client B successfully connects client A's NFTRelay. Thereon, the NFTRelay informs client A and B of each other's IP address and predicted port. Unfortunately, client A and B fail to establish a direct connection, but the reason is hard to find, since I cannot observe the packets outside of my NAT devices. Most probably one of the NAT devices filters the SYN-in succeeding the SYN-out packet.

5. Relayed Connection between client A and B

Because the direct connection failed, client B requests a relayed connection to A through the NFTRelay. This relayed connection succeeds and thereon client A and B can exchange their test messages.

Conclusion

To prove proper functionality of the complete implementation, some tests with an artificial network environment are necessary. The network structure could look like in figure 21. But the exact behaviour of the NAT devices must be well known or controllable. Also packet sniffers on each network edge would help to determine the detailed events.

7. Conclusion

7.1 Review

NAT and Firewall traversal is a widespread problem. Especially for NAT traversal, there exist a lot of solutions. Because of the variety of NAT implementations and application requirements, it is not possible to have one straight solution for the problem. As NAT itself, most solutions are complex and a sort of a 'hack'. As a consequence, it is difficult to develop a proper solution. Each approach has its drawbacks; the goal is to find the best solution.
After comparing a set of widespread solutions for NAT traversal, my conclusion is that the concept developed for Celeste is the best fitting (out of my horizon).

7.2 Future work

In this work a first evaluation version was implemented. But no thorough tests in the lab have been performed. The next step in the future is to test correct functionality in a controlled environment. Depending on the results, improvements on the implementation have to be done.
Further, a Firewall traversal solution must be chosen. From my point of view, the Celeste developers must come to this decision, since it can basically affect Celeste (e.g. selecting a well known port as Celeste's standard TCP port).

When this solution turned out to work in the lab, then a prototype has to be integrated in Celeste. After that, the complete system is ready to be tested in the field.
Usage statistics or predictions concerning questions like

- What is the ratio between hosts with a public address and hosts behind NAT?
- What is the success rate of direct connections between two peers behind NAT?

have to be done. This helps to finally determine the performance of this traversal approach.

References

[1] G. Caronni, R. Rom, G. Scott:
Celeste - An Automatic Storage System. White Paper,
http://www.sun.com/products-n-solutions/edu/whitepapers/pdf/celeste_automaticstorage.pdf

[2] The Internet Engineering Task Force: UDP defined in RFC 768,
http://www.ietf.org/rfc/rfc0768.txt

[3] The Internet Engineering Task Force: TCP defined in RFC 793,
http://www.ietf.org/rfc/rfc0793.txt

[4] S. Guha, Y. Takeda, P. Francis:
STUNT in NUTSS: A SIPbased Approach to UDP and TCP Network Connectivity,
http://saikat.guha.cc/pub/fdna04-nutss.pdf

[5] A. Biggadike, D. Ferullo, G. Wilson, A. Perrig:
NATBlaster: Establishing TCP Connections Between Hosts Behind NATs,
http://www.andrew.cmu.edu/user/ggw/natblaster.pdf

[6] B. Ford, P. Srisuresh, D. Kegel:
P2PNAT: Peer-to-Peer Communication Across Network Address Translators,
http://pdos.csail.mit.edu/papers/p2pnat.pdf

[7] J. Rosenberg: TURN: Traversal Using Relay NAT,
http://www.jdrosen.net/papers/draft-rosenberg-midcom-turn-03.txt

[8] uPNP Forum,
http://www.upnp.org

[9] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, A. Rayhan:
MIDCOM: Middlebox communication architecture and framework,
http://www.ietf.org/rfc/rfc3303.txt

[10] Microsoft Corporation:
Windows XP SP2 - Network protection Technology,
http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2netwk.mspx

[11] SIPfoundry: UDP NAT Types,
http://list.sipfoundry.org/archive/ietf-behave/pdf00000.pdf

[12] skype.com : Skype - Guide for Network Administrators,
http://www.skype.com/security/guide-for-network-admins.pdf

[13] W. Stallings: Data and Computer Communications, Prentice Hall

[14] S. Guha, P. Francis:
Characterization and Measurement of TCP Traversal through NATs and Firewalls
http://nutss.gforge.cis.cornell.edu/pub/imc05-tcpnat/