AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY
Faculty of Electronics, Communications and Automation

Veera Andersson

Network Address Translator Traversal for the Peer-to-Peer Session Initiation Protocol on Mobile Phones

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, May 10, 2010

Supervisor: Professor Jörg Ott
Instructor: Jouni Mäenpää

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author:	Veera Andersson
Name of the thesis:	Network Address Translator Traversal for the Peer-to-Peer Session Initiation Protocol on Mobile Phones
Date:	May 10, 2010 **Number of pages:** 100
Faculty:	Electronics, Communications and Automation
Professorship:	S-38
Supervisor:	Prof. Jörg Ott
Instructor:	Jouni Mäenpää, M.Sc.

Network Address Translators (NATs) allow multiple hosts to share one or more IP addresses. The initial decision to use NATs as one of the solutions to Internet Protocol (IP) address depletion, has later induced further challenges; NATs are specially problematic in connection with peer-to-peer (P2P) communication. Interactive Connectivity Establishment (ICE) is a NAT traversal mechanism that helps peers in creating a direct path in the presence of NATs. ICE largely relies upon utilizing the mechanisms of Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) protocols.

Nowadays P2P applications are speading to mobile phones that can also have a NATed address. Knowing the constraints of mobile phones, we were interested in the applicability of NAT traversal mechanisms for mobile phones in the context of Peer-to-Peer Session Initiation Protocol (P2PSIP). SIP was used for controlling communication sessions between the peers. We implemented an ICE prototype for measuring CPU load, memory consumption, packet drop rate and battery consumption of a mobile phone acting as a STUN or TURN client or server. Additionally, we measured the impact of ICE on delays in P2PSIP.

The downside of relaying messages via a TURN server is the increase in delay and the increased overhead due to STUN encapsulation. A TURN server running on a mobile phone has to limit the number of allocations and the type of data being transmitted through it. A mobile phone works well as STUN server, especially if keepalives can simply be ignored. Mobile phones can act as P2PSIP peers and TURN servers, even in the presence of NATs, however, it is preferable to have NATs using address and port-independent mapping, since then no relaying is needed.

Keywords: NAT, STUN, TURN, ICE, NAT traversal, P2PSIP, mobile

AALTO-YLIOPISTO DIPLOMITYÖN
TEKNILLINEN KORKEAKOULU TIIVISTELMÄ

Tekijä	Veera Andersson		
Työn nimi:	Osoitteenmuuntajien läpäisy vertaisverkon istunnon-aloitusprotokollaa käyttävälle matkapuhelimelle		
Päivämäärä:	10.5.2010	**Sivuja:**	100
Tiedekunta:	Elektroniikka, tietoliikenne ja automaatio		
Professuuri:	S-38		
Työn valvoja:	Prof. Jörg Ott		
Työn ohjaaja:	DI Jouni Mäenpää		

Osoitteenmuuntajat sallivat useiden isäntäkoneiden jakavan yhden tai useamman IP osoitteen. Päätös käyttää osoitteenmuuntajia yhtenä ratkaisuna IP osoitteiden ehtymiseen, on myöhemmin tuonut mukanaan lisähaasteita; osoitteenmuuntajat ovat erityisen ongelmallisia vertaisyhteyksille. ICE (Interactive Connectivity Establishment) on osoitteenmuuntajien läpäisymenetelmä, joka auttaa vertaiskoneita luomaan suoran polun osoitteenmuuntajien läsnä ollessa. ICE perustuu suurilta osin STUN (Session Traversal Utilities for NAT) ja TURN (Traversal Using Relays around NAT) -protokolliin.

Nykyään vertaissovellukset ovat levinneet matkapuhelimiin, joilla voi myös olla osoitemuutettu osoite. Matkapuhelinten rajoitukset tietäen, on kiinnostavaa tietää osoitteenmuuntajien läpäisymenetelmien soveltuvuus matkapuhelimille P2PSIP:n (Peer-to-Peer Session Initiation Protocol) yhteydessä. SIP:iä käytettiin kommunikointi-istuntojen hallintaan vertaiskoneiden välillä. Toteutimme ICE-prototyypin mitataksemme STUN tai TURN asiakkaana tai palvelimena toimivan matkapuhelimen suorituskykyä huomioiden keskusyksikön kuorman, muistinkäytön, pakettien pudotusmäärän ja akun kulutuksen. Lisäksi työssä tutkittiin ICE:n vaikutusta P2PSIP:n viiveisiin.

TURN välityspalvelimen käytön haittapuoli on kasvanut viive ja STUN koteloinnista johtuvat ylimääräiset tavut. Puhelimessa toimivan TURN palvelimen tulee rajoittaa asiakkaiden määrä sekä millaista dataa se voi välittää. Puhelin toimii hyvin STUN palvelimena, etenkin jos yhteyden ylläpitoviestit voidaan jättää huomiotta. Puhelimet voivat toimia osana P2PSIP-verkkoa myös osoitteenmuuntajien läsnä ollessa. On kuitenkin suotavaa, että osoitteenmuuntajat käyttäisivät osoite- ja porttiriippumatonta kuvausta, koska silloin välitystä ei tarvita.

Avainsanat: NAT, STUN, TURN, ICE, osoitteenmuuntajien läpäisy, P2PSIP, mobiili

Acknowledgements

This Master's thesis has been carried out at Ericsson Research Finland, NomadicLab, as part of the Decentralized Inter-Service Communication (DECICOM) project and the ICT cluster of the Finnish Strategic Centres for Science, Technology and Innovation (ICT SHOK) Future Internet Programme.

I wish to thank my thesis supervisor Jörg Ott for his guidance and valuable feedback. It was really nice to work with him.

Jouni Mäenpää has instructed my thesis and I want to thank him for all his efforts. I would also like to thank my other colleagues at the NomadicLab; especially Ari Keränen for all the valuable discussions on the topic.

Finally, I owe my deepest gratitude to my friends, and especially to my mom and my sister for all the support throughout my studies.

Espoo, May 10, 2010

Veera Andersson

Contents

Abbreviations

3G	Third Generation
ALG	Application Layer/Level Gateway
AoR	Address-of-Record
API	Application Protocol Interface
ASCII	American Standard Code for Information Interchange
ASP	Address Settlement by Peer-to-peer
CAN	Content Addressable Network
CLDC	Connected Limited Device Configuration
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DB	Database
DHT	Distributed Hash Table
DNS	Domain Name Server
DTLS	Datagram Transport Layer Security
GSM	Global System for Mobile Communications
HIP	Host Identity Protocol
HMAC	Hash-based Message Authentication
HSDPA	High Speed Downlink Packet Access
ICE	Interactive Connectivity Establishment
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IM	Instant Messaging
IP	Internet Protocol
ITU	International Telecommunication Union
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JP	Java Platform

JSR	Java Specification Request
JVM	Java Virtual Machine
MIDP	Mobile Information Device Profile
NAPT	Network Address Port Translator
NAT	Network Address Translation / Translator
P2P	Peer-to-Peer
P2PP	Peer-to-Peer Protocol
P2PSIP	Peer-to-Peer Session Initiation Protocol
PC	Personal Computer
PDA	Personal Digital Assistant
QoS	Quality of Service
RELOAD	REsource LOcation and And Discovery
RRC	Radio Resource Control
RTP	Real-time Transport Protocol
RTT	Round-Trip Time
SDP	Session Description Protocol
SHA	Secure Hash Algorithm
SIP	Session Initiation Protocol
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLV	type-length-value
TURN	Traversal Using Relays around NAT
VoIP	Voice over IP
UDP	User Datagram Protocol
UML	Unified Modeling Language
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
USB	Universal Serial Bus
WCDMA	Wideband Code Division Multiple Access
WLAN	Wireless Local Area Network

List of Figures

List of Tables

Chapter 1

Introduction

The word 'Internet' was first introduced in the early 1980's. Since then the Internet has changed tremendously, both in size and the way people communicate using it; not to mention its increased performance. At first, the use of Internet was primarily limited to a small number of users, such as scientists, universities, and the military. Therefore, the 32-bit Internet Protocol (IP) version 4 [2] address space with approximately four billion addresses was considered more than enough for future utilization – an assumption that turned out to be very wrong. Nowadays the Internet is accessible by almost everyone and nearly everywhere. In addition, the number of different access devices has grown, varying from wired computers to handheld personal digital assistants (PDAs). As a result, the need for globally uniquely addressable devices has grown as well as the challenges in routing between these devices.

Network Address Translators (NAT) were devised as a short-term solution to the problem of IP address depletion and scaling in routing [48]. NATs are devices capable of sharing one or more globally unique addresses between multiple hosts. The hosts behind a NAT form a private network that uses internally unique private addresses making the private network also more secure. These private addresses can be reused within other private networks. Despite the fact that many long-term solutions were identified during the same time, such as larger IP address space, NATs have gained widespread popularity in the Internet [48, 22]. One reason for the increased popularity is their simplicity of deployment; only changes in the routers at the edge of the network are required [48].

Some of the drawbacks that NATs cause were known from the start, such as problems with end-to-end security mechanisms and applications using IP address information in the IP

packet payload, others have followed from the fact that all communication is no longer client-server based. In peer-to-peer (P2P) communications, a session can be initiated by either one of the communicating parties. Since any host is capable of acting as a peer, it is possible for peers to be located behind a NAT, which makes contacting them a bit more problematic. To help setting up a connection, peers often need to make use of a signaling channel, such as one created using the Session Initiation Protocol (SIP) [11].

The signaling channel is used for exchanging control messages, but it is inefficient for carrying data traffic, since it usually results in a possibly indirect path between the peers. NAT traversal is the way to work around the problems caused by NATs in peer-to-peer environments. Session Traversal Utilities for NAT (STUN) [44] and Traversal Using Relays around NAT (TURN) [43] are tools for other protocols to use to traverse NATs. How well the tools work depends on the characteristics of the NATs between the peers. STUN and TURN can be used as such, but they are usually used as part of a full NAT traversal solution, such as Interactive Connectivity Establishment (ICE) [42]. STUN is a mechanism that a host can use to discover its globally routable address that another host may try to use for contacting it. This address might be either an address on one of the host's directly attached interfaces or an address assigned by the NAT. STUN tries to provide a direct path to the host, whereas TURN is used to provide a relayed path. Thus, when using the TURN protocol, a client can request a TURN server in the external network to relay data messages between the client and its peers. This solution is very likely to work, but it is not considered efficient due to the relaying costs. ICE is used to find the optimal path for communication, which means relaying messages only as a last resort.

P2P networks, such as Peer-to-peer Session Initiation Protocol (P2PSIP) networks, have to implement multiple protocols to keep the P2P overlay function. Due to the decentralized nature of a P2PSIP network a distributed database algorithm is needed for locating a peer with a particular data item within the network. Furthermore, a P2P signaling protocol is required for purposes of maintaining the network. SIP is used for enabling communication between the peers. Even though the network maintenance requires more complicated co-operation of peers than a simple client-server model, the efforts pay of since a P2P network is self-organized and more scalable.

In addition to computers, also mobile phones can have a NATed address. Even though the problem with NATs and the solution to the problem stay the same, there are other issues to be taken into consideration. The constraints of mobile phones relative to computers are well-known: limited memory, battery and processing power. An interesting subject for research is knowing whether these constrains are a limiting factor for the use of NAT traversal tools in mobile phones.

1.1 Objectives and Scope

The applicability of the ICE protocol as a NAT traversal solution has been shown in setups where the nodes running the protocol are computers [28]. The objective of this thesis is to examine the applicability of the protocol when computers are being replaced by mobile phones. The interest for the study arose as the need for NAT traversal in the context of mobile phones became apparent. To be able to provide meaningful results on the power consumption and the sufficiency of processing capacity of a mobile phone, an ICE protocol library was implemented for Java 2 Micro Edition (J2ME). Some measurements are also made in comparison to the protocol usage on computers. Our main focus is on establishing a phone call between wired and wireless peers in the presence of different type of NATs in a P2PSIP network of reasonable size. This makes it possible to compare the difference in call setup times for different NAT scenarios. Additionally, the performance of a mobile phone as a STUN and TURN client and server is measured.

Protocols intended to traverse NATs might also be usable for traversing other type of middle-boxes. However, regarding this thesis, those are out of scope. There is also a restriction pertaining to the supported transport protocols: we only discuss NAT traversal for UDP-based traffic due to the multiple additional challenges related to TCP NAT traversal, and also due to the lack of standardized mechanisms for TCP NAT traversal [20]. Additionally, questions regarding the incentives of peers to act as relays in P2P networks are out of scope.

1.2 Structure

In this Chapter the subject area and its scope were briefly introduced, as well as the purpose and goal of writing the thesis. Chapter 2 provides a deeper insight to the topic of the thesis. Chapter 3 describes the implemented ICE prototype, including some experiences and challenges concerning the implementation. Chapter 4 describes the prototyping environment and includes the actual measurement results. The chapter also analyzes the measurement results. In Chapter 5, we discuss the applicability of the NAT traversal protocols in mobile peer-to-peer networks based on the results presented in the previous chapter. Finally, in Chapter 6, we sum up the results and draw final conclusions.

Chapter 2

Background

In this chapter we present the concepts of NATs, NAT traversal, and P2P communication. We start by introducing NATs and take a more detailed look at two of the most common NAT types. We will explain the advantages and disadvantages of NATs. Additionally, we present how NATs are classified based on their characteristics. Then we explain NAT traversal and cover some of the most relevant NAT traversal techniques in more detail. To show the importance of NAT Traversal in today's Internet, we give a brief overview on the existence of different NAT types. The background on P2P communication includes an overview of Peer-to-Peer Session Initiation Protocol (P2PSIP) and how peer-to-peer networking makes use of distributed hash tables, such as Chord. Moreover, we introduce two peer-to-peer signaling protocols: REsource LOcation And Discovery (RELOAD) and its predecessor protocol, known as Peer-to-Peer Protocol (P2PP). Finally, we briefly describe the basic capabilities of a mobile phone.

2.1 Network Address Translation

Network Address Translation is a method that enables IP address reuse. It is realized by placing a Network Address Translator (NAT) at the border of a local network with private IP addresses. It is the responsibility of the NAT to translate these locally unique addresses to globally unique addresses while accessing the external network. The NAT device maintains a table with private-public address mappings. Since private and public addresses have to differ for network address translation to function, the IP address space is divided into two parts. Private IP addresses used within a local network can be reused within any other local

network. The network address translation is done transparently to the end hosts. This means that a host behind a NAT sees as though packets it receives from the external network would have been designated to its own local node IP address already at the source of the message. Likewise, the external host does not know that packets it sent to an external address of the NAT are actually forwarded to a local host behind the NAT. The illusion is created by careful header manipulation done by the NAT. Header manipulation concerns at least the IP address, the IP checksum and TCP checksum (in case of TCP). Also certain type of Internet Control Message Protocol (ICMP) error messages, which include the original IP packet, require changes in their payload [16, 50].

The life cycle of a single NAT session consists of three phases. In the first phase, a local address gets an external IP address assigned to it. In static address assignment, a given host is configured to be always associated with the same public IP address when connecting to the external network. In the case of dynamic assignment, addresses are assigned dynamically, meaning that a host might receive different bindings at different times, and that a binding used by the host might be reused by other hosts after the session with the host in question has terminated. In the second phase, when the NAT receives a packet belonging to an already existing session, it needs to perform address lookup to confirm the existence of the session and modify the packet to keep the translation transparent. Finally, as the NAT assumes the session to have finished, it needs to unbind the address association. The address may now be freely used by other sessions. To prevent a premature unbinding from happening, hosts need to keep the binding alive via regular transmissions. [50]

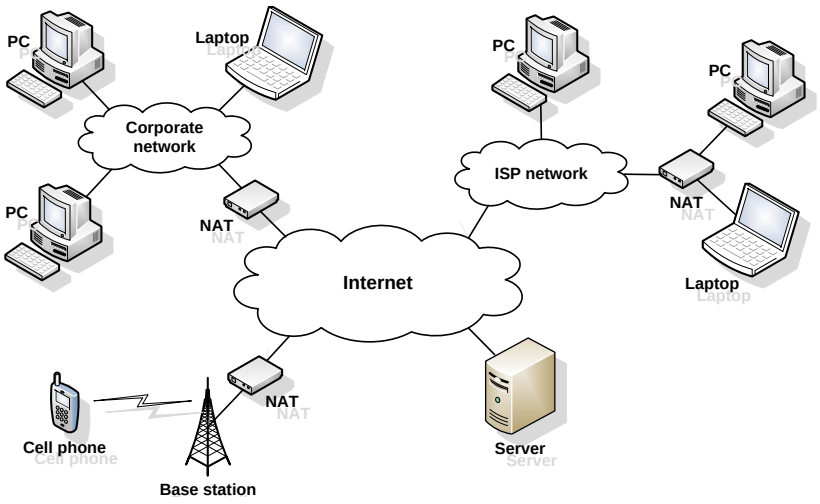

Figure 2.1: Example NAT scenarios

Some typical NAT scenarios are shown in Figure 2.1. A company uses private addressing to hide its addresses and address structure from the external network. Since it is a company network, most of the communication is internal, meaning that an address translation does not even always take place [16]. A NAT can also be used to block unknown traffic from entering the network. Another typical scenario is a user at home wanting to gain access to the Internet via a laptop and a PC simultaneously, but having only one IP address available. By connecting both of the devices to the Internet through a NAT makes it possible to share a single public address between both of them. In addition, when a mobile phone connects to the network, it can get an address assigned to it that is only unique within the cellular network it resides in. NATs come in many flavors, but in the following subsections only traditional NATs, as being the most common ones, are presented in more detail. Traditional network address translation can be based on either basic network address translation or Network Address and Port Translation (NAPT).

2.1.1 Basic Network Address Translator

In Basic Network Address Translation (Basic NAT), a NAT has a set of public IP addresses to be shared by the hosts behind the NAT. The translation applies only to the IP addresses. The maximum number of hosts being able to simultaneously access the public network is equal to the number of public IP addresses that the NAT has. If there are more hosts, access to the Internet cannot be guaranteed. All sessions initiated from the same host use the same mapping. Taking advantage of static address assignment, certain hosts can have guaranteed access at all times. [48]

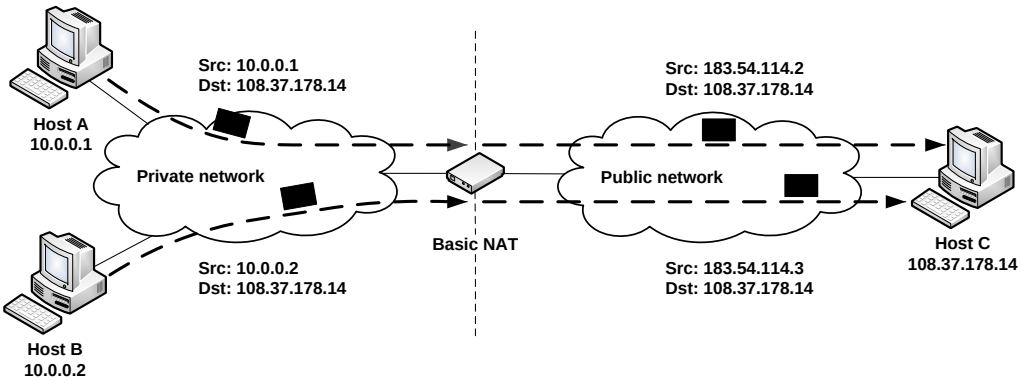

Figure 2.2: Basic NAT with outbound traffic

Figure 2.2 shows the operation of a Basic NAT. Two hosts, Host A (10.0.0.1) and Host B (10.0.0.2), which are located in the same private network, both want to set up a connection to Host C (108.37.178.14) in the public network. Initially, neither of the hosts has any ongoing sessions. As the first outgoing packets traverse through the NAT, the NAT assigns the hosts new addresses (183.54.114.2 and 183.54.114.3) that are used outside the private network. The assigned address does not depend on the destination address of the packet. The same translation is used for all subsequent packets belonging to the same host until all of its sessions have finished. After that the address can be used by other hosts.

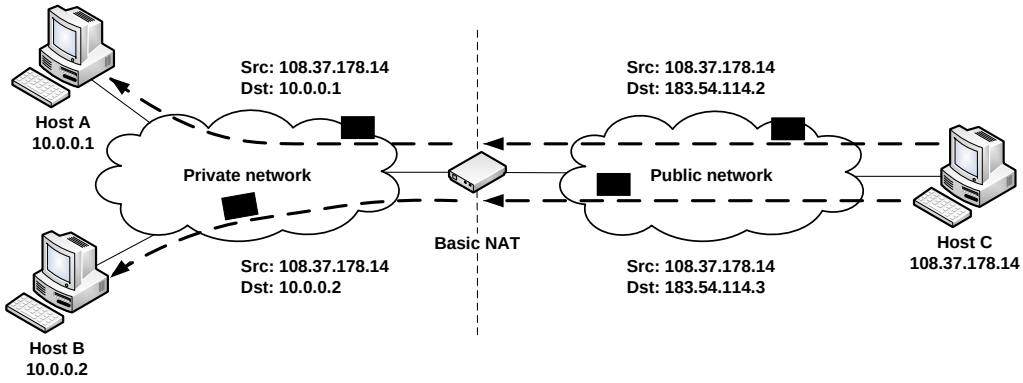

Figure 2.3: Basic NAT with return traffic

Figure 2.3 shows the operations that the Basic NAT makes for the return traffic. The translation is performed by replacing the destination address of the packet with the private address. As Host C receives packets, it does not need to know that the addresses it sees as source addresses are actually public addresses assigned by the NAT. This is why it sends its return packets back to the assigned public addresses just like to any other node with a public address. Regarding basic NAT, the translation done by the NAT applies to the IP address and the IP checksum of the IP header, as well as the checksum in the TCP and UDP headers (if used). Likewise, modifications to the ICMP error messages concern only the IP address and checksum parts in the payload. [48]

2.1.2 Network Address and Port Translator

In Network Address and Port Translation (NAPT), the translation is applied to the IP address as well as the transport layer identifier (i.e., port number) part of the packets. The advantage of this is that only one public IP address is needed to enable multiple hosts to

simultaneously access the public network. Different hosts and the different sessions on the same host are distinguished by assigning them different transport layer identifiers. The services supported by a NAPT router are restricted to UDP, TCP and ICMP query sessions.

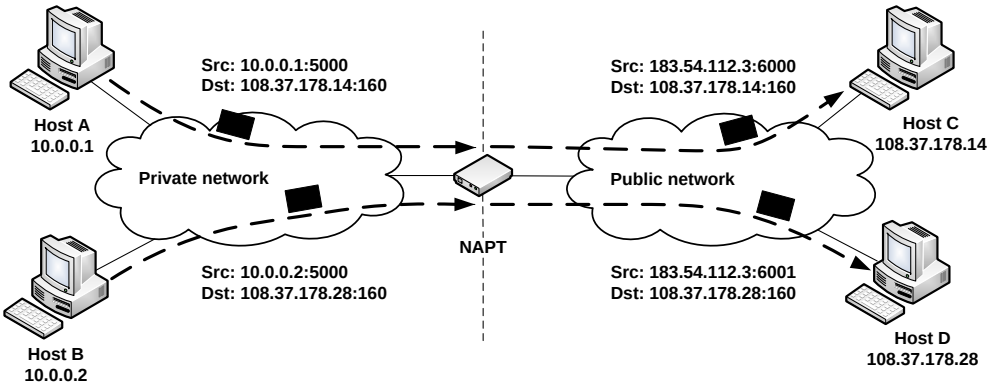

Figure 2.4: NAPT with outbound traffic

Figure 2.4 presents two ongoing sessions through a NAPT. Hosts A (10.0.0.1) and B (10.0.0.2) are in the private network and hosts C (108.37.158.14) and D (108.37.178.28) are in the public network. Host A has set up a connection with Host C and Host B similarly with Host D using the same transport identifiers as source and destination ports. The figure shows the translations that take place for the outgoing packets of the two ongoing sessions. Since NAPT has only one single address to share, both sessions have the same public address as source after the translation. However, the packets have different source ports to distinguish that they belong to different hosts and sessions (183.54.112.3:6000 and 183.54.112.3:6001).

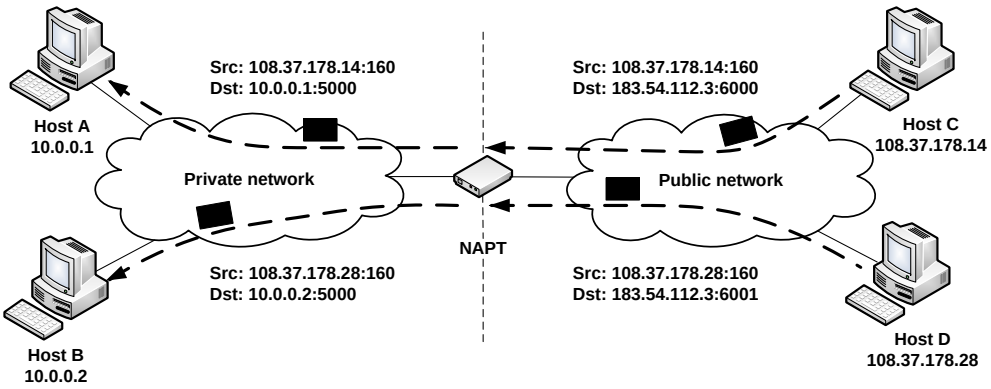

Figure 2.5: NAPT with return traffic

Figure 2.5 shows how the translation is performed for the return traffic. The translation is done by replacing the destination address and port with the corresponding private address and port.

In case of NAPT, translating the IP address and the IP checksum of the IP header, as well as the checksum in the TCP and UDP headers is not enough. In addition, the TCP/UDP ports of the TCP and UDP headers must be modified. Also the modifications in the payload of an ICMP error packet must be extended to cover port modifications. [48]

2.1.3 Benefits of Network Address Translation

One of the advantages is allowing multiple hosts to share a single IP address for accessing the Internet, which naturally slows down the IP address depletion. However, NATs might also be useful for other reasons, such as privacy. The fact that connections cannot be initiated from the outside provides privacy for the internal network, and the actual addresses of the private hosts are usually not seen in the external network. [48]

Moreover, there is no need to change the internal addresses if the outside topology changes. A NAT can take care of the changes in a centralized way. Another benefit comes from is its simple deployment. The installation requires only changes to the address translation router. For hosts and other routers the translation is entirely transparent. [48]

A private network does not have to be geographically located in the same place. This is common with corporate networks that are spread to different locations along with the offices. To avoid the need of address translation, it is possible to share the same private address space between the separately located corporate networks. Since the packets have to traverse via the external network, where the private addresses are invalid, NATs encapsulate the packets inside an IP packet that uses externally available addresses. From the private hosts point of you, all the hosts belong within the same private network. NATs have one address especially for the encapsulation. This type of network is called a backbone-partitioned stub [16]. In case of virtual private networks (VPN), the tunneled packets need to be encrypted to ensure privacy. [50]

2.1.4 Drawbacks of Network Address Translation

Network address translation violates some very fundamental architectural principles of the Internet. First of all, it breaks the end-to-end model by moving intelligence from the edge to

the core network. NAT also breaks the concept of fate-sharing by reducing robustness. Fate-sharing states that only an end point itself can destroy a state of its own. The principles also refer to the concept of "transparency", which means unaltered communications between the end points using unique labels. Breaking the principles is not only a principle level issue, but incurs some practical drawbacks. By making all communications flow through the NAT creates a single point of failure. This could be avoided by using multiple NATs, but it would cause further challenges, like timely communication of the state and routing related failures. [12, 22]

One of the disadvantages with NATs are the complications they cause on several protocols. The problem is that some protocols include IP address or port information into the payload of IP packets. This breaks the fundamental idea of transparency that NATs are supposed to provide. Some of the protocol complications can be worked around by extending NATs to offer protocol specific aid by means of Application Level Gateways (ALG), others will fail anyway. But since the goal should be the ability to add new applications at end points without requiring changes to the infrastructure, adding ALGs increases the complexity. When using ALGs, application updates are required to multiple locations and applications need to be multiplexed. In addition, debugging gets more complicated. One of these protocols with complications is Session Initiation Protocol (SIP), which exchanges address and port information in the payload of the packets it sends. One example of a protocol unable to work through NATs as such is IPsec, since one of its explicit purposes is to detect alterations to the IP packet header by encryption. The problem with IPsec can, however, be solved with NAT-T (NAT Traversal in the Internet Key Exchange). Moreover, some protocols require retaining the same mapping for multiple sessions. However, a NAT cannot know that this is required, since it is designed to recycle addresses between different hosts and different sessions. [22, 16, 24, 50]

NAT, ALG and firewall work together to achieve an even more secure private network. If the NAT router is part of the trusted boundary, it can be used to implement IP security by serving itself as the other end of the tunnel. For example, ALGs can be used for checking that the payload or header of a packet does not include private IP addresses. The firewall filter can be used to drop such packets. It is also good to keep in mind that a NAT itself can be a target of an attack. This is why a NAT should use protection mechanisms similar to a server. [50]

One disadvantage closely related to the subject of this thesis are the challenges with peer-to-peer applications. Most of the NATs in use have been designed for client-server based communications, where the other node acts as a server in the external network and connec-

tions to it are initiated by a client either in private or public network. In P2P communications, there is no clear distinction between a client and a server. Communication can be initiated by either of the communicating parties. Additional problems are due to the variety of different NATs. Different NATs use different binding as well as filtering rules. Another problem is that the hosts inside the private network have no externally visible addresses for the hosts outside to contact. Therefore, the addresses to contact have to be communicated via a signaling protocol, like SIP. NAT traversal takes care of finding the best working path between the peers based on the signaled addresses. [49, 24]

Even though network address translation is compute intensive, it is not necessarily a problem. Especially as long as the NAT device is able to process packets at a higher speed than the transmission speed the line is able to achieve. NAT uses an effective checksum adjustment algorithm, since every packet needs to be translated. [24]

2.2 NAT Classification

To understand the different circumstances that NAT traversal is supposed to cope with, there is a need to classify the different NAT behaviors. The following sections clarify the properties of NATs by classifying them based on mapping, filtering and port assignment behavior. The firewall specific settings that might inhibit communications do not belong as part of the NAT classification. [4]

2.2.1 Mapping Behavior

Every time a private host initiates a new session to a host in the public network it gets a public address and port assigned to it (assuming NAPT). The assigned address and port are used for translating all the subsequent packets belonging to the same session. Mapping behavior is used to describe the different rules for mapping a private address and port into a public IP address and port. Even if different sessions would originate from the same source, it does not guarantee that the same mapping will be used. In fact, it also depends on the endpoint the packets are destined to. Thus, the mapping behavior can be divided into endpoint-independent mapping, address-dependent mapping, and address and port-dependent mapping. [4]

Endpoint-independent mapping means that the NAT re-uses the same mapping for all packets originating from the same private address and port regardless of the destination address

and port. Figure 2.6 shows an example of endpoint-independent mapping. In the figure, the NAT uses the same mapping for both of the sessions from the same origin, no matter what the address or port of the external sessions endpoint is.

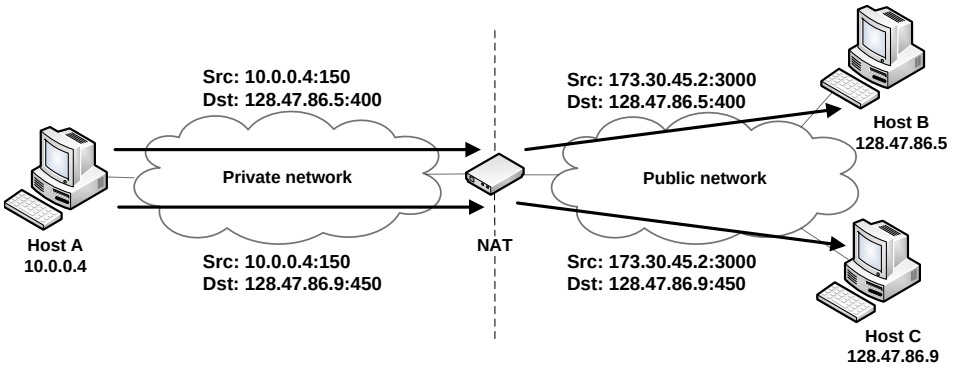

Figure 2.6: Example of endpoint-independent mapping

Address-dependent mapping means that the NAT re-uses the same mapping for all packets originating from the same address and port to the same destination address, regardless of the port used. Figure 2.7 shows the same situation as Figure 2.6, except that in this case, the NAT assigns a different mapping since the session endpoints have different addresses.

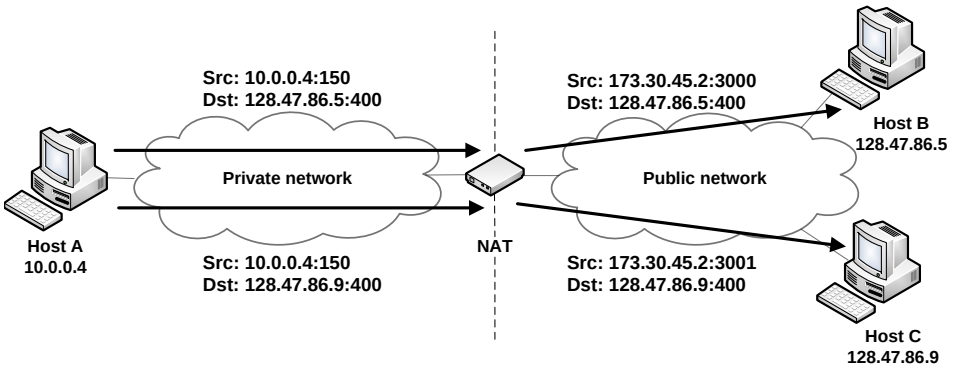

Figure 2.7: Example of address-dependent mapping

Address and port-dependent mapping means that the NAT re-uses the same mapping for packets originating from the same address and port only if the destination address and port are the same. Figure 2.8 shows two sessions between same hosts, however, with two different mappings, because the sessions use different ports on the external hosts.

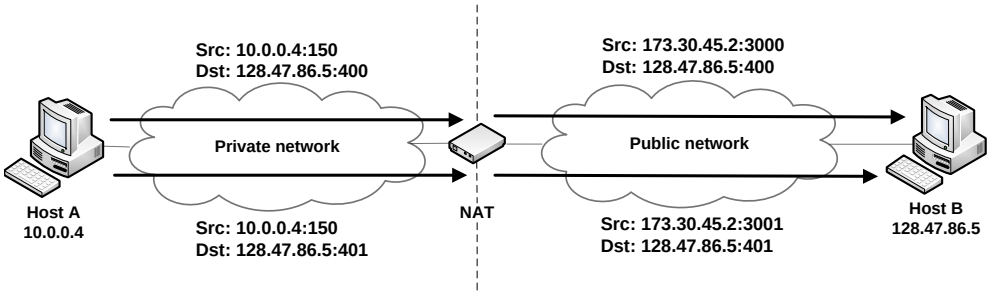

Figure 2.8: Example of address and port-dependent mapping

2.2.2 Filtering Behavior

Filtering behavior is used to describe the filtering rules applied to incoming packets on an external address and port of a NAT. The criteria for the filtering is based on filtering rule defined for an internal and external endpoint pair. The rule can be based on either endpoint-independent, address-dependent, or address and port-dependent filtering. [4]

When *endpoint-independent filtering* is applied, all messages destined to an internal address and port with a mapping are forwarded. The external address and port used for sending a packet does not effect the filtering.

In *Address-dependent filtering*, only packets originating from the same external address where the internal host has already sent packets to are accepted. This means that the external host may use any source port for reaching the internal host.

Address and port-dependent filtering means that only packets originating from an external address and port, that the internal host has already sent packets to, are accepted. All other packets are rejected.

2.2.3 Port Assignment Behavior

Port assignment behavior describes the rules that NATs use for assigning ports. [4]

When a NAT supports *port preservation*, it means that a NAT tries to use the same port for the assigned external address as is used for the private address. In case that port is already assigned for another mapping, a port collision occurs. It is up to the NAT how such a collision is handled; it can either override the previous mapping, assign a different port or

use the same address with a different external address (in case a NAT has multiple external addresses to choose from and one of them is unused). If a NAT uses *port overloading*, most applications will fail, since port preservation is always used despite of possible collisions.

If port preservation is not supported, a NAT does not try to make any efforts in preserving a port. This is known as the "*No Port Preservation*" -behavior.

2.2.4 Hairpinning Behavior

A NAT is said to support hairpinning, if two hosts behind the same NAT are capable of communicating with each other using the public addresses assigned by the NAT. The Figure 2.9 shows an example of a NAT that supports hairpinning. Host A sends a message to host B using B's public address, and the NAT relays the message to reach its destination. The figure shows the relevant translations.

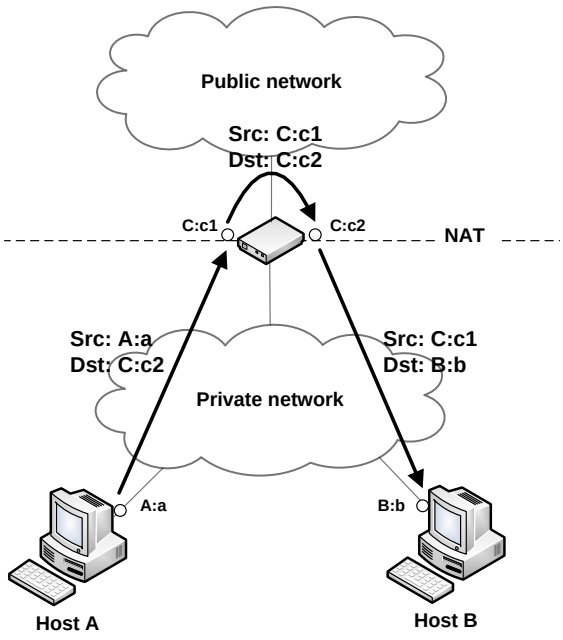

Figure 2.9: Example of a NAT supporting hairpinning

2.3 NAT Traversal

With the advent of peer-to-peer applications, such as file sharing, Voice over IP (VoIP) and online gaming, the need for NAT traversal became apparent. Some applications rely on a central server to maintain a list of the connected hosts and the resources they have, while the actual data traffic is transmitted directly between the hosts. A direct communication is preferred in order to reduce transmission latency. Nowadays, a popular alternative is creating an entirely decentralized P2P network that is managed in collaboration between the peers, with no or little involvement of servers.

As described earlier, the problem with NATs is that hosts behind a NAT, without a public address assigned to them, can only be contacted by hosts within the same private network. Even if a host would have a public address assigned to it, it depends on the type of the NAT whether an incoming connection is accepted or not. Further, NATs were not until recently standardized, so one workable solution for all situations is not achievable [49, 4]. Consequently, the requirements for P2P communication work badly with NATs. But due to the popularity of P2P applications and the generality of NATs, several NAT traversal techniques have been developed. Almost all connection attempts to a private network are inhibited, unless there are static bindings. The Internet's architecture is designed to work for client-server based communications where all the connections are initiated by the client and servers are located in the public network. However, peer-to-peer communications is not as straightforward in the presence of NATs. Peer-to-peer communication requires direct connections that can be initiated by either of the peers. [49]

NAT traversal is a collection of different mechanisms to create connections through different type of NATs. Since the type of NATs is not known by the endpoints, multiple mechanisms may need to be tried out before finding a working one. One simple and practical technique for UDP that many NATs support is known as "UDP Hole Punching". Hole punching does not require any changes to be made to the infrastructure, instead it tries to work around the security policies of most NATs. UDP hole punching makes use of a well-known rendezvous server for setting up a direct peer-to-peer UDP session. By means of a rendezvous server it is possible to know if a client is behind a NAT or not by comparing the address that the server sees the client is using with the address the client thinks it is using. If the addresses differ, the client is behind a NAT. STUN is one example of a protocol that uses UDP hole punching for letting a client know its globally valid address and finding out whether it has a NATed address. [49]

The most reliable, but least efficient method is to relay messages between the peers via a server in the external network. This solution is likely to work since it makes the communication look like a normal client-server communication. Of course, with the basic assumption of being able to connect to a server. This solution has, however, clearly and easily identifiable drawbacks. Firstly, it consumes processing power and network bandwidth. Secondly, it induces additional latency between the communicating peers. But since it provides a guaranteed solution, peers keep it as an option in case other attempts fail. TURN is an example of a protocol that provides such a relaying service. [49]

Despite the drawbacks of NATs, the need for NAT traversal is likely to remain for a long time. NATs are useful for the incremental deployment of IPv6 [15] addresses in translating between IP version 4 and version 6 addresses [22]. Additionally, to fully upgrade an existing infrastructure to support IPv6 services takes time [3]. Even after a full integration of IPv6, the need for NAT traversal will still remain; NATs are not the only type of middleboxes that need to be traversed, there are also other types of middle-boxes that benefit from using it, such as firewalls. Besides, even if more than enough IP addresses would be available, the benefits of private networks still make network address translation a desirable feature.

2.3.1 STUN

Session Traversal Utilities for NAT (STUN) is a protocol that a host can use to discover its globally routable address. This is done by sending a request to a server in the public network to learn the address that the server sees as the address of the client. This is called the (server) reflexive transport address. The returned address might be an address on one of the client's directly connected interfaces or it might be an address assigned by a NAT. A client knows the address on its directly connected interface. Thus, if this address differs from the one returned by the server, the client knows it is behind a NAT. Depending on the type of the NAT, a host in the public network could be able to use the returned address for contacting the host behind the NAT. STUN also provides a mechanism to keep the binding on the NAT alive. This solution works properly for NATs performing endpoint-independent mapping [54]. STUN is not a complete NAT traversal solution in itself, but works as part of a full solution. [44]

Figure 2.10 shows a possible STUN configuration. The host behind a NAT is called a STUN client, and the server on the external side is called a STUN server. Communication between the STUN agents is possible since STUN is a client-server protocol. With STUN agents we refer to entities implementing the STUN protocol. There are two types of STUN

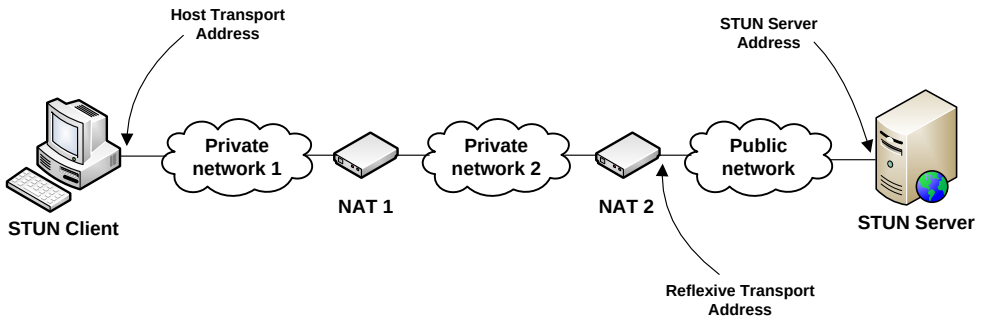

Figure 2.10: Example of a STUN configuration

transactions: request/response and indication. In a request/response transaction the client sends a request to the server and the server replies with a response. Indication messages are never replied to. STUN transactions are distinguished via a transaction ID. The transaction ID is a randomly chosen 96-bit identifier used to associate requests with responses. The transaction ID is included in the fixed 20-byte header of a STUN message alongside with the message type, length and magic cookie. A 32-bit magic cookie value is one of the ways used to distinguish STUN messages from other messages. The format of a STUN message is shown in Figure 2.11.

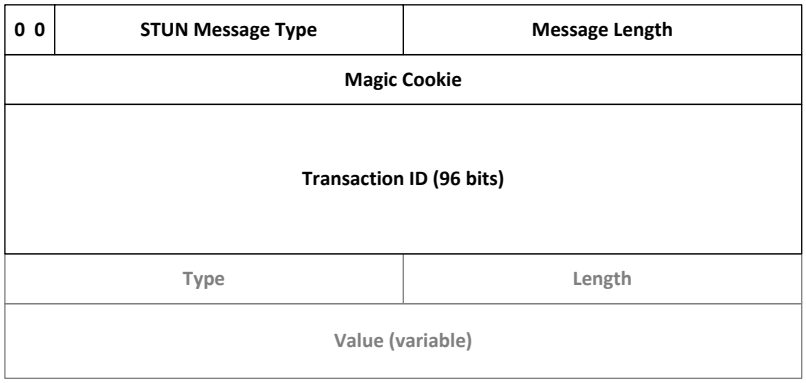

Figure 2.11: Format of a STUN message including a STUN attribute

The STUN message type field is a combination of the method and class values. A single method, called Binding, is needed for the basic STUN functionality to work. The message classes are request, success response, failure response and indication. The different STUN message types are summarized in Table 2.1. Following the header, a message can have

Table 2.1: STUN message types

Message type	Description
Binding request	Client requests for a Binding response
Binding success response	Includes the reflexive transport address
Binding error response	Includes the type of the detected error
Binding indication	Keeps the binding on the NAT alive

zero or more attributes specified by the method or the usage. In the Figure 2.11, the header is followed by a single attribute (marked in grey). The STUN protocol defines a generic message format that can be extended to provide additional features for other protocols using STUN for NAT traversal. The STUN attributes are listed in the Table 2.2 below.

The basic STUN functionality involves the client sending a Binding request to the server and the server replying with a Binding response, that includes a XOR-MAPPED-ADDRESS attribute containing a combination of IP address and port, identifying the client as seen by the server on the public network. This way the client learns its reflexive transport address. Since UDP is not a reliable transport protocol, STUN request might need to be retransmitted. The retransmission timeout doubles after every retransmission starting with a 500 ms timeout. A maximum of 7 retransmissions is used, in case no response is received before that. After a successful request/response transaction, the client starts sending Binding indication messages to the server address to keep the binding alive. The recommended frequency for Binding indications is 15 s. The binding on the NAT breaks after a while, if no messages are sent using the binding. The server does not need to be informed of the unbinding.

Table 2.2: STUN attributes

Attribute type	Description
(XOR-) MAPPED-ADDRESS	Indicates a reflexive transport address
USERNAME	Identifies the username and password combination
MESSAGE-INTEGRITY	Contains an HMAC-SHA1 of the message
ERROR CODE	Used in Error response messages
REALM	Used with long-term credentials
NONCE	A random value used for replay protection
FINGERPRINT	Used for distinguishing STUN messages

When receiving a message, a STUN agent first needs to confirm that the message in question is indeed a STUN message before further processing. If the message is a response message, the agent needs to check that the transaction ID of the received message matches one of the outstanding transactions. Some other checking might be necessary, such as checking the FINGERPRINT extension discussed below. Additionally, possible authentication mechanisms are checked if specified by the usage. In case of an error specified by the STUN protocol, an error response is sent. The type of error is announced in the ERROR-CODE attribute.

The FINGERPRINT mechanism is an additional mechanism for distinguishing STUN messages from other messages. It is needed when multiplexing multiple protocols using the same transport address. When utilized, an agent adds the FINGERPRINT attribute to the messages it sends. STUN provides also authentication and message-integrity mechanisms, known as the short-term credential mechanism and the long-term credential mechanism. When using short-term credentials, another protocol is needed for exchanging a username and a password. The username and the password are applicable only for the duration of the media sessions, which makes the credentials time-limited. When forming a request or an indication the USERNAME and MESSAGE-INTEGRITY attributes must be included, whereas a response message must only include the MESSAGE-INTEGRITY attribute. The message integrity is a value computed using the password exchanged earlier.

The long-term credentials rely on long-term username and password credentials. They are called long-term since they stay valid until a user is no longer a subscriber of the system or they are changed. Replay attacks are prevented by using realm and nonce values. Initially, the client sends a request without any credentials or integrity checks. The server rejects the request by providing the user a realm and a nonce. For the next request, the client adds the username, realm and the nonce it just received. Also a message-integrity is included providing an HMAC (Hash-based Message Authentication Code) over the entire request, including the nonce. The server checks the values, and if approved, the message is authenticated. If the nonce is no longer valid the whole procedure is repeated by rejecting the request and providing a new nonce. Indications are not protected by long-term credential. All responses sent by the server should include the message-integrity attribute computed using the username and password used to authenticate the request, but the username, realm, and nonce attributes should not be included.

2.3.2 TURN

There are situations in which a direct connection between hosts behind different NATs is impossible, especially if a NAT uses address or address and port dependent mapping [54]. In such cases, it is necessary to relay the communication via an external intermediate node. Traversal Using Relays around NAT (TURN) [43] is a protocol that provides a relaying service via a TURN server. The relaying service of TURN can be used as such to guarantee communication between peers or as part of ICE to make traversal of NATs more optimized – using relaying only as a last resort. TURN is an extension to the STUN protocol, that allows a client to allocate an address on the TURN server using a new method type called Allocate.

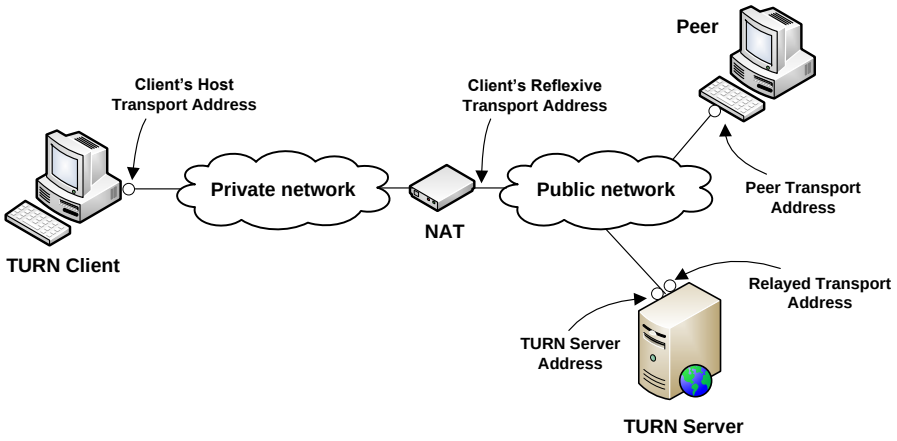

Figure 2.12: Example of a TURN configuration

Figure 2.12 shows a possible TURN configuration. The client sends an Allocate request to the server and the server replies with an Allocate response including a XOR-MAPPED-ADDRESS (see Section 2.3.1) and a XOR-RELAYED-ADDRESS attribute that contains a relayed transport address. The relayed transport address is a combination of IP address and port on the server that peers can use for contacting the client. The Allocate request and response messages can include additional attributes to describe the allocation. TURN clients send both STUN and data messages to the TURN server address. The TURN server sends the data messages forward to the clients' peers using the clients' corresponding relayed transport address. Another method type is called Refresh, and it can be used either to refresh an existing allocation, update the expiry time of an allocation, or to delete an allocation. All the new TURN related STUN message types (excluding error responses)

Table 2.3: STUN message types for TURN

Message type	Description
Allocate request	Makes an allocation on the server
Allocate response	Includes the reflexive and relayed transport address
Refresh request	Refreshes the allocation
Refresh response	Acknowledges the Refresh request
CreatePermission request	Permits a peer to send data for the client
CreatePermission response	Acknowledges the CreatePermission request
ChannelBind request	Requests to send ChannelData messages
ChannelBind response	Acknowledges the ChannelBind request

and attributes are collected in the Tables 2.3 and 2.4. Since a relayed transport address unambiguously defines a client's reflexive transport address and the used transport protocol, relaying the messages from the client's peers to the corresponding client is straightforward. However, the client must beforehand approve the peer, whose messages are to be forwarded. This is done by the client sending a CreatePermission requests to the server specifying the permitted IP addresses as XOR-PEER-ADDRESS attributes in the message. The lifetime of a permission is 300 s (= 5 min) and it can be refreshed by retransmitting the CreatePermission request.

When sending an Allocate request, the client must choose a currently unused transport address for sending the message to the server transport address. The only obligatory attribute to be added to the allocate request is the REQUESTED-TRANSPORT attribute that specifies the transport protocol to be used. When receiving an Allocate request the server checks the credentials, that the requested allocation does not already exist, and that the REQUESTED-TRANSPORT attribute is included. After this any additional attributes might be checked. If all checks are passed, an allocation is created. A response message sent from the server to the client includes, in case of a success response, the relayed transport address, lifetime of the allocation, unique identifier for the allocation on the server, and the server reflexive transport address. TURN itself does not make use of the server reflexive transport address; it is included for the convenience of other protocols, such as ICE.

The TURN specification recommends the use of long-term credentials for the authentication to avoid the need to negotiate new credentials for every session. Once the allocation is done, the allocation needs to be refreshed before the lifetime of the allocation expires. This

Table 2.4: STUN attributes for TURN

Attribute type	**Description**
XOR-RELAYED-ADDRESS	Indicates a relayed transport address
REQUESTED-TRANSPORT	Specifies the desired transport protocol
XOR-PEER-ADDRESS	Specifies the address and port of the peer
DATA	Consists of the application data
LIFETIME	Represents the lifetime of the allocation
CHANNEL-NUMBER	Indicates the number of the channel

is done by sending a Refresh request. The lifetime of an allocation can be requested in the Allocate request and it can be modified in the Refresh request. It is up to the server to decide whether the requested lifetime is approved, and inform the client about the actual lifetime in the response. When a client no longer wishes to use an allocation, it sends a Refresh request with a LIFETIME attribute of value 0. Sending and receiving of data through the relay does neither refresh an allocation nor renew a permission. The time-to-expiry is by default 600 s (10 min).

Data can be transmitted in one of two ways between the client and its peers through the TURN server. The first uses Send and Data indications, and the second uses channels. When using the first way data is sent using a Send indication STUN message including two attributes: the DATA attribute including the data and the XOR-PEER-ADDRESS attribute including the address of the peer. Data is encapsulated in a Send or Data indication only between the client and the server. When the Send indication is received by the server, data is extracted and forwarded in a UDP datagram to the peer. To the opposite direction, as the server receives a data message on a relayed transport address, it places its content into the DATA attribute of a Data indication, puts the peer's address into the XOR-PEER-ADDRESS attribute and forwards the message to the client.

Using channels produces less overhead. This results from using a new message format, known as ChannelData message, instead of the STUN message format. It uses a 4-byte header, that indicates the channel number, a value bound to a peer's address to work as a reference number. The binding is done by sending a ChannelBind request to the server, which includes an unbound channel number (CHANNEL-NUMBER attribute) and the transport address of the peer (XOR-PEER-ADDRESS attribute). After the binding is made, the client is able to send data to its peer by sending a ChannelData message to the server. Likewise, it is able to receive ChannelData messages on the channel from the server. Channel bindings

are always initiated by the client. All other messages, except for the ChannelData message, are STUN message formatted. Permissions are granted based on IP addresses and a permission for a peer to send data back to the client expires after 5 minutes. A permission can be refreshed by repeating the ChannelBind (or CreatePermission) transaction. A channel binding lasts for 10 minutes unless refreshed.

2.3.3 Interactive Connectivity Establishment

Interactive Connectivity Establishment (ICE) [42] is a protocol developed by the Internet Engineering Task Force's (IETF) Multiparty Multimedia Session Control Working Group (MMUSIC). It provides a complete NAT traversal solution by utilizing the functionalities of STUN and TURN. ICE was originally defined to be used by UDP-based multimedia communication protocols, but later has been extended to handle TCP transport protocols [39]. This work only focuses on ICE for UDP-based media. A host using ICE, known as an ICE agent, needs to know the address of a STUN or TURN server and to have a signaling path with the host it wishes to communicate with.

In the beginning of the ICE processing, an agent discovers its transport addresses, known as candidates, that are potential points for contacting it. An agent uses the mechanisms of TURN to learn its server reflexive transport address and relayed transport address. Each agent knows the transport addresses obtainable from its local interfaces. To establish sessions using an offer/answer mechanism, a signaling path is required between the peers. The signaling path is used for exchanging the candidates a peer has gathered, for example within SDP (Session Description Protocol) offers and answers. The candidates received from the other agent are known as peer candidates. An agent pairs up its own candidates and the candidates of the peer and arranges them in a priority order, forming a list of possibly working candidate pairs. Then, the agent starts performing connectivity checks, which are done by sending checks on each candidate pair in priority order and at the same time replying to received checks. Checks are done by STUN Binding request transactions. When a check succeeds on a candidate pair, the pair is considered valid for communication.

One of the agents is controlling agent and the other one is a controlled agent. The agent in controlling role nominates one of the valid pairs for sending and receiving of media. This is done by performing a check on the chosen pair with a USE-CANDIDATE attribute added to the STUN request message. There are two approaches on nominating candidate pairs: regular nomination and aggressive nomination. In aggressive nomination, the USE-CANDIDATE attribute is added to each check and the first check to produce a reply is

chosen for media. In regular nomination, the controlling agent nominates a valid pair in the valid list as soon as it meets its stopping criteria. The stopping criteria is entirely up to the controlling agent's local optimization. As the controlled pair receives a STUN message including a USE-CANDIDATE attribute, it knows to use the pair in question for the media.

ICE can be implemented either as a lite or full implementation. A lite implementation does not gather candidates, since it only uses the addresses from local interfaces. In addition to this, it does not generate connectivity checks. It simple responses to checks. A lite agent never takes the role of a controlling agent. The ICE lite version is implemented on agents that are directly connected to the public network, but want to support ICE. A full implementation, in addition to generating the checks, also implements some additional procedures, such as detecting and repairing of role conflicts and discovering new peer reflexive candidates. A peer reflexive candidate identifies a peer as seen by the other peer. A full implementation uses triggered checks to speed up the ICE processing: if an agent receives a check on a pair that is not in succeeded state, it schedules a connectivity check on that pair.

Sometimes, a single ICE negotiation is used to establish connectivity for more than one transport layer port on a single transport address, for example, one for an audio and one for a video stream. Candidates and peer candidates for different media streams (i.e., components) are identified by different component IDs and only candidates and peer candidates with the same component ID are paired. To optimize the checks, in such a case, ICE implements a frozen algorithm based on the "similarity" of certain pairs. If two candidates are similar, they are said to have the same foundation – same transport address type and obtained from the same host candidate and STUN server, using the same protocol. The foundation of a candidate pair is simply the concatenation of the foundations of its candidates. The assumption behind the frozen algorithm is that checks performed on pairs with the same foundation value will end up with the same result for the checks. This makes it possible to optimize the order in which connectivity checks are performed despite of the priority order of the pairs.

Example of ICE message flow

The message sequence diagram in Figure 2.13 shows an example message flow of an ICE session. The figure outlines the messages exchanged during the connection establishment phase and after the connection has been set up. The ICE agents are located in different private networks, however, they both use the same TURN server in the public network.

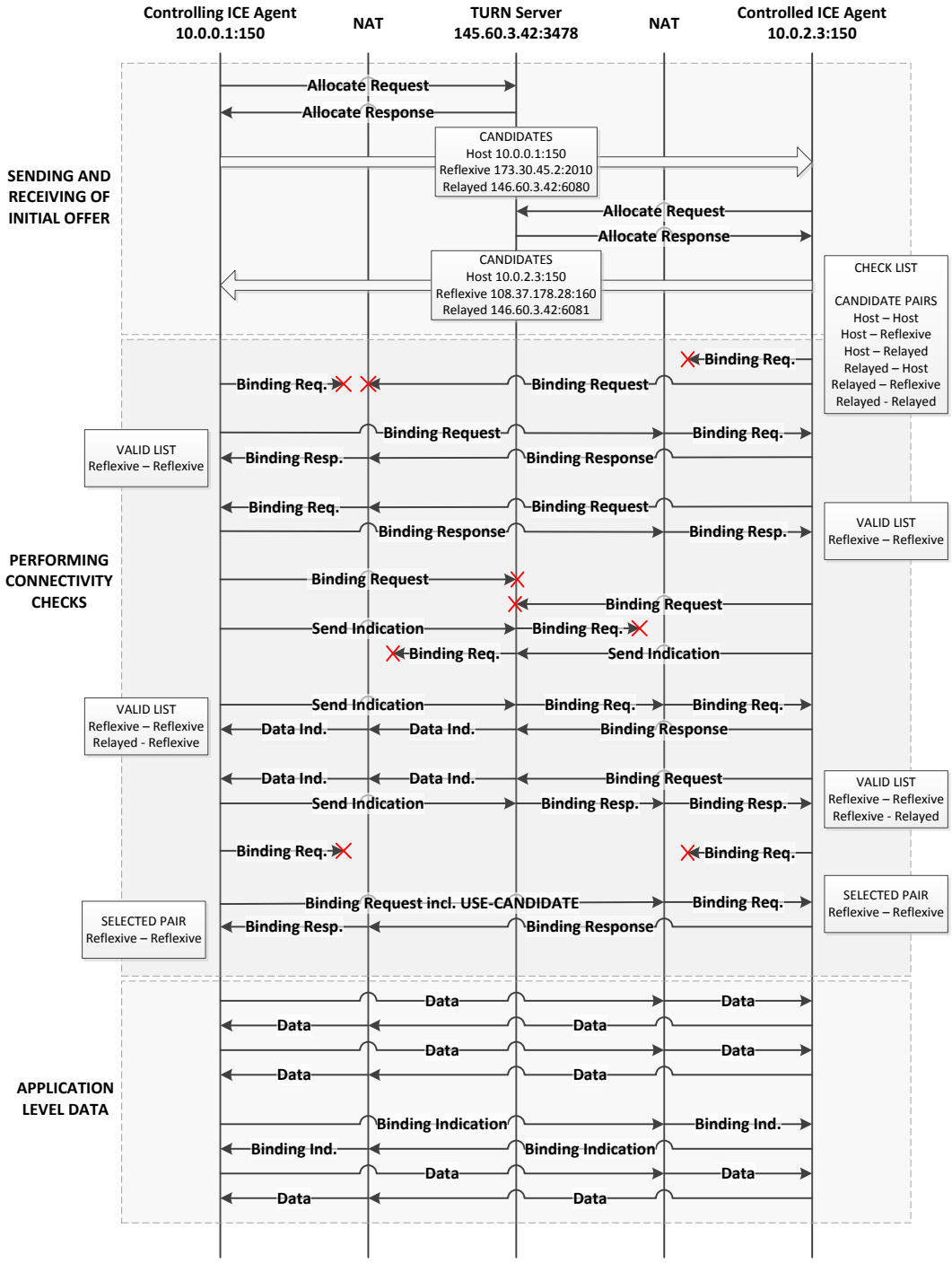

Figure 2.13: Message exchange during an example ICE session

The agent on the left is in controlling role, so it first gathers its candidates and sends the initial offer. As the agent in controlled role receives the initial offer it likewise performs the candidate gathering procedure and sends its answer to the controlling agent. After the candidate exchange is done, ICE agents start performing the actual checks according to their check list. As the figure shows, the controlling agent might continue performing the checks as usual even after a candidate pair is added to the valid list. The valid list consists of candidate pairs that are considered valid for communication. An agent usually wants to make sure that none of the higher priority candidate pairs work before meeting its stopping criterion. An agent will ensure this via retransmission of yet unanswered Binding requests.

In this example, the checks to the reflexive addresses either directly or via the TURN server work as long as permissions on the TURN server exist. On the other hand, transmissions to the host addresses will not work since the agents belong to different private networks. As soon as the stopping criterion is met, the highest priority valid pair gets nominated by sending a Binding request with an USE-CANDIDATE attribute. After this the agents can start sending data using the selected pair, which in this case means using the reflexive addresses as transmission endpoints. Messages need to be sent regularly to keep the binding on the NAT alive; Binding indications are used if no data has been sent for a while.

2.4 Existence of Different NAT Types

Defining the type of an existing NAT is not straightforward due to the non-deterministic nature of some NATs. Non-deterministic nature means that a single NAT might behave differently depending on the prevailing conditions. For example, this is the case with some of port preserving NATs. As a host tries to use a port that is already in use on the NAT, it gets a different port assigned to it, and additionally ends up with a different mapping and/or filtering behavior. A NAT that does not change its mapping or filtering behavior under any circumstances, is referred to as a deterministic NAT. A NAT that can change

Table 2.5: Mapping and filtering behavior of existing NATs

	Endpoint Independent	**Address Dependent**	**Address and Port Dependent**
Mapping	92.31%	-	7.69%
Filtering	36.54%	4.81%	58.65%

its behavior, a non-deterministic NAT, is said to have a primary and secondary type. The primary type refers to the behavior in the absence of conflicts, and secondary type to the changed behavior after a conflict. The most recent results, in 2008, on existing NAT types are based on a field test [36] including 104 NATs. The test did not take into consideration the non-deterministic behavior of NATs. So, we can not be sure whether the behavior from a reported data point is based on the primary or secondary type of the NAT. The results on the mapping and filtering behavior of existing NATs based are shown in Table 2.5.

Table 2.6: Probabilities for NAT types between two random hosts

		Host 2				
		Public	**EIm/EIf**	**EIm/ADf**	**EIm/PDf**	**PDm/PDf**
Host 1	**Public**	4.00%	5.85%	0.77%	8.15%	1.23%
	EIm/EIf	5.85%	8.55%	1.12%	11.92%	1.80%
	EIm/ADf	0.77%	1.12%	0.15%	1.57%	**0.24%**
	EIm/PDf	8.15%	11.92%	1.57%	16.62%	**2.51%**
	PDm/PDf	1.23%	1.80%	**0.24%**	**2.51%**	**0.38%**

Not all hosts are behind NATs – there are also hosts directly (or with the assistance of a UPnP) connected to the Internet. Yet the percentage of directly accessible hosts is quite small; around 20% [7]. We can also calculate the probability for relaying in the Internet, when we assume that 80% of all hosts are behind NATs and use the NAT type frequencies from the field test. Table 2.6 presents the probabilities for the NAT types between two arbitrary hosts; for example, the likelihood for both peers to be behind symmetric NATs is 0.38%. We use the following abbreviations in the table: EI for endpoint-independent, AD for address-dependent, PD for address and port-dependent, m for mapping and f for filtering.

The percentage values in the table marked in bold represent the combination of NAT types between two hosts that cannot create a direct path but instead need a relay. The NAT hole punching support is not directly a property of a NAT. Whether hole punching works, rather depends on the context. To be more precise, it depends on the interoperability of a NAT with another NAT. Relaying is needed if at least one of the peers is behind a NAT that uses address and port-dependent mapping and filtering, and neither of the peers is behind an endpoint-independent mapping and filtering NAT (or directly connected to the public network). Based on the frequency of different NAT combinations, one can calculate (by summing the cells printed in bold) the probability for relaying between two random hosts to

be 5.88%. This value is slightly smaller than the corresponding estimates made on Skype's traffic (9.6%) [21] and GoogleTalk traffic (8%) [19].

2.5 Peer-to-Peer Networking

In peer-to-peer networking each peer has the same capabilities allowing them to share resources equally. This basically means that a peer has characteristics of both servers and clients. One major challenge in peer-to-peer networks is in locating the resources. Using a decentralized architecture has several advantages, such as improved scalability, greater fault tolerance and self-organization. Moreover, peer-to-peer communication gives opportunities for creating new user scenarios.

At the moment the probably most widespread P2P application is file sharing, in particular the sharing of music files. Among the popular P2P applications is also Skype [47], which is a free P2P application that enables calls over the Internet to other Skype users. In peer-to-peer communication, the applications need to be interoperable and the peers need to be able to trust each other. [52]

2.5.1 Peer-to-Peer Session Initiation Protocol

The Session Initiation Protocol (SIP) is a text-based signaling protocol for creating, modifying and terminating multimedia communication sessions between peers. SIP runs on the application layer, independent of the underlying transport protocol, such as TCP or UDP. The traditional SIP relies on hierarchical proxy servers that help routing SIP messages between the SIP clients. The proxy servers are also used for uploading and requesting a user's current location (i.e., contact address), as well as authentication and authorization. Example application scenarios for P2P (or P2PSIP) are related to real-time IP communications, such as VoIP, Instant Messaging (IM), and presence. [45, 9]

In Peer-to-Peer Session Initiation Protocol (P2PSIP), the hierarchically organized proxy servers are replaced by a P2PSIP overlay of nodes called P2PSIP peers. A P2PSIP overlay is a collection of nodes running a distributed database algorithm, such as a distributed hash table (DHT). The main responsibility of the P2PSIP peers is to provide a transport service and a storage service. The transport service makes it possible to transport messages between any two peers in the overlay. The storage service means storing information for the purpose of mapping between Address-of-Records (AoRs) and Contact Uniform Resource

Identifiers (URIs). The overlay may have other type of nodes, known as P2PSIP clients, which make use of only a logical subset of the P2PSIP peer protocol, known as the P2PSIP client protocol. The peer protocol and the client protocol are both implemented using RESource LOcation And Discovery (RELOAD), discussed in more detail under section 2.5.4. The elements of a P2PSIP overlay are shown in the Figure 2.14. [8]

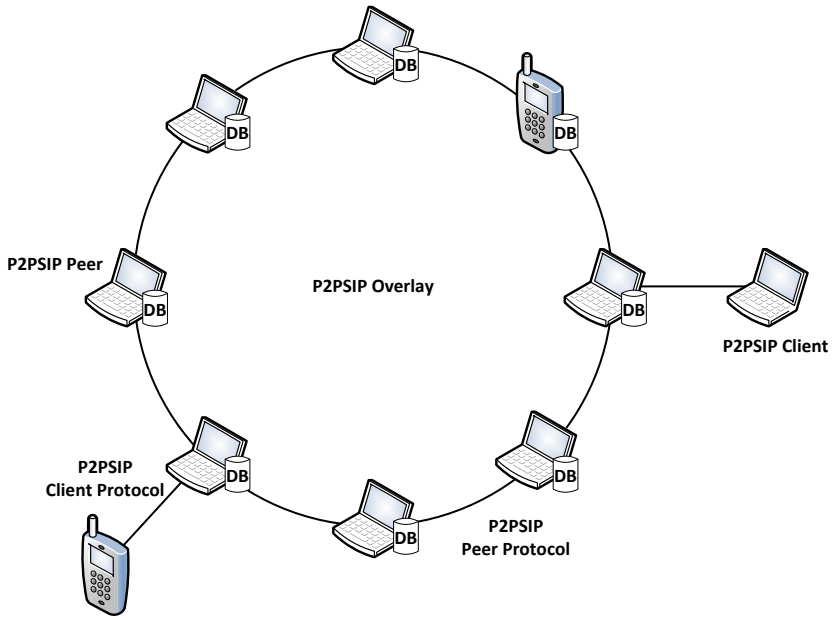

Figure 2.14: Elements of a P2PSIP Overlay

Since peers from different address spaces are allowed to join the overlay, NAT traversal is necessary for peer protocol and SIP (and media) connections. RELOAD mandates the use of ICE for NAT traversal. RELOAD uses a "mesh of connections" approach to NAT traversal. In the approach, a partial mesh of connections is created between the peers and messages are routed according to the existing mesh topology. So even peers with private addresses can be full peers in the overlay. [26]

2.5.2 Use of Distributed Hash Tables

The lack of centralized control in a peer-to-peer network makes it difficult to efficiently locate a peer with a particular data item. Several distributed lookup protocols have been implemented to address the problem. Many of them make use of a lookup service based on distributed Hash Tables (DHTs) containing $(key, value)$ pairs.

Hash functions are used to convert a variable-size input into a fixed-size value which is called the hash value. Hash tables make use of hash functions to map keys to table entry indexes to retrieve an associated value. Hash tables enable fast lookup since there is no need to compare all values in sequential order in a table but rather use a hash function that directly points the corresponding table entry.

Due to the decentralized architecture of peer-to-peer networks, the hash table cannot be stored in a single node, instead it has to be somewhat equally distributed among the peers. Each peer is given responsibility of a certain set of keys. A DHT algorithm can be used for distributing the information. DHT scales well to networks with lots of peers and can take care of peers joining and leaving the network. We take a more detailed look at one of the DHT algorithms, Chord, since it is the default algorithm used by the RELOAD protocol. Several other algorithms exists, such as CAN [41], Pastry [46] and Kademlia [38].

Chord

Chord [51] is a DHT algorithm that uses consistent hashing to arrange nodes in a ring and to map keys to nodes responsible for them. Nodes are placed in the ring based on the hashed value of the node's key (e.g., peer's IP address) calculated using a hash function, such as Secure Hash Algorithm (SHA-1) [1].

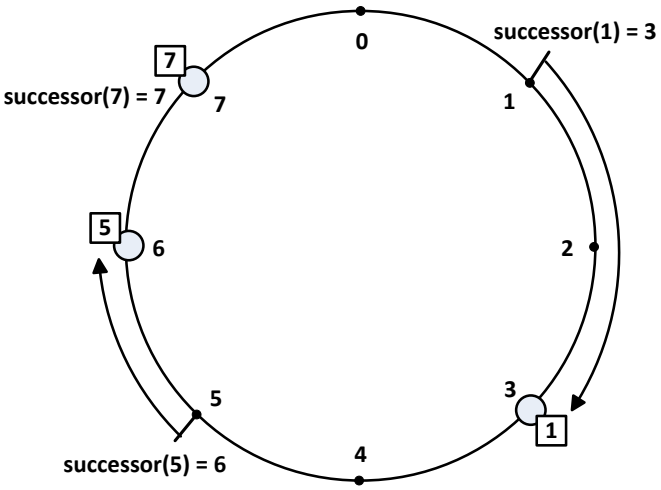

Figure 2.15: An identifier circle with three nodes

In RELOAD, the key is assigned by the an enrollment server or if there is no central server, it is calculated as hash of the node's public key. Data localization can be implemented by associating a key with a data item. One of Chord's advantages is its simplicity: mapping a given key to a node is the only function that it supports. It is efficient in locating each key with a small number of hops ($O(\log(N))$), even in networks with a large number of peers.

The Chord protocol can also recover from the failure of existing nodes. However, if information is out of date, performance decreases. Each peer maintains information about $O(\log(N))$ other peers in its routing table. Figure 2.15 shows how nodes get arranged in an identifier circle based on their hashed values, called identifiers. Each identifier has a predecessor and a successor. $Successor(k)$ denotes the successor node of key k, the node responsible for the key. It is either the first node whose identifier is equal to the key or the next node in the identifier circle when moving clockwise in the ring. Whenever a node joins or leaves the network, the predecessor and successor states of the adjacent nodes need to be updated.

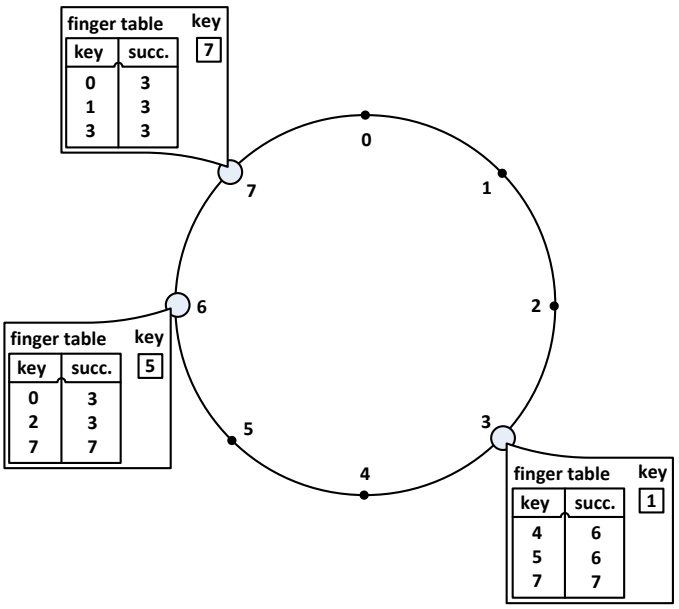

Figure 2.16: An identifier circle with nodes maintaining finger tables

However, knowing only the successor node would lead to a very insufficient solution since queries would need to be routed via all the intermediate nodes in the ring. To speed up the process, additional routing information is needed. The additional routing information is implemented by maintaining a routing table known as finger table. A finger table entry

contains the Chord identifier and the IP address of the corresponding node. The entries are calculated using the formula $finger[i] = successor(n + 2i - 1)$, which returns the ith entry in the table at node n. Figure 2.16 shows the finger tables of the nodes originally shown in Figure 2.15. As the figure shows, each node knows more about nodes closely following it than about nodes further away in the circle. The Chord protocol uses a stabilization protocol to take care of the successor pointer and finger table updates.

2.5.3 Peer-to-Peer Protocol (P2PP)

The Peer-to-Peer Protocol (P2PP) [6] is an application layer protocol for forming and maintaining a P2P overlay. The participating peers provide routing and storage services to be utilized by the P2P peers themselves or by P2P clients through the peers. P2PP can use either a structured (e.g., Chord) or unstructured (e.g., Gia [13]) overlay algorithm, for creating the overlay. The data that is stored in the overlay is referred to as a resource-object. Each resource-object has a resource-ID that can be used for locating the peer responsible for the resource. To assure that the peer and user identifiers (peer-IDs and user-IDs) are unique, a central enrollment server can be used [26]. User identifiers can be for instance SIP URIs (e.g., alice@work.com).

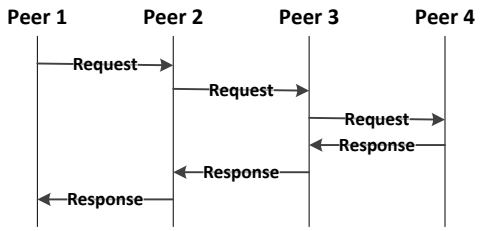

Figure 2.17: Recursive routing

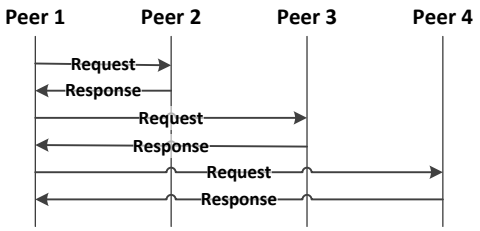

Figure 2.18: Iterative routing

The overlay layer below the application layer takes, among other functions, care of routing the messages and provides mechanisms for NAT traversal. In a P2PP overlay, request messages are forwarded either in a recursive or iterative manner. Figure 2.17 shows how recursive request routing is done in comparison to iterative routing, shown in Figure 2.18. In recursive routing, the request is forwarded from one peer to another towards the destination. As the request finally reaches its destination, the destination sends a response back along the same route. In iterative routing, every intermediate peer sends a response back to the sender telling the next hop peer. Thus, the response from the destination is sent back

to the sender using a direct connection. On the transport layer, messages can be sent using reliable or unreliable transmission.

2.5.4 REsource LOcation And Discovery (RELOAD)

REsource LOcation And Discovery (RELOAD) [26] is a peer-to-peer signaling protocol that allows nodes to route messages to each other and to store and retrieve data in the overlay. The RELOAD protocol was chosen among several competing protocols to be developed by the Internet Engineering Task Force's (IETF) Peer-to-Peer Session Initiation Protocol Working Group (P2PSIP). The current version of the RELOAD Internet-Draft is based on the combination of some of the initially competing proposals: RELOAD, Address Settlement by Peer-to-Peer (ASP) [27], and Peer-to-Peer Protocol (P2PP).

Even though RELOAD has been especially designed to fulfill the requirements for P2PSIP, it can be used to support various different applications. RELOAD makes use of pluggable overlay algorithms. To increase interoperability, the Chord DHT algorithm is mandatory to implement. Otherwise it is possible to select an overlay algorithm that is best optimized for a certain application. RELOAD also works in networks were some of the peers are behind NATs or firewalls. ICE is the protocol used by RELOAD for traversing NATs as new connections need to be established between peers either to be utilized by RELOAD or the application protocol.

The P2PP protocol discussed in the previous Section 2.5.3 and the RELOAD protocol are very alike, since they both use a very similar approach to solving the same problem. Their probably biggest difference is in the way they encode the messages they send. P2PP uses a common header followed by a variable size payload of type-length-value (TLV) objects. RELOAD, on the other hand, uses a more complex message structure that consists of three parts: forwarding header, message contents and signature. Moreover, in P2PP intermediate nodes store information to route a response message from the destination back to the sender, whereas RELOAD has an additional option to implement this by adding via and destination lists to the forwarding header. This option reduces the state information needed to be stored into nodes. However, it increases the size of the messages sent.

RELOAD Architecture

As shown in Figure 2.19 the architecture of RELOAD can be depicted by dividing the main components of RELOAD into different layers: usage layer, routing layer, forwarding layer

and transport layer. The figure also shows the APIs (Application Protocol Interfaces) that can be used by the upper layers to access services provided by the lower layers.

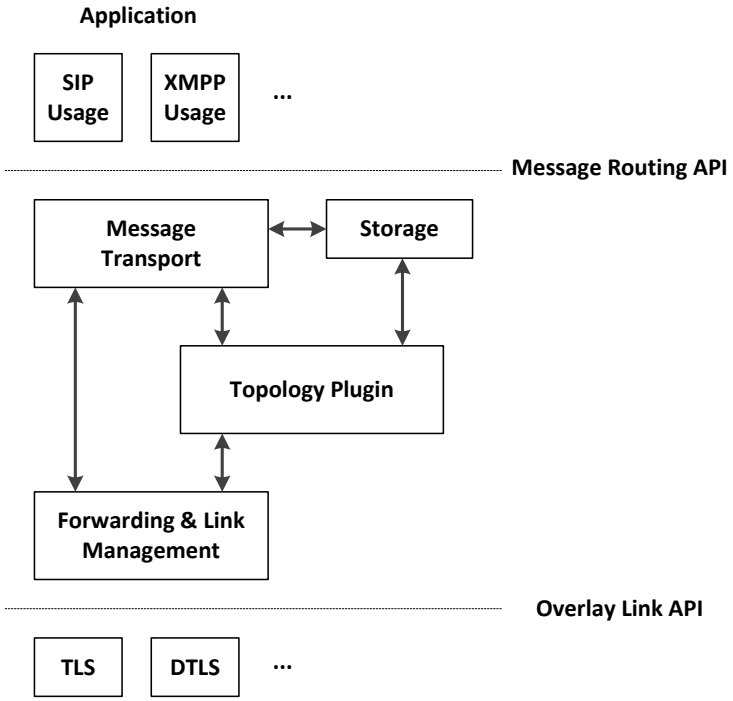

Figure 2.19: The main components of RELOAD

It depends on the application how it wants to use the services provided by RELOAD. Because of this, each application needs to define a usage layer that communicates with RELOAD through the Message Routing API. One example of RELOAD usage is SIP usage. The main responsibility of a usage is to specify how data is handled in the overlay and how applications can get the data.

The routing layer is composed of three components: the message transport, the storage component and the topology plugin. All these components talk directly with each other. The routing layer takes care of routing messages from one peer to another through the overlay. The storage component handles storage and retrieval requests from the routing layer as well as manages data replication if required by the topology plugin. The topology plugin maintains a routing table that is formed based on the chosen overlay algorithm. The routing layer makes its routing decisions based on the overlay algorithm routing table maintained by the topology plugin.

The forwarding layer is below the routing layer and it talks directly with the routing layer and the topology plugin. The forwarding layer sets up the network connections as determined by the topology plugin and makes sure that a packet gets forwarded to the next peer as determined by the routing layer. The forwarding layer takes also care of NAT traversal. The transport layer below the forwarding layer takes care of actually carrying the RELOAD messages. The forwarding layer communicates with the transport layer through the transport API. The RELOAD draft specifies how to use DTLS (Datagram Transport Layer Security) and TLS (Transport Layer Security) on the transport layer.

Establishing and Managing the RELOAD Overlay

For the RELOAD overlay to be able to provide services, nodes need to be able to join the network, create connections with each other, and route messages through the overlay. Each node in the overlay has an identifier, known as the Node-ID. The Node-ID determines the node's position in the overlay topology and the resources for which the node is responsible. A node must have a Node-ID before joining a RELOAD overlay, either by receiving it from an enrollment server or by generating it by itself in a trusted network.

To become part of an overlay a node must first create a connection to a "bootstrap node" that is publicly reachable. The overlay algorithm is responsible for taking care of the the details of the joining procedure, like creating connections to some other peers and transferring resources between the joined node and its adjacent nodes. As soon as the node is part of the overlay, it can send Attach requests to be routed through the overlay to other nodes it wishes to contact. The overlay routing mechanisms take care of forwarding the message to its destination. Naturally, messages can be lost on the way. To provide end-to-end reliability, RELOAD sends up to 4 retransmissions if no response is received in 3 seconds after the latest request has been transmitted.

To help routing the messages the forwarding header can include via and destination lists. The via-list contains all the destination nodes that the message has passed and the destination-list contains all the destinations that the message should pass through. As P2PP, RELOAD supports two routing mechanisms recursive and iterative routing. RELOAD uses symmetric recursive routing as its basic routing mechanism: a request message is always forwarded closer to its destination and the response follows the same path. This can be implemented using via-lists or storing state in the intermediate nodes on how to return the response. RELOAD supports iterative routing due to its advantages, such as the possibility to route around misbehaving NATs and to cause less processing in intermediate nodes. However,

when peers are located behind NATs it becomes very resource consuming since every hop requires a direct connection to be established.

Direct connections are formed either for the purposes of RELOAD messages or application layer protocol to improve the efficiency of routing. Since a direct connection may not be created off hand in the presence of NATs, Attach messages are used for establishing the connection. In contrast to the way ICE is used with SIP, the ICE parameters are exchanged using a more restricted binary encoding in the Attach method, instead of SDP. There are also other restrictions concerning the RELOAD environment: only one single stream (for RELOAD or application protocol) is supported and only a single offer/answer exchange is needed. The peer sending the initial Attach request is always in controlling role. Every peer is capable of working as a STUN server, however, each peer may need more that one STUN server due to complicated NAT topologies.

2.6 Mobile Phone Capabilities

Even though the capabilities of a mobile phone are improving, they still have their limitations in comparison to computers. The most significant limitations to consider from the point of view of this thesis are the battery life-time and the processing efficiency. On the other hand, the storage capacity is nowadays not considered to be such a limiting factor, since the memory size in a single mobile phone has increased substantially, even up to several gigabytes.

2.7 Summary

Network Address Translators (NAT) take care of address translation between different address realms. By means of NATs it is possible to separate private networks from the public network and reuse the private addresses in different private networks. Even though NATs have their benefits, such as slowing down the IP address depletion and providing privacy, they have become a challenge for direct communication between hosts.

The way to make peer-to-peer communication work in the presence of NATs is by utilizing NAT traversal techniques, such as Interactive Connectivity Establishment (ICE). ICE tries to find the optimal path for communication between two peers. A host using ICE needs to know the address of a Session Traversal Utilities for NAT (STUN) or Traversal Using

Relays around NAT (TURN) server and to have a signaling path with the host it wishes to communicate with. A host can use the STUN server to discover its globally routable address and the TURN server helps relaying data messages as a last resort. According to the literature 80% of hosts are estimated to be behind NATs. Thanks to the capabilities of ICE, such as the ability to learn peer reflexive addresses, the probability that relaying is needed between two arbitrary hosts is around 6%.

Peer-to-Peer Session Initiation Protocol (P2PSIP) can be used for forming, modifying and terminating sessions between peers in a peer-to-peer environment. Application layer protocols, such as SIP, make use of a transport and storage service for transporting SIP messages and storing information. Such a service can be implemented by a peer-to-peer signaling protocol like REsource LOcation And Discovery (RELOAD) or its predecessor, Peer-to-Peer Protocol (P2PP). However, the lack of centralized control makes it difficult to efficiently locate a peer with a particular data item. Distributed Hash Table (DHT) algorithms that use consistent hashing can map keys to nodes responsible for them. One example of a DHT algorithm is Chord, which arranges peers in a ring topology to achieve effective mapping.

Chapter 3

Implementing Mobile NAT Traversal Using ICE

This chapter starts with some motivation for implementing NAT traversal for mobile phones. We also introduce the programming language used for the implementation. The architecture of the implemented ICE prototype is shown in more detail, including a class diagram and a description of the main tasks of each class. The differences between the implementation and specifications of STUN, TURN, and ICE are listed. Finally, a few words are mentioned on the design decision made regarding the stopping criterion of the ICE connectivity checks.

3.1 Need for Mobile NAT Traversal

The need for NAT traversal becomes apparent from the increasing amount of P2P applications available. As mentioned earlier, it is difficult for two nodes behind different NATs to contact each other directly, which is essential to applications such as Voice over IP (VoIP) and online gaming. Peer-to-peer communication does not use servers to store the content, however, servers are still sometimes necessary for implementing the NAT traversal functionality [49]. Now that the peer-to-peer applications are moving also to the mobile side, it is worth measuring how well NAT traversal functionality, such as ICE, works on mobile phones. We need to consider the client side as well as the server side of a NAT traversal protocol. Of course, the mobile phones implementing STUN or TURN servers need to be publicly reachable. In that case, data would be transferred directly from mobile phone to another or relayed via a third mobile phone. This entirely eliminates the need for a non-

mobile server. It would have been nice to have some statistics on the proportion of private addresses in mobile networks. Unfortunately, we were not able to find any with a reasonable amount of effort. This is why we simply have to depend on the results presented in Section 2.4 for wired networks on the existence of different NAT types.

3.2 Java 2 Micro Edition

The ICE library was implemented using the Java 2 Micro Edition (J2ME), which is a version of Java designed for small devices. J2ME takes into account the resource-constrains of a small device, such as a mobile phone, by providing only a subset of the Java Standard Edition. To ease the efforts to run code during implementation, it is possible to use a mobile phone emulator on a PC and transfer the code later to real phones. [29]

J2ME is divided into configurations, profiles and optional APIs. A configuration defines the core APIs for certain type of devices. A profile provides more specific APIs based on the configuration and optional APIs provide additional APIs, such as the Mobile Media API [14]. A mobile phone uses the Mobile Information Device Profile (MIDP) based on the Connected Limited Device Configuration (CLDC). [29]

The use of optional APIs was not necessary for the implemented ICE prototype. As a result of the limited libraries in the MID Profile, some minor modifications from the ICE specification were necessary. The differences from the specification are described in more detail in Section 3.4.1.

3.3 Implementation Architecture

We implemented an ICE prototype to test the suitability of the ICE protocol in a mobile peer-to-peer environment. The ICE library consists of 13 classes and approximately 7000 lines of code. The classes and their dependencies are depicted in Figure 3.1 using an Unified Modeling Language (UML) class diagram. The figure shows how the STUN library and its TURN extension form the basis of the ICE library. Three additional classes were needed for implementing the ICE specific functionality. The STUN library and the STUN library with the TURN extension can be used solely without the ICE library.

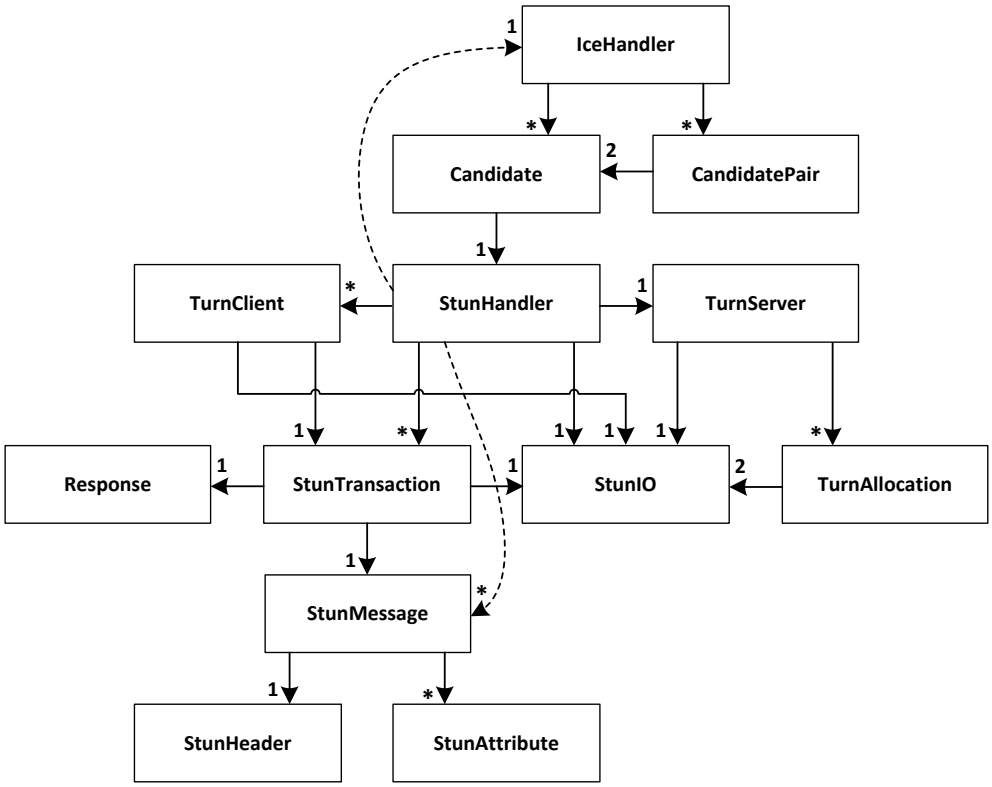

Figure 3.1: ICE implementation architecture

3.3.1 STUN Library

The StunIO class provides simple methods for sending and receiving of messages, both STUN and data messages. Each StunIO object receives and sends messages on a UDP Datagram Connection bound to a local endpoint address [35]. If STUN or data message needs to be encapsulated in a Send indication message, the StunIO object takes care of that.

The StunMessage class has methods for parsing and constructing a STUN message. The StunHeader class describes the header part of the STUN message and the StunAttribute class describes a single type-length-value (TLV) encoded attribute of the payload. A STUN message can be composed of zero or more STUN attributes. StunMessage also provides a method to examine whether the received message was a STUN message or not.

The StunTransaction class takes care of request/response transactions, including retransmissions of the requests and the scheduling of keep alive messages (Binding indications).

The information of a response is condensed into the Response class. A request and its corresponding response are associated via the transaction ID in the STUN header.

Most of the STUN functionality and logic is implemented in the StunHandler class. An instance of the StunHandler class is always bound to a single local address and port. The StunHandler class decides which class further processes a received message: STUN Binding requests and Data indications are handled by the class itself, whereas Allocate requests and Send indications it gives for the StunServer class to handle. In addition to this, the responsibility of the StunHandler is to take care of the client side of STUN request/response transactions, both Binding and Allocate, either to server addresses or during the connectivity checks to its peer candidate addresses. As we can notice, there is some TURN specific functionality included also in the STUN library. This is done to avoid having same functionality in both STUN and TURN specific classes. Moreover, to avoid cross-referencing between STUN and TURN classes, Data indications are processed in the StunHandler class since the StunHandler class anyway decides how to handle the message encapsulated in the Data indication.

StunHandler also examines if the received STUN Binding Request message is part of the ICE processing and if so forwards it to the IceHandler for further processing: to detect and repair role conflicts, to learn peer reflexive remote candidates, to trigger checks and possible nominate a candidate pair. Nonetheless, if the other end point receives connectivity checks before having started its own connectivity checks, it needs to postpone the further processing of the request until the checks are started. A response is, however, sent back immediately.

3.3.2 TURN Extension

The implementation of the TURN functionality is divided mainly into two classes: TurnClient and TurnServer. As the names imply, TurnClient class takes care of the client procedures and TurnServer of the server procedures, respectively. It is important to notice that an endpoint can act as a server and a client at the same time.

An instance of the TurnClient class is created as soon as a success response to an Allocate request is received. The TurnClient instance is server-specific, meaning that a single endpoint is able to run multiple instances of the TurnClient class related to one StunHandler. TurnClient takes care of refreshing the allocation on the TURN server for a certain local address. The TurnServer class handles Allocate requests and Refresh requests. It keeps track of all the allocations it is responsible for and cleans up the ones that have expired. Its

essential functionality is relaying messages between the client and its peers. Send indications received from a client are given for the corresponding TurnAllocation class for further handling. The TurnAllocation class represents an allocation on the TURN server. Its task is to store allocation specific variables, such as the transport addresses and the expiry time of the allocation. It also has a reference to two StunIO instances: one for the server address and one for the relayed address of the given allocation. The TurnAllocation class keeps up a list of permissions of peers allowed to send messages through the relay. Once a message from a permitted peer is received on the relayed address, the TurnAllocation encapsulates it in a Data indication and sends it forward from the server address to the corresponding client's server reflexive address.

3.3.3 ICE Library

The ICE library is implemented by adding three more classes on top of the STUN library with TURN extension. The resulting libraries as a whole implement the ICE functionality. The IceHandler class takes care of the main ICE logic: the candidate gathering, the scheduling of the connectivity checks and stopping the checks once the chosen stopping criterion is met. The IceHandler also provides additional methods for detecting and repairing of role conflicts, learning peer reflexive remote candidates and triggering checks. Once the connection is set up the IceHandler makes sure that application layer data is sent using the selected candidate pair(s).

The Candidate class describes a candidate. The same class is used to represent both local and remote candidates. The CandidatePair class represents a candidate pair by keeping a reference to a local and a remote candidates forming the pair. A candidate knows the StunHandler that was used for getting the candidate in the first place. The StunHandler in question takes care of performing the connectivity checks for the candidate pairs that have the given candidate as their local candidate.

3.4 Implementing ICE

The implemented ICE prototype is based on the ICE specification. The implementation can be considered as a full implementation that uses regular nomination. However, it does not include all the features specified in the ICE specification. Only the most important features for measurement purposes and for future usage were implemented. The biggest shortcoming of the implementation compared to the specification are the security options and error

messages. Also, the usage of the J2ME programming language set some implementation limitations that needed to be taken into account. These are covered in the Section 3.4.2. The differences from the specification are given in more detail in the following subsection. In addition to the basic ICE functionality, the following ICE features were implemented:

- Frozen algorithm based on the foundations of the candidate pairs; it is used to optimize the ICE processing when multiple connections are created between the peers during the same connectivity establishment.
- ICE-CONTROLLING and ICE-CONTROLLED attribute for detecting and repairing of role conflicts; in the case of SIP it is possible for both peers to end up having the same role.
- Triggered check queue; triggered checks are prioritized before the ordinary checks.

3.4.1 Differences from the specification

This section lists the features of the ICE specification [42] that were not implemented in the ICE prototype. The fact that some of the features are lacking has no impact on the validity of the results. The following features were not implemented:

- Support for multiple media streams; only one check list per ICE processing was implemented.
- The FINGERPRINT attribute, that provides additional multiplexing reliability.
- The authentication of Binding Indication messages; the Binding Indications are sent without any attributes.
- Error Response messages; without Error Responses the sender of the Request is unable to determine the reason for a failed transaction.
- ChannelData messages; only Send and Data Indications are implemented.
- CreatePermission transaction; permissions are created by sending data to a peer.
- No two minutes wait after an expired allocation; a relayed address can be recycled immediately after it has been freed.
- Utilization of short-term and long-term credentials.

- Encoding and exchanging of Session Description Protocol (SDP) messages using offer/answer exchange; the upper layer protocols take care of the offer/answer exchange.
- Subsequent offer/answer Exchanges; the implementation provides no means for sending an updated offer.
- Changing the RTO and Ta values during ICE processing; the transaction timers use configurable values that stay the same for the entire ICE processing.

3.4.2 Non-Specification Additions

Due to the limited J2ME library, three additional attributes were needed for the functions to work according to the ICE specification. Self-created attributes naturally violate the interoperability with other implementations. However, there are no known better alternative ways to work around the problem. For our measurement purposes, this fix works fine, since all the test devices use the same implementation. But taken into consideration future usage, our implementation will require some changes to be compatible. The three additional attributes are called HOST-NAME-ADDRESS, PEER-HOST-NAME-ADDRESS and RELAYED-HOST-NAME-ADDRESS. The attributes are used as aid in situations where the use of attributes (XOR-)MAPPED-ADDRESS, PEER-ADDRESS and RELAYED-ADDRESS as defined in the ICE specification is not possible.

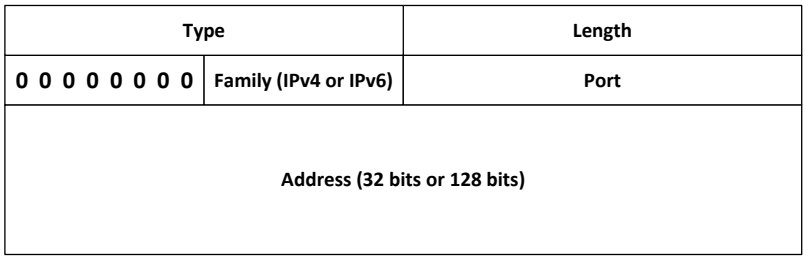

Figure 3.2: Format of STUN address attribute

The format of an address attribute is as shown in Figure 3.2 above. The format expects the address part to be an IP address of fixed length (either 32 or 128). The Java 2 Standard Edition (J2SE) provides methods for getting an address of a datagram socket or a datagram packet as either host name (if available) or IP address. This way it is possible for the programmer to always get the IP address instead of the host name. But this is not the

case in J2ME, where the IP address is most likely to be returned only if the host name is not available. This makes it impossible to encode the addresses given as host name in the format that the specification defines. This is the reason for defining new attributes that have the address as host name in ASCII format.

Another disadvantage as a consequence of the additional attributes is the increase in the amount of peer reflexive candidates. This results from the fact that transport addresses in IP address format and as host names are considered to be different transport addresses. As a consequence the amount of candidate pairs in the the check list grows. Additionally, the discovery of a new peer reflexive remote candidate causes a new check to be triggered for the newly created candidate pair.

3.4.3 Stopping the Connectivity Checks

The ICE specification only defines that a controlling agent should stop connectivity checks as soon as it meets its stopping criterion. However, an exact rule for the criterion is not defined to give the opportunity for local optimization. The stopping criterion is a compromise between the duration of the connectivity checks and the confidence on having selected the best working candidate pair for communication.

The stopping criterion chosen for our implementation is the following: The controlling agent nominates a highest priority candidate pair immediately after having received a success response to a check on that pair. In case that no success response is received to a highest priority candidate pair after 2 seconds wait, the highest priority non-relayed valid pair (if one exists) is nominated. A maximum of 10 seconds is waited, from the start of the checks, before nominating a valid pair that requires relaying. If the valid list is still empty at that point, the connectivity checks are said to have failed.

The decision for the 2 seconds wait is based on the recommendations by the International Telecommunication Union (ITU). ITU states that the mean post-selection delay on local connections under normal load should be on average 3 seconds [53]. Therefore due to the additional delays caused by the ICE processing, a 1 second margin is left for exchanging the candidates and for nominating the selected candidate pair(s).

The implemented stopping criterion takes into consideration the fact that certain pairs are more preferable, as well as bounds the maximum duration to an upper limit. The favorability is implemented by giving the higher priority candidate pairs more time to succeed in case retransmissions are required. The disadvantage of the criterion is that a total of 10

seconds needs to be waited every time that a relayed candidate pair is the only working pair. Additionally, the user experience substantially decreases if an operation takes a long time to complete without a response. If no progress indicators are shown, 10 seconds is about the limit for keeping the user's attention [37].

The same maximum duration for the connectivity checks is also used by other implementations [34]. Without an upper limit for the duration, in case of no working candidate pair, it might take a minimum of 40 seconds for the checks to conclude. This is the approximate time it takes for a candidate pair to use up all its retransmissions.

3.5 Summary

For the purpose of NAT traversal on mobile phones, an ICE prototype was implemented using Java 2 Micro Edition (J2ME). The ICE prototype consists of 13 classes and around 7000 lines of code. A STUN prototype with a TURN extension was first implemented since it forms the basis of the ICE prototype.

The idea of the ICE prototype is to provide the main ICE functionality. However, some features were implemented in addition to the basic functionality, such as the frozen algorithm used to optimize the ICE processing, and the triggered check queue used to prioritize certain checks. Some features of ICE were not implemented since they were seen as non-essential for the measurements performed for the thesis. The biggest difference of the implementation compared to the specification is the lack of security options and error messages. Additionally, some differences from the specification were unavoidable due to the limited J2ME library: three additional attributes were implemented for the functions to work somewhat according to the ICE specification.

The ICE specification does not specify an exact rule for when to stop the ICE checks. Our ICE prototype meets its stopping criterion if any of the following rules applies: (1) the highest priority candidate pair works, (2) a non-relayed candidate pair works after 2 second, or (3) the maximum time for connectivity checks (10 seconds) is exceeded. When the stopping criterion is met, the highest priority candidate pair in the valid list gets chosen for communication. If the valid list is still empty after 10 seconds, the ICE checks have failed.

Chapter 4

Measurements and Evaluation

In this Chapter, we describe the prototyping environment and show the measurements that were conducted to examine NAT traversal on mobile phones. We start this chapter by describing the P2PSIP prototype that was integrated to our ICE prototype for the purpose of testing the delays that NAT traversal causes in P2PSIP networks. Then we describe the prototyping environment, including the mobile and fixed access networks, as well as the different end devices. Before going to the actual measurements, we show some baseline measurements performed on the test mobile phone. Some of actual measurements were performed using the ICE prototype alone, such as the TURN and STUN client and server tests. Finally, we present the measurement results and finally evaluate the outcome.

4.1 P2PSIP Prototype

Before describing the prototyping environment in more detail, we need to take a look at the P2PSIP prototype, since it plays a major role in the call setup delay measurements. The prototype was not presented in the previous chapter, since we use an existing P2PSIP implementation [17] that consists of roughly 100000 lines of code for managing the P2PSIP overlay. In order for the P2PSIP prototype to work in the presence of NATs, it was integrated with the ICE prototype. The contribution of this thesis was to implement the ICE stack. The work required to integrate ICE to P2PP, SIP, RTP, and the Chord connection management logic was done by other developers.

The P2PSIP prototype is provided in two versions, one for PCs and one for mobile phones. Both are implemented in Java programming language, but they use different editions of

Java, J2SE (Java 2 Standard Edition) and J2ME, respectively. The two versions of the prototype are interoperable. Although we are mainly concentrating on the mobile implementation, it is interesting to make comparison as some of the tests are run on PCs as well. The J2SE implementation proved also useful in most of the mobile measurements in helping to implement those parts of the test setup that were not being examined.

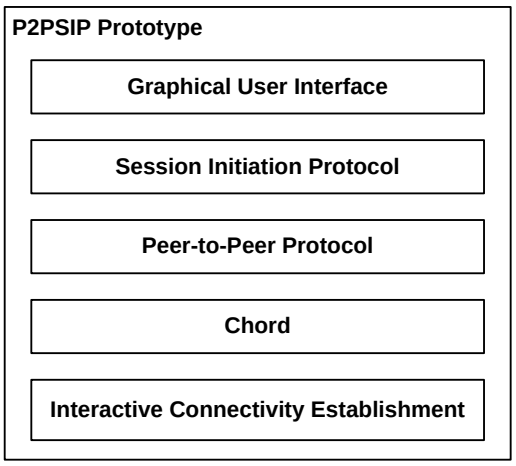

Figure 4.1: Architecture of the P2PSIP prototype

Figure 4.1 shows the architecture of the P2PSIP prototype after integration to the ICE library. All protocols run on top of the UDP transport protocol. The P2PSIP prototype provides a graphical user interface to make the prototype easier to use. The prototype uses SIP for controlling media communication sessions, such as voice calls. SIP uses the predecessor protocol of RELOAD, P2PP, for creating and maintaining the P2PSIP overlay and for mapping SIP Address-of-Record (AoR) values to contact Uniform Resource Identifiers (URIs). The DHT algorithm used by the prototype is Chord, since it is specified by the P2PSIP working group of the IETF as mandatory to implement. ICE takes care of NAT traversal by utilizing the functionalities of STUN and TURN. The P2PSIP prototype supports two types of nodes, P2PSIP peers and P2PSIP clients. The clients run only a subset of the P2P protocol stack, even though they implement the same stacks as the P2PSIP peers.

4.1.1 Call Setup between P2PSIP Clients

The call setup flow of P2PSIP is shown in Figure 4.2. Alice and Bob are both P2PSIP clients. They are connected to the overlay via P2PSIP peers, Peer 1 and Peer 3, respectively. They both use Peer 2 as their TURN server to simplify the figure; in reality Alice and Bob

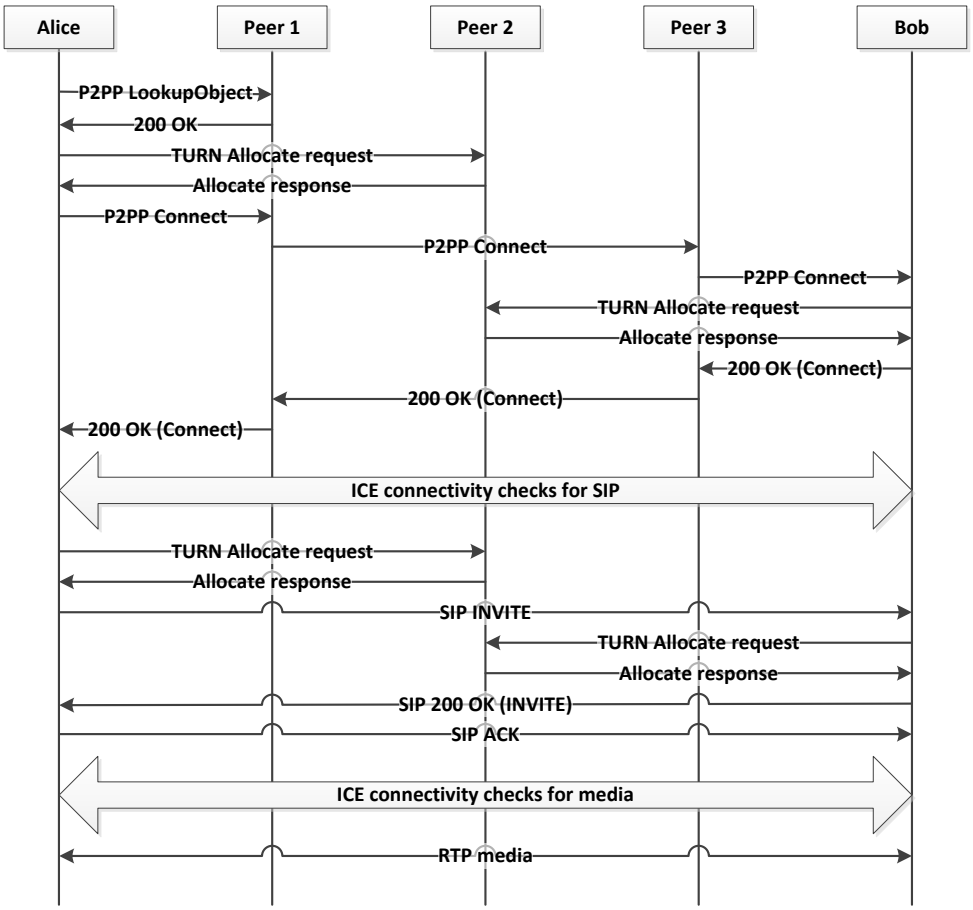

Figure 4.2: P2PSIP call setup

might be using different TURN servers. For the caller, Alice, to be able to contact the callee, Bob, she needs to discover Bob's contact information by sending a P2PP lookup into the overlay. The P2PP lookup may go through multiple overlay nodes before reaching its destination. After receiving a response to the lookup, Alice gathers her ICE candidates using her designated TURN server. For the purpose of creating a direct connection for SIP, a Connect request, including the candidates, is sent via the overlay. Next, Bob gathers his ICE candidates and sends them to Alice in a Connect response. After the candidate exchange, the connectivity checks are performed. If the checks succeed, Alice repeats the candidate gathering for RTP (Real-time Transport Protocol) and sends her candidates in an INVITE request to Bob, but this time using the connection just created for SIP. Once again Bob gathers his candidates and sends them in a SIP response to Alice. The SIP response

is acknowledged. Once the ICE checks for RTP have successfully finished, a call has been successfully established and RTP packets can be sent using the connection created by ICE.

4.1.2 Organizing Peers as STUN and TURN servers

Having more than one STUN server in a P2PSIP overlay is essential for each peer to be able to create direct connections even in complex NAT topologies. Since every peer in the overlay can act as a STUN server for another peer, STUN servers are learned when connecting to new peers. It is definitely not reasonable to use all the learned servers since the number of possible servers increases quickly as new connections are created. To maximize the chance of achieving a direct connection, a peer groups the servers based on equality of the peer-reflexive addresses it discovered through them. When new connections are being created, one server from each group is selected. However, not every peer is allowed to act as a TURN server. A peer can register itself as a TURN server only if the address of its local network interface matches to every peer-reflexive address it has discovered. [26]

Yet another interesting matter is how TURN servers are discovered in the overlay. The peers capable of acting as TURN servers need to save pointers in the overlay to ensure that other peers are with a high probability to locate them using a bounded number of lookups. The P2PSIP prototype calculates the number of pointers needed by using a birthday paradox based formula that takes as an input an estimate of the worst case TURN server density and an estimate of the total network size. [33]

4.2 Prototyping Environment

The measurements were carried out in either mobile or fixed access networks. The mobile access network was used when measuring the capabilities of a TURN server or STUN client in a mobile phone or when measuring the call setup delay between two mobile phones running the P2PSIP client protocol. The fixed access network, on the other hand, is used in measurements where the P2PSIP clients are run on PCs. In all P2PSIP measurements, the P2PSIP overlay is run over a fixed network. The following sections take a more detailed look at both of these access networks, including the devices used for accessing them.

The measurements were carried out on two mobile phone models: Sony Ericsson C905 and W910. Table 4.1 shows the specifications of these phones. The phones were connected to the network of the Finnish operator DNA in the Helsinki region. The phones reported

Table 4.1: Mobile phone specifications

	C905	**W910**
Memory	Up to 160 MB	Up to 40 MB
Battery	3.6 V, 930 mAh	3.6 V, 920 mAh
Standby Time	GSM: 400 hours, UMTS: 350 hours	GSM: 400 hours, UMTS: 350 hours
Talk Time	GSM: 9 hours UMTS: 3.5 hours	GSM: 9 hours, UMTS: 3.5 hours
Data transfer speed downlink	Up to 3.6 Mbps (HSDPA)	Up to 3.6 Mbps (HSDPA)
Data transfer speed uplink	Up to 384 kbps (WCDMA)	Up to 384 kbps (WCDMA)
J2ME	CLDC 1.1 / MIDP 2.1	CLDC 1.1 / MIDP 2.1
Java platform version	JP-8.4.3	JP-8.2.1

the received signal level to be at the maximum. We measured using network speed test [5] the downlink bandwidth to be 1037 kbit/s, when connected to the Internet through Third Generation (3G) High Speed Downlink Packet Access (HSDPA). The uplink bandwidth was 318 kbit/s. The measured bandwidths were averaged over several measurements at different times of a day. Although the uplink bandwidth was lower, it should not affect our measurements as shown later.

We used PCs also in the mobile measurements. The PCs acted either as TURN or STUN clients or servers not being the test subject. This was done to minimize the effects that the components not directly being experimented would have on the results. For example, if we were to measure the drop rate caused by relaying messages via a mobile phone acting

Table 4.2: Laptop specifications

	Dell Latitude D610
Processor	Intel Pentium M Processor 750 (1.86GHz, 2MB L2)
Memory	Up to 2048MB using 533MHz DDR2 SDRAM

as a TURN server between two wired PCs, one acting as a TURN client and the other its peer, we can assume that the possible packet drops are most likely to occur due to the communication over the air interface or the insufficient processing capacity of the phone. The Table 4.2 gives the specifications of the PC model used in the measurements. The PCs were connected to a corporate network in the Helsinki region.

A PC was also used to simply log test data from the mobile phone. The Sony Ericsson resource monitor was the application on the PC for logging the CPU load and memory consumption from the phone via an USB cable. JSR (Java Specification Request) Mobile Sensor API was utilized within the J2ME test code for measuring the battery consumption on a mobile phone.

4.2.1 P2PSIP Parameters

The call setup delay measurements were carried out in a global P2PSIP overlay consisting of approximately 500 peers. The same P2PSIP overlay has been used in previous work [30, 31, 32]. The overlay consisted of Planetlab [40] nodes running the P2PSIP prototype. Planetlab is an open platform that the members of the PlanetLab Consortium, mostly corporations and universities hosting PlanetLab nodes, can freely use for distributed systems research and network services development. The measurements were carried out by connecting two clients to the P2PSIP overlay and by initiating calls between them.

Table 4.3: P2PSIP traffic model and parameters

Parameter	**Value**
Peer interarrival time	28.8s
Network size	500 peers
Busy hour call attempts	2.21 calls/user/hour
% of calls to buddies	66.6
Finger pointers	10
Successors	10
Chord stabilization interval	60s

Table 4.3 describes the parameters that the P2PSIP overlay was using. The traffic model is similar to the one used in [32]. The arrival and departure of the peers in the P2PSIP overlay were modeled as a Poisson process. The average uptime of peers was 4 hours meaning that

the average interarrival and departure time was 28.8 seconds. Every user in the P2P overlay made on average 2.21 calls/hour and roughly 66.6% of the calls were made to buddies. When a call is made to a buddy no lookup is performed during the call setup. This is because a lookup to a buddy has already been performed before initiating a call in order to get the presence status of the buddy.

Since the successor or predecessor nodes in the overlay may disappear due to a failure or a departure, it is recommended for Chord to keep a table of multiple successors and predecessors. The sizes of Chord's successor and predecessor tables were set to 10 [32]. To speed up Chord's search method, finger tables of size 10 were used in the P2PSIP overlay. Moreover, the nodes run a stabilization process every 60 seconds to make sure that the routing table is up-to-date [51]. This information is also used to correct finger and successor table entries [51]. The stabilization interval should reflect the frequency of node arrivals and departures, i.e. the more frequently nodes arrive and departure, the more often the stabilization process should be run. The stabilization interval chosen for the measurements is based on the results in [31]. More detailed information on the P2PSIP prototype can be found in [30, 32].

The bootstrap peer and one of the P2PSIP clients of the overlay were located in Helsinki, whereas the other P2PSIP client was located in Hamburg, Germany. The Round-Trip Time (RTT) between the clients was measured to be on average 47.8 milliseconds.

4.2.2 ICE Parameters and Message Sizes

Table 4.4 shows the parameters that the ICE prototype was using. The stopping criterion for ICE was already presented in Section 3.4.3 since it was an implementation specific choice. However, there are ICE parameters that have to be configurable. One such parameter is the

Table 4.4: ICE parameters

Parameter	**Value**
ICE keepalive interval (Tr)	15s
Time between ICE checks (Ta), RTP	20ms
Time between ICE checks (Ta), non-RTP	500ms
Deadline, highest priority pair	2s
Maximum ICE check execution time	10s

Ta that defines the pacing of STUN or TURN transactions during the candidate gathering phase or the connectivity checks. The value of Ta is different if ICE is being used for establishing a connection for a real time media stream, such as RTP, or something else. For non-RTP sessions, such as SIP and P2PP in our case, Ta was set to 500 milliseconds, which is the default value specified in the ICE specification. The audio codec used by the P2PSIP clients was G.728, which means that 32 byte RTP packets were sent every 20 milliseconds. Thus, the Ta value for RTP was set to 20 milliseconds according to the formula in the ICE specification. [42]

Table 4.5: STUN message sizes

Method and class	**Size**
Binding request	20 bytes
Binding response	28 bytes
Binding indication	20 bytes
Allocate request	28 bytes
Allocate response	44 bytes
Refresh request	28 bytes
Refresh response	28 bytes
Data indication	32 bytes + data
Send indication	32 bytes + data

Table 4.5 shows the message lengths of the different STUN message types. The given message lengths may not apply to all implementations, since other ICE implementations might include additional STUN attributes to the messages. The length of STUN Data and Send indications depend on the length of the data that the message carries as its DATA attribute's value.

4.3 Baseline Measurements on a Mobile Phone

To be able to better design the measurements and interpret their results, some baseline measurements were performed on the Sony Ericsson C905 mobile phone, whose specifications were listed in Table 4.1. The baseline measurements reveal the fundamental limits of the mobile phones. The focus of the baseline measurements is on finding the maximum number of open sockets and threads a mobile phone is capable of maintaining. Simple test programs

were implemented in J2ME for the purpose of determining the limits. The measurements showed that the maximum number of open sockets the mobile phone is capable of maintaining is 50. The phone is able to create more than 50 sockets during the execution of the program, but the maximum number of simultaneously open sockets is 50. Two programs were created for testing the number of threads that the mobile phone can simultaneously support. The first test put the created threads to sleep, the other test kept all the threads running. The results showed that it makes no difference whether the started threads are running or sleeping: the maximum number of simultaneously supported threads is 509. If this limit is exceeded, the application terminates with an application error.

Based on these limiting values it is possible to conclude some theoretically limiting threshold values for the STUN, TURN and ICE protocols on a mobile phone. A STUN or TURN client can have a maximum of 50 ongoing sessions. Naturally the number of sessions is less if some sessions use multiple component IDs. A TURN server can serve a maximum of 49 TURN clients. One socket is reserved for the TURN server address and port. Also in this case, if a single client requires multiple allocations for its sessions or multiple transport layer ports per session, the number of supported TURN clients decreases. Moreover, if the TURN server itself has multiple ongoing communication sessions the number of sockets available for allocations decreases.

In general, when writing J2ME code, an own thread is needed for each listening socket since it possible to create only blocking receive() -calls on the sockets. However, the number of threads is not a limiting factor in our ICE implementation because when the maximum number of sockets is used, the maximum number of threads on the client side with 50 sessions is 150 ($3n$, where n is the number of sessions) and on the server side with 49 TURN clients is 51 ($m + 2$, where m is the number of TURN clients). More specifically, in our implementation, a single TURN client requires three threads: one for listening to the client address and port, one for refreshing the allocation on the TURN server, and for keeping the binding on the NAT alive. A TURN server, on the other hand, requires one thread for listening to the server port, one thread for checking the expiry times of all the allocations on the server, and one thread per allocation. The number of threads needed is naturally implementation dependent.

On the other hand, the case of running out of threads is more critical than running out of sockets. As a consequence of running out of threads, an application error occurs, which means that the whole application crashes. Running out of sockets causes an exception, that simply needs to be handled inside the program, but otherwise the program execution can continue normally. The conclusions we made based on the number of sockets and threads

are somewhat simplistic in the sense that they do not take into consideration the threads and sockets used by other running applications and because only a single phone model was used for the tests. Another point to take into consideration is whether it is even reasonable to consider having 50 clients running on a single mobile phone. Based on these initial measurements, we cannot make any definite conclusions on the performance of a mobile phone. That is why it is worth examining whether there are even more limiting factors for mobile peer-to-peer networking, such as the processing capacity of the phone when the number of allocations on the TURN server, or the number of sessions on a STUN or TURN client grows.

4.4 Measurement Results

The measurement results on the STUN and TURN client and server performance are presented under the following sections: mobile phone as a TURN server in Section 4.4.1, mobile phone as a STUN server in Section 4.4.2, and mobile phone as a STUN or TURN client in Section 4.4.3. In these measurements the STUN and TURN client and server are not part of a P2PSIP overlay too see the effect that keepalives or data relaying as such have on a mobile phone running the STUN or TURN prototypes. However, to see how the STUN and TURN prototypes work, when ran on a mobile phone acting as a P2PSIP peer, a few additional measurements were conducted. Those results are shown in Section 4.4.4. Finally, we present the results regarding the impact of ICE procedures on the delays in P2PSIP call establishment in Section 4.4.5.

4.4.1 Mobile Phone as a TURN Server

To assess how well a mobile phone can act as a TURN server, we measure how efficient the mobile TURN server is at relaying voice and overlay maintenance data. Additionally, we measured the battery and memory consumption, as well as the CPU load that keepalive traffic causes on a mobile phone.

Due to their limitations, it can be expected that mobile phones will act as P2PSIP clients in a P2PSIP overlay. This way they are not obliged to store data objects and route requests. They only need to maintain the connection to the P2PSIP peer they are connected to. In these measurements, we considered a case where there are no centralized servers to take care of the TURN server functionality and no PCs connected to the P2PSIP overlay. Instead, there are only mobile phones creating the P2P network. In such a case it is necessary for

the mobile phones themselves to be able to act as P2PSIP peers. Moreover, if they have a public address they also need to be able to run the TURN server functionality to allow other mobile phones with private addresses join the network.

In our measurements, the Sony Ericsson phone model C905 was used to run the TURN server prototype, W910 acted as a STUN server, and the PC Dell Latitude D610 took care of the client side functionality. In the P2P signaling message relaying measurements, we assumed the mobile phone to be acting as a TURN server for a 10000-peer overlay, where each peer initiates or forwards a message on the average every 2.65 seconds and the average message size is 819 bytes [30].

Impact of Keepalives on a TURN Server

The purpose of the test was to examine a simplified case where only the effect of keepalive traffic on a TURN server was considered. The TURN server had only one client. In the test scenario, the client and its peer ended up using a relayed path between them after performing ICE. A simple TCP relay was used as the signaling channel for exchanging the candidates. After the connection had been established, no data was sent between the TURN client and the peer, the connection was simply kept up via keepalives: STUN Binding indications were sent every 15 seconds and Refresh requests every 10 minutes. The client and its peer used the relayed path for sending keepalives to each other. Test duration was 1 hour, starting after the ICE connectivity checks had finished. The CPU load and memory consumption measurements start logging data only after the ICE connectivity checks have finished, where as the battery consumption measurements include the ICE checks.

The Figure 4.3 shows the memory consumption of the TURN server during the measurement. The used and free Java memories were observed for an hour by taking a sample of the Java memory usage once in a second. The average amount of used memory on the java phone heap was 766 kilobytes of the initially allocated 1.049 megabytes. As the Figure shows, the size of the Java heap grows repeatedly to a maximum of 1.048 megabytes, which is very close to the initially allocated memory size. The memory consumption stays far below the maximum Java heap size available to applications, which is 30 megabytes on the phone model used in the measurement. The mobile phone platform is able to increase the initially allocated Java heap size during program execution. However, we could see that the heap size was not increased during this measurement since the sum of the used and free Java memories (not shown in the figure) remained constant.

The CPU usage on the phone is shown in Figure 4.4. The CPU load was measured over the same time period as the memory consumption, thus also collecting information once

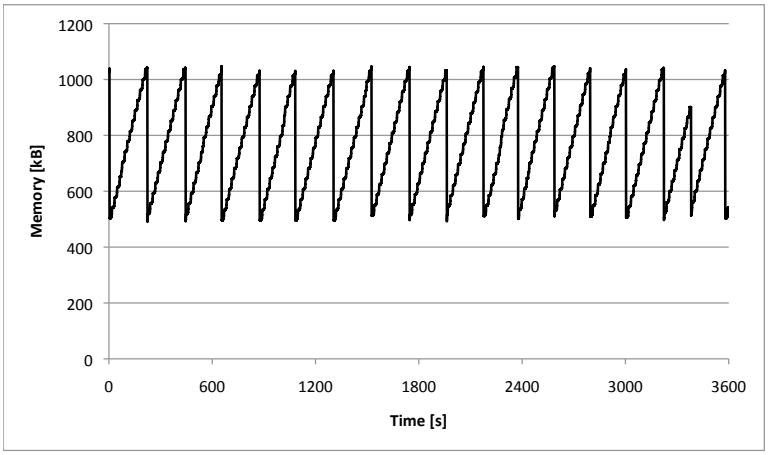

Figure 4.3: Memory consumption of a mobile TURN server caused by keepalives

per second. The average CPU load during the measurement was 3.1% with a standard deviation of 3.6. By taking the 95th percentile CPU load, we see that 95 percent of the observed CPU loads fall below 10%. As the results show, the CPU load that a single TURN client and its peer cause on the TURN server is quite low. This is because there are no heavy computational operations required and messages are received at a relatively steady rate. When the mobile phone acting as a TURN server has no clients to serve, its average CPU load is close to zero.

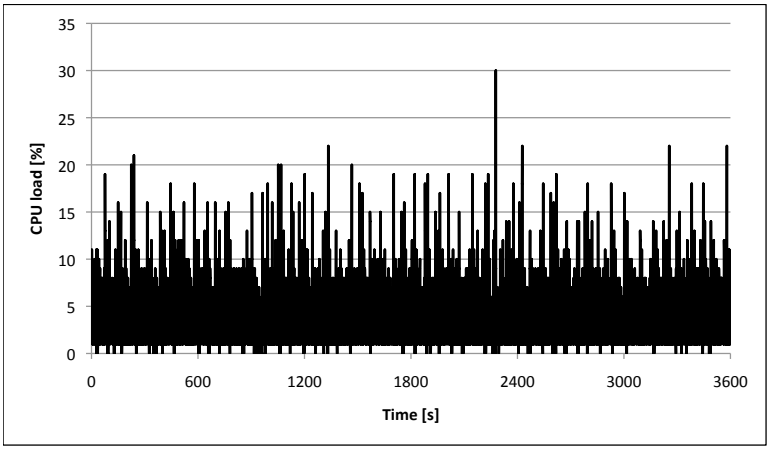

Figure 4.4: CPU load of a mobile TURN server caused by keepalives

The measurement on battery consumption lasted until the mobile phone acting as TURN server ran out of battery. The battery consumption of the mobile phone acting as TURN server is shown in Figure 4.5. The remaining battery level was measured once per minute. The mobile phone ran out of battery in 8 hours and 58 minutes. The main reason for the battery consumption was the reception and sending of STUN messages and the handling of the messages. Every time a message was received, it was checked whether the message was indeed a STUN message. All STUN messages needed to be parsed for further interpretation.

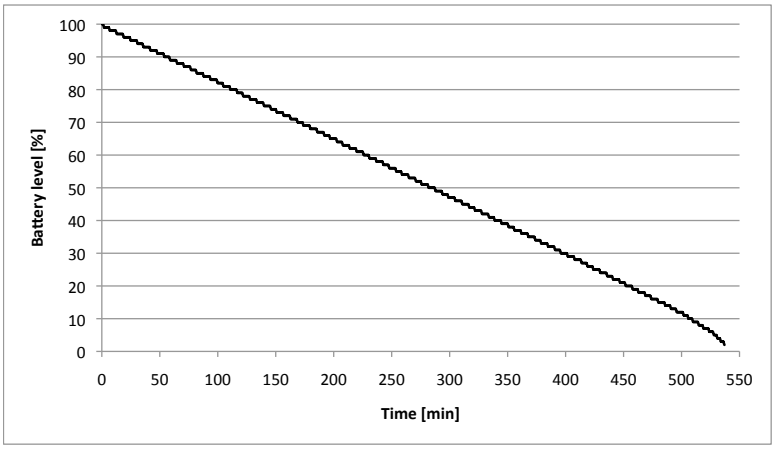

Figure 4.5: Battery consumption of a mobile TURN server caused by keepalives

The TURN server received Binding indications from the client that were addressed to the server and Binding indications that needed to be relayed from the client to its peer and vice versa. Additionally, the TURN client sent Refresh requests to the TURN server that were replied by sending back a Refresh response. During its battery lifetime the mobile phone received 2152 messages that were simply ignored (keepalives), 4304 messages that needed to be relayed and 54 messages that needed to be answered. This means that the phone received a message over the radio channel approximately once per 4.96 seconds. The average size of a received message was 34.6 bytes.

To see how the number of clients effects the CPU load and memory consumption of the mobile phone, the same test was also run with different number of clients. Table 4.6 presents the relevant values on a mobile phone acting as TURN server with 2, 5 and 10 clients. The initially allocated 1.049 MB java phone heap size was enough regardless of the number of clients used. As the table shows, the CPU load and memory consumption increase as the number of clients increases.

Table 4.6: TURN server with different number of clients (keepalives)

	CPU load			Memory consumption		
# of clients	**Average**	σ	**95th percentile**	**Average**	**Min**	**Max**
1	3.1%	3.6	10%	766 kB	492 kB	1.048 MB
2	4.9%	5.0	15%	774 kB	505 kB	1.047 MB
5	9.2%	6.9	22%	796 kB	538 kB	1.048 MB
10	14.3%	8.0	27%	820 kB	573 kB	1.048 MB

Data Relaying on a TURN Server

In the previous measurements, we tested the impact of keepalives on a TURN server running on a mobile phone. However, there is not much point in maintaining an allocation just for the purpose of sending keepalives. This section takes a look at the capabilities of a mobile phone to act as a TURN server for the purpose of relaying data, especially real-time data, such as voice. Naturally, keepalive messages are still sent in the background if no data has been sent for 15 seconds. Refresh requests are sent every 10 minutes regardless of other messages being sent.

We used the same setup as in the previous keepalive measurements but this time the TURN client and its peer started transmitting data after having established a connection using ICE. Thus, data was being relayed from the client to its peer and vice versa. When speaking of the bandwidth of an audio codec under this section, we refer to the bandwidth of a one way transmission excluding the additional bandwidth caused by STUN encapsulation. The test setup is such that the relayed path gets chosen for communication.

In the first measurement, we assumed the same audio codec G.728 Annex H that was also later used for the media in the P2PSIP call setup measurements. Using the codec, 32 byte data packets are sent every 20 milliseconds. The dropping rate of packets for a voice stream is considered acceptable if it is less than 0.05; otherwise, it is considered unacceptable. After the ICE connectivity checks had finished, we transmitted data messages between the TURN client and its peer for a 30 minute period. During this period, approximately 58% of the 180000 packets were dropped. This was of course clearly above the acceptable dropping level.

Knowing the drop rate to be that high, the next step was to look at the possible reasons for such a high drop rate when using a mobile phone to relay data. As a starting point,

we knew that the limiting factor cannot be the bandwidth, since it was measured to be 318 kbit/s in the uplink direction. The corresponding downlink speed was even higher. The bandwidth required by the audio codec G.728 Annex H was 12.8 kbit/s. Some other possible limitations to consider were the CPU power, as well as the receiving and sending buffers.

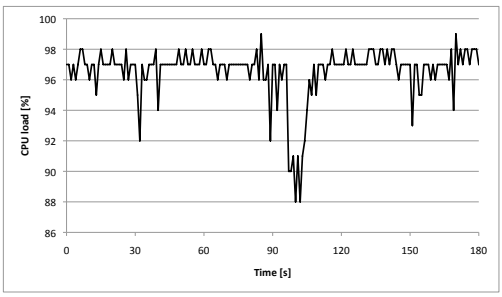

Figure 4.6: Calculating loop

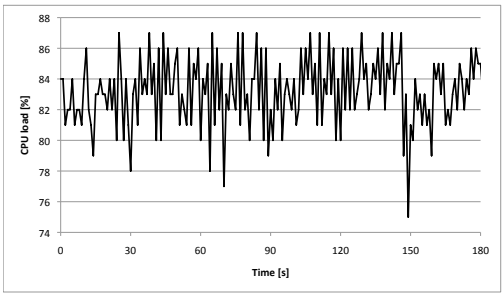

Figure 4.7: Sending loop

We made two simple programs for testing the limiting factors. One of the programs simply kept on making never-ending calculations and the other program kept on sending messages. Figure 4.6 shows the CPU load when making calculations and Figure 4.7 when the program was just sending. The average CPU load for the calculating program was 96.6% and for the sending program 83.2%. From this we could conclude that while making the calculations the mobile phone was able to utilize the entire CPU capacity allocated for the application, whereas when transmitting data the phone was not able to make use of the whole CPU processing power since the sending buffer could be limiting the speed of transmission. Consequently, if our program is relaying data (receiving and sending is handled in the same thread) and the sending buffer fills up, it reflects to the receiving buffer so that it most likely starts dropping packets.

The next test results show how well the mobile phone acting as a TURN server suits for relaying a narrow band data stream. The chosen audio codec G.723.1 uses less than half of the bandwidth utilized by G.728 Annex H. The G.723.1 codec sends a 20 byte frame once per every 30 milliseconds, thus requiring a bandwidth of 5.3 kbit/s. To further examine the kind of data transmission a mobile phone would be most suitable for relaying, we used the same bandwidth with different data packet sizes. The measurements for each data size lasted 15 minutes and were repeated 6 times. Figure 4.8 depicts how the packet drop rate depends on the transmission frequency. The figure shows the drop rate as a function of data packet size. In this context, when referring to data packet size, we mean the length of the voice data in the payload of a packet ignoring the overhead of lower layers, such as STUN, UDP,

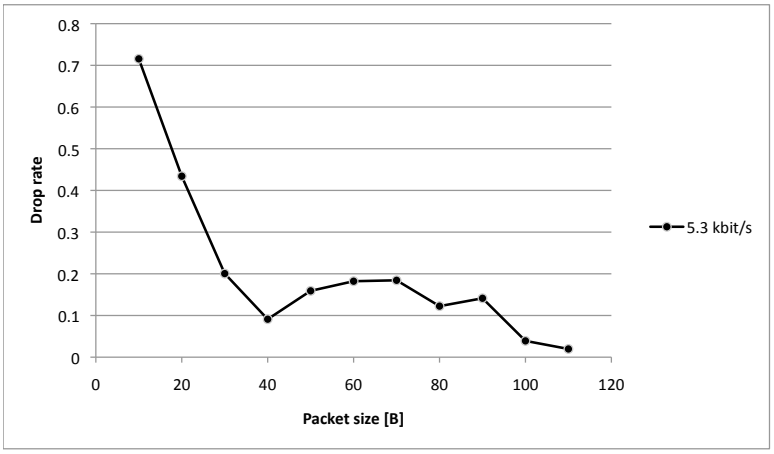

Figure 4.8: Drop rate on different packet sizes

and IP. The overhead decreases as the packet size increases. Thus, the actual bandwidth usage is much higher for small packets. Naturally, as we increased the data packet size, we decreased the transmission interval in proportion to keep the data bandwidth the same. As an example, for 20-byte data packets, the transmission interval was 30 milliseconds and for 80-byte data packets it was 120 milliseconds. For data packet sizes up to 40 bytes, the drop rate decreased as the packet size increased. However, on the tested data packet sizes of 50-90 bytes, the drop rate was higher than for the data packet size of 40 bytes.

The codec G.723.1 with a default packet size of 30 bytes falls far above the acceptable drop rate, with a drop rate of 0.43. In addition to measuring the average drop rate using the same codec, we wanted to examine whether the dropped packets were single packets every now and then, or dropped in bursts. We measured the average number of sequentially dropped packets to be 8.4. However, the maximum burst of dropped packets was as high as 46.

Despite the fact that the packet drop rate for packets over 100 bytes is less than 0.04, we chose the data packet size of 40 bytes for further measurements. This is done because we do not want the transmission interval to get too low, especially if voice data is being transmitted. Besides, G.723.1 offers an audio codec that uses 40 byte data packets for the bandwidth of 5.3 kbit/s. However, the average drop rate of the packets was above the acceptable limits, that is, on average 0.09 with a standard deviation of 0.005. Despite this, it was the most appropriate codec available, so we further measured the delays for the given data stream. G.114 is an ITU recommendation that defines acceptable delays for voice transmissions. According to the specification, voice quality is good if delays are less that 150 milliseconds, average if delays are between 150-400 milliseconds, and poor if delays are over 400 milliseconds. The average end-to-end delay for our data packets was

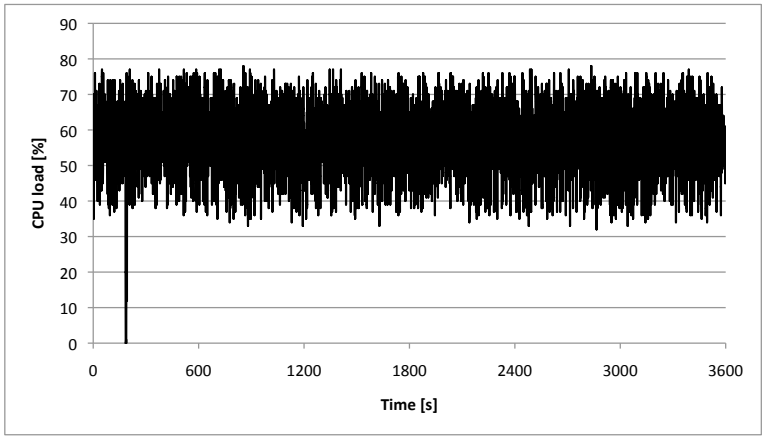

Figure 4.9: CPU load of a mobile TURN server caused by data relaying

384 milliseconds, which is just within the acceptable limits if we only observe the average value. However, this was the mean value of all the measured delays, meaning that actually only 65.1% of the measured packet delays were below the acceptable level. Moreover, the additional delay caused by the playout buffer is not even included. By examining the delays of the same frame size, only with lower transmission rate, we could confirm that the delays do depend on the transmission rate, but can only be optimized to a certain limit. Meaning that having a mobile phone as relay is nearly always costly on the delay. For example, by using the same frame size with a halved transmission rate we get only a slightly smaller average end-to-end delay of 375 milliseconds.

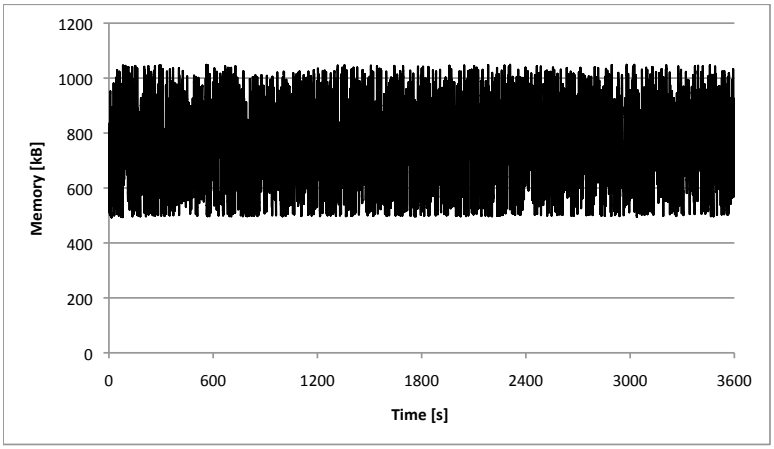

Figure 4.10: Memory consumption of a mobile TURN server caused by data relaying

Using the chosen audio codec G.723.1 with 40 byte frames, the TURN server is able to serve barely a single TURN client. Already with two clients, the drop rate is higher than 0.4. Moreover, the average end-to-end packet delay is 1429 milliseconds and the average burst of sequentially dropped packets is 15.0. Even when testing data relaying with two clients transmitting 100 bytes of data every 150 ms, the drop rate grew to 0.26.

Using the audio codec G.723.1 with 40 byte frames, we measured the CPU load and memory consumption on the TURN server. Figures 4.9 and 4.10 show the results of the measurements. The CPU load and the memory consumption were measured by taking a sample once per second. The average amount of used memory on the 1.049 MB java phone heap was 765 kilobytes. The average CPU load was 57.8% with a standard deviation of 10.2. 95 percent of the observed CPU loads fell below 73%. This is an unacceptable high load on a relay. The server received a data message that needed to be relayed approximately once every 30 ms.

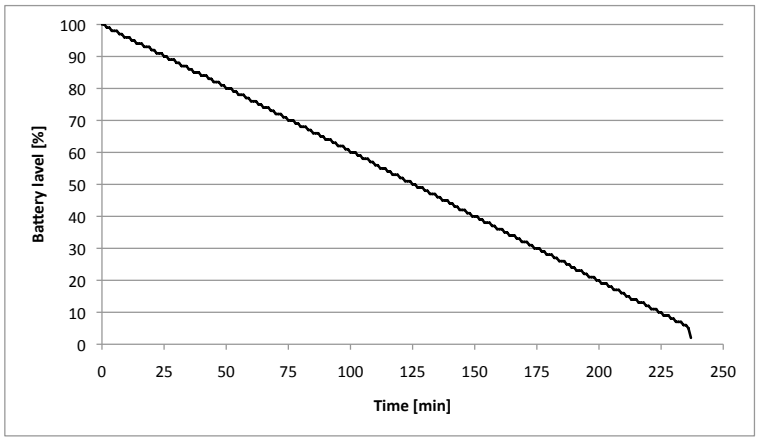

Figure 4.11: Battery consumption of a mobile TURN server caused by data relaying

We measured that a mobile phone relaying data ran out of battery in 4 hours as shown in Figure 4.11. This is a little longer than the talk time of the phone using UMTS, which is 3.5 hours. During its battery lifetime, the mobile phone received 840 messages that were simply ignored (keepalives), 420,000 messages that needed to be relayed and 21 messages that needed to be answered. No keepalives were sent through the server due to the frequent transmission of data messages. The phone received a message over the radio channel approximately once per 29.9 milliseconds and the average size of a received message was 55.9 bytes. On average, a 3.5 hour call would be rather long, but we wanted to study the maximum time that a mobile phone can act as a relay for a phone call. Moreover, we can think that the 3.5 hours call consisted of multiple sequential phone calls of different users.

P2P Signaling message Relaying on a TURN Server

One important difference between media and signaling data is that media data is much more delay-constrained. Moreover, peer-to-peer signaling uses retransmissions if an acknowledgement to a message is not received within a certain time period. However, if the drop rate of packets is high, it might be that the initial request and its retransmissions get all dropped, which makes managing the overlay impossible. The test setup stays the same as in previous measurements, except that this time signaling data is being transmitted along the relayed path. We assumed to have a 10000-peer overlay, where every node sends on average 819 bytes of overlay maintenance data every 2.65 seconds [30]. All of the maintenance data of a peer is signaled using the same allocation, since we assumed that the peer-to-peer signaling data to different peers is multiplexed using the same port.

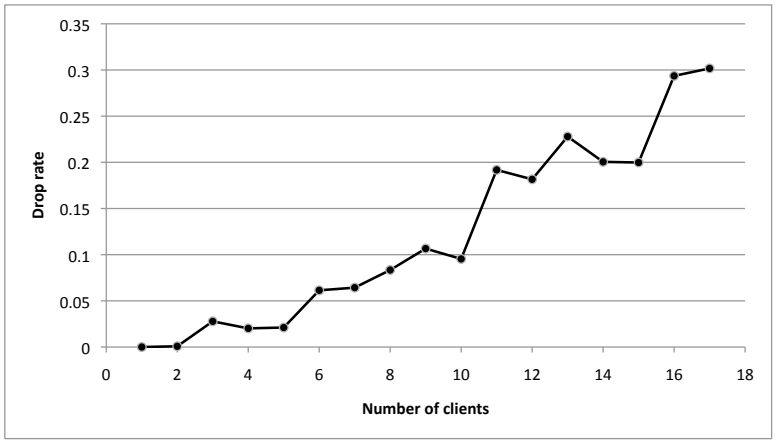

Figure 4.12: Drop rate with different number of clients

The purpose of the first measurement is to find out the maximum number of TURN clients that can send and receive peer-to-peer signaling data through a mobile phone acting as a TURN server with an acceptable drop rate. The Figure 4.12 shows the drop rate as a function of the number of clients. The drop rate was examined over 15 minute period and the measurement was repeated three times for each number of clients. As the figure shows, the TURN server was able to handle the signaling data connections of a maximum of 10 clients. Already with 11 clients the drop rate almost doubled compared to 10 clients.

With 10 clients the average time between incoming signaling messages on the server was approximately 133 milliseconds. This means that the average bandwidth consumption of 10 clients' signaling messages during the measurement period was 49.45 kbit/s for both the

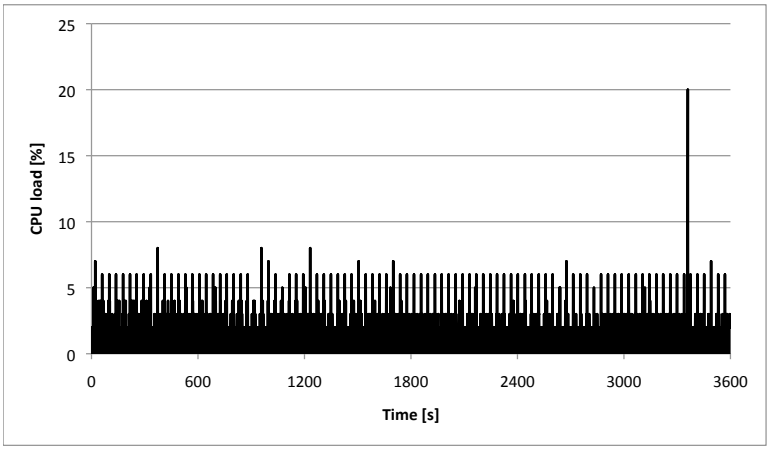

Figure 4.13: CPU load of a TURN server caused by signaling data relaying

up- and the downlink, and the dropping rate was 0.095. In the measurement, the clients did not perform any retransmissions even if packets were dropped. With retransmissions there would have been even more traffic.

Let us examine the CPU load, as well as memory and battery consumption that is caused by the signaling data of one client and its peer. The CPU load and the memory consumption were measured by taking a sample once per second for a duration of one hour. The battery consumption was again measured by recording the time it took for the mobile phone to run out of battery.

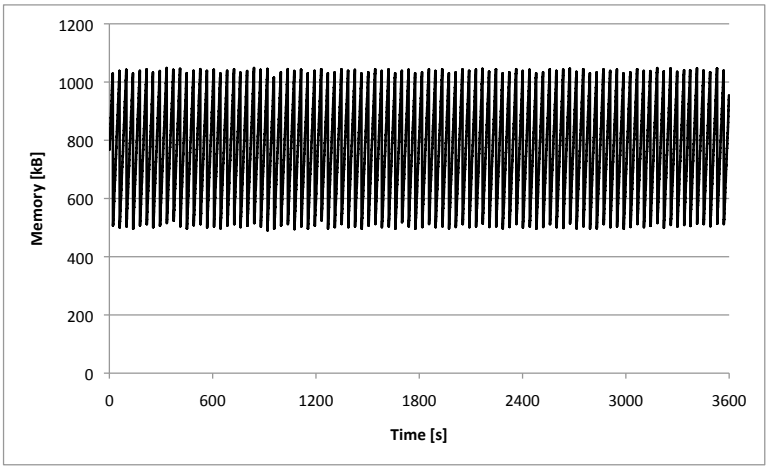

Figure 4.14: Memory consumption of a TURN server caused by signaling data relaying

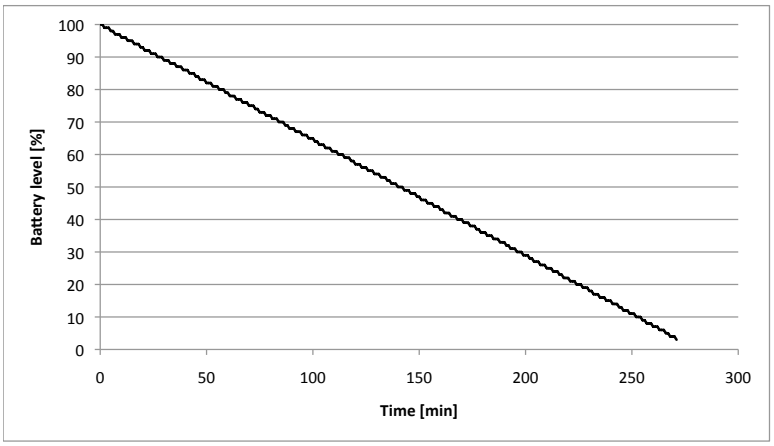

Figure 4.15: Battery consumption of a TURN server caused by signaling data relaying

During the one hour measurement the average CPU load was 1.71% with a standard deviation of 1.26. The CPU load of the phone is shown in Figure 4.13. We can calculate that 95 percent of the observed CPU load values fall below 3%. The average memory consumption on the TURN server was 772 kilobytes as depicted in Figure 4.14. The size of the Java heap varied between 490 kilobytes and 1.048 megabytes. As shown in Figure 4.15 the battery was drained in 4 hours 32 minutes. During its battery lifetime, the mobile phone received 1088 messages that were simply ignored (keepalives), 12317 signaling messages that needed to be relayed and 27 messages that needed to be answered. This means that the phone received a message over the radio channel approximately once per 1.22 seconds and the average size of a received message was 767.4 bytes.

Table 4.7: TURN server with different number of clients (signaling data)

	CPU load			**Memory consumption**		
# of clients	**Average**	σ	**95th percentile**	**Average**	**Min**	**Max**
1	1.71%	1.26	3%	772 kB	490 kB	1.048 MB
5	8.8%	3.66	15%	801 kB	547 kB	1.048 MB
10	17.32%	5.37	26%	833 kB	608 kB	1.048 MB

Table 4.7 shows the results of the CPU load and memory consumption for scenarios in which 1, 5, or 10 clients are relaying signaling data with their peers via the TURN server. As expected, the CPU load and the memory consumption grow as the number of clients grows.

4.4.2 Mobile Phone as a STUN Server

We measured the CPU load and memory consumption on a STUN server as it served different number of STUN clients. Additionally, the battery consumption of a STUN server with one STUN client was measured. The STUN server measurements were conducted similarly as the TURN server measurements. We had a STUN client and its peer that established a connection using ICE. In our measurements the Sony Ericsson phone model C905 was used to run the STUN server prototype and the PC Dell Latitude D610 ran the STUN client prototype. In these experiments, the clients did not need a relay.

Impact of Keepalives on a STUN Server

The measurements excluding the battery measurement were started as soon as the ICE checks had finished. Information on the CPU and memory usage was collected once per second. Since the STUN server does not perform relaying, once the checks have finished, the STUN server simply keeps on receiving keepalives. All other data, whether it is keepalive, voice or signaling, is transmitted directly between the client and its peer.

Table 4.8 shows the CPU load and memory consumption on a STUN server having different number of clients. As can be expected, the CPU load increases as the number of clients grows. The increase is more notable with a small number of clients; the increase from one client to 10 clients causes a 4.12 percentage unit increase, whereas the increase from 20 clients to 30 clients causes only a 0.04 percentage unit increase. The memory consumption is the same irrespective of the number of clients because the server does not maintain any state information for a client. Additionally, if we compare the table to the Table 4.6, which

Table 4.8: STUN server with different number of clients (keepalives)

	CPU load			**Memory consumption**		
# of clients	**Average**	σ	**95th percentile**	**Average**	**Min**	**Max**
1	1.91%	1.93	7%	787 kB	486 kB	1.045 MB
5	3.77%	3.56	11%	767 kB	489 B	1.046 MB
10	6.03%	4.56	13%	766 kB	490 B	1.048 MB
20	10.32%	4.55	18%	767 kB	488 B	1.045 MB
30	10.36%	4.25	19%	765 kB	492 B	1.044 MB

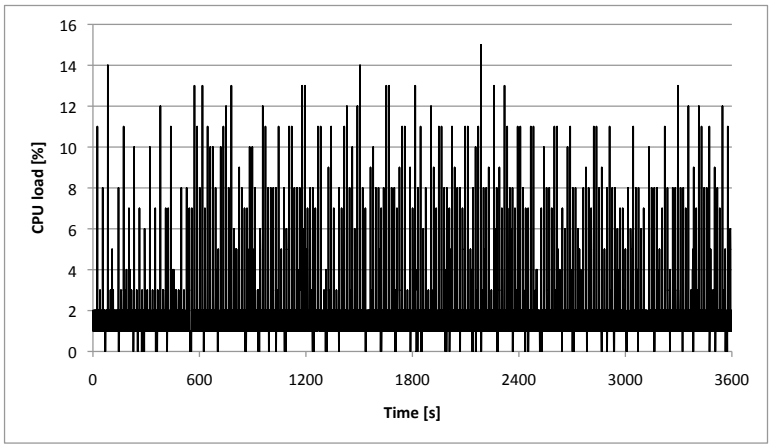

Figure 4.16: CPU load of a STUN server caused by keepalives

shows the corresponding results on a TURN server, we can notice that the CPU load of a STUN server caused by STUN clients is lower than the CPU load caused by the same number of clients on a TURN server (assuming that the clients relay packets through the server). A STUN server with 30 STUN clients has an average CPU load of 10.4%, whereas a TURN server with 10 TURN clients has an average CPU load of 14.2%.

The level of the CPU load and the amounts of used Java memory caused by one client on a STUN server were collected once per second, and are shown in Figures 4.16 and 4.17, respectively. During the one hour measurement, the server received 240 keepalives. As

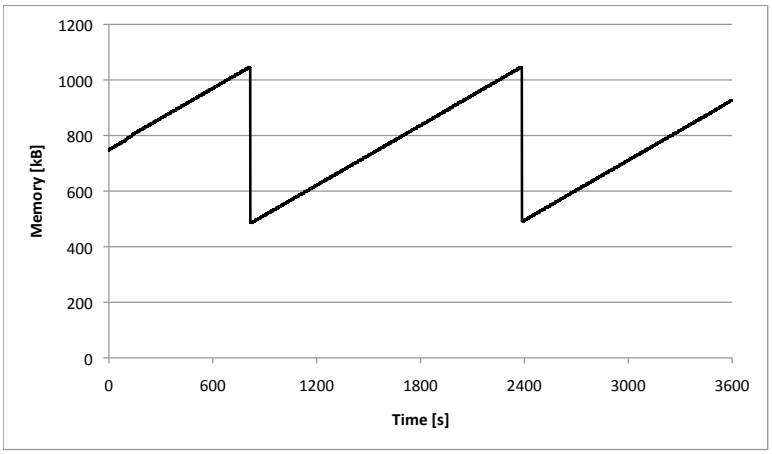

Figure 4.17: Memory consumption of a STUN server caused by keepalives

depicted in Figure 4.18, it took 9 hours and 10 minutes for the mobile phone acting as a STUN server to run out of battery. All the received messages were of the same size, that is 20 bytes, and a message was received every 15 seconds. The corresponding battery duration of a TURN server was slightly shorter, that is 8 hours and 58 minutes.

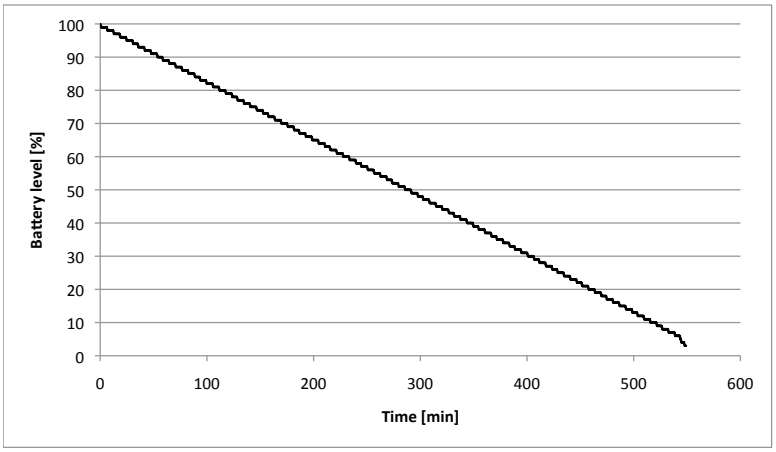

Figure 4.18: Battery consumption of a STUN server caused by keepalives

4.4.3 Mobile Phone as a STUN or TURN Client

In this section we will show the measurements that were performed in order to assess the capabilities of a mobile phone to act as a STUN or TURN client. Our purpose is in determining the impact of keepalives and signaling data on a STUN or TURN server. In our measurements, we assume that the client using a TURN server ends up using a relayed path after the ICE connectivity checks. Hereby, the difference between being a STUN or TURN client, in addition to sending Refresh requests, is that the messages that the client sends to or receives from the peer are encapsulated (when using a relay). Encapsulation requires more processing of the sent and received packets, and additionally produces overhead. The test setup is such that the Sony Ericsson phone model C905 was used to act as a STUN or TURN client, the server side functionality was run on a Planetlab machine located in the Helsinki region, and the PC Dell Latitude D610 acted as the client's peer.

Impact of Keepalives on a STUN or TURN client

We will examine the impact of keepalives on STUN and TURN clients when the client is sending keepalives to its peer and its respective server, and receiving keepalives from its peer. During an one hour measurement, information on the CPU load and memory consumption was collected once per second. The average CPU load of a STUN client (see Figure 4.19) is 2.94% with a standard deviation of 3.27. This is only a bit lower than the average CPU load of a TURN client (see Figure 4.20), which is 3.01% with a standard deviation of 3.26. For both STUN and TURN clients, the 95th percentile CPU load is 11%.

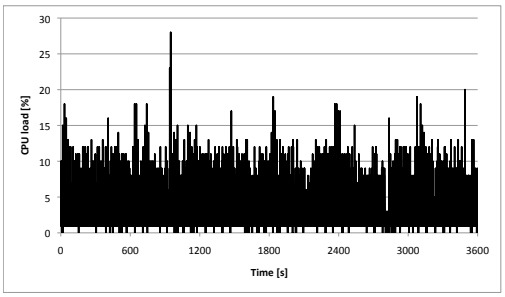

Figure 4.19: CPU load of a STUN client caused by keepalives

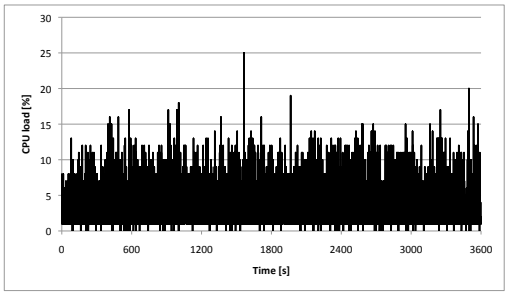

Figure 4.20: CPU load of a TURN client caused by keepalives

The average memory consumption is 768 kilobytes for the STUN client and 774 kilobytes for the TURN client. The memory consumption of STUN and TURN clients are shown in Figures 4.21 and 4.22, respectively. When looking at the figures, we can notice that the differences in being a STUN or TURN server with one connection and no data traffic is very subtle. So, we additionally tested the impact of signaling data on a STUN or TURN client. In the signaling measurements, the test setup stayed the same, except that this time, after the ICE connectivity checks the client and its peer transmitted signaling data over the created connection. Due to the increased transmission rate between the client and its peer, no keepalives were sent between them following the rules of ICE. So keepalives were sent only between the client and the server. Even with signaling, there was only a small difference in the CPU load. The results of a mobile phone acting as a STUN or TURN client with signaling data considering the CPU load and the memory consumption are shown in Table 4.9. On the other hand, there is a slight difference when a STUN or TURN client is sending and receiving keepalives or signaling data. However, this difference could be also explained by the fact that a little more code is included in the program with signaling data and the extra memory required by the variables when running the code.

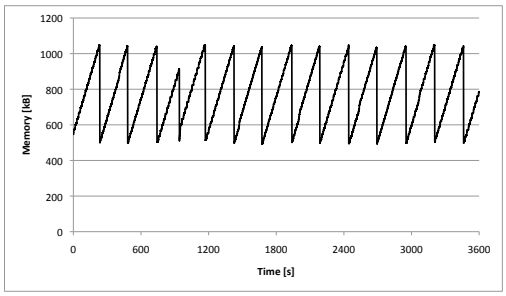

Figure 4.21: Memory consumption of a STUN client caused by keepalives

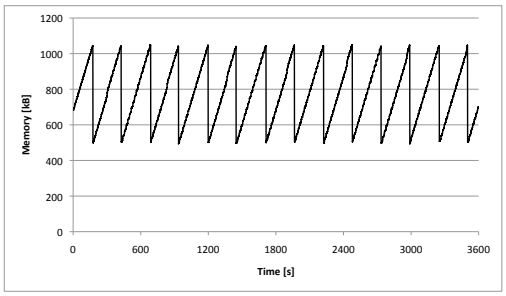

Figure 4.22: Memory consumption of a TURN client caused by keepalives

A STUN client sending only keepalives to the server and its peer, and receiving keepalives from its peer drains the battery in 9 hours 22 minutes (see Figure 4.23). For a TURN client the battery lasts 9 hours and 23 minutes (see Figure 4.24). During the measurement, the bandwidth consumption by both a STUN and TURN client is very low: the STUN client uses a 21.33 bit/s uplink bandwidth and a 10.67 bit/s downlink bandwidth, whereas the TURN client uses a 38.77 bit/s uplink bandwidth and a 28.11 bit/s downlink bandwidth. The uplink bandwidth is higher in consequence of the keepalives (Binding indications) sent to the server. The difference in bandwidth required by a STUN and TURN client for keepalives ensues from the difference in the packet sizes, not so much the transmission interval. The message transmission interval of a STUN client is 7.50 seconds and the average reception interval is 15 seconds. For the TURN client the intervals are 7.41 seconds and 14.63 seconds, respectively. The average message size for a STUN client sending and receiving keepalives is 20 bytes. For a TURN client the average message size is due to encapsulation 35.9 on the uplink and 51.4 bytes on the downlink.

We measured the battery consumption of a mobile phone acting as a STUN and TURN client sending and receiving signaling data for an one hour period. The battery charge

Table 4.9: STUN and TURN client with signaling data

	CPU load			**Memory consumption**		
# of clients	**Average**	σ	**95th percentile**	**Average**	**Min**	**Max**
STUN	0.88%	0.92	2%	780 kB	512 kB	1.048 MB
TURN	1.02%	1.31	3%	780 kB	509 kB	1.048 MB

dropped from 100% to 78% in both cases. This similar to the result in [31] where battery charge dropped from 99% to 79% during an hour, and the small difference could be explained by the fact that keepalives were not considered. But from [31] we can use the total battery duration time for signaling data to be 4 hours and 50 minutes (for both STUN and TURN client). The respective battery charge drop in one hour with keepalives only was from 100% to 89% for the STUN client and from 100% to 88% for the TURN client.

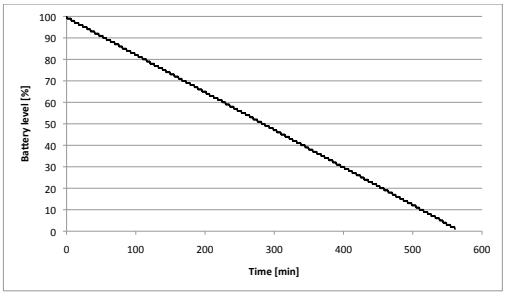

Figure 4.23: Battery consumption of a STUN client caused by keepalives

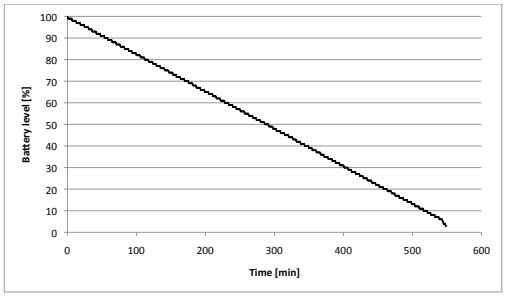

Figure 4.24: Battery consumption of a TURN client caused by keepalives

4.4.4 Mobile phone as P2PSIP peer

In the previous measurements, we examined the impact that keepalives, signaling, or data transmissions have on a mobile phone acting as a STUN or TURN client or server. Since the phone was not actually part of a P2PSIP overlay, it was easier to detect the effects that, for example, keepalives have on a STUN or TURN client or server. However, we next wanted to know the limits of a mobile phone acting as a STUN or TURN server, and at the same time acting as a peer in a P2PSIP overlay. To be able to act as a TURN server, the peer itself had to be publicly reachable. We used a publicly reachable peer also as the STUN server.

Table 4.10: Maximum numbers of STUN and TURN clients

Description	**Maximum # of clients**	**Drop rate**
TURN server with keepalives	14	-
TURN server with signaling	9	0.10
TURN server with voice data	0	-
STUN server with keepalives	1018	-

In these measurements, we were only interested in the maximum number of clients that a STUN or TURN server can serve when simultaneously sending and receiving its own overlay maintenance data. The maximum number of clients was determined solely based on drop rates. Table 4.10 summarizes the results by showing the maximum number of clients a STUN or TURN server is capable of serving and drop rate on that number of clients.

The test setups were exactly the same as in the previous corresponding measurements, except that this time STUN or TURN server exchanged overlay maintenance data with a Planetlab [40] node. The test setup in this measurement was very simplistic: all the signaling data was actually sent and received from a single node and received from a single node, no retransmissions were sent even if a packet was lost, and there was no handling of the received or sent packets (cryptographic operations, verification of certificates and signatures, and generation of signatures [30]).

The measurement started by finding an estimate for the maximum number of clients. The actual test cases covered measuring the drop rates with different numbers of clients close to the estimate. Each of the test cases was measured 3 times and the duration of a single measurement was 15 minutes. When examining a STUN or TURN server with clients sending keepalives, we did not actually measure the drop rates, but simply the fact whether a given number of clients were able to receive a Binding or Allocation request, and in case of TURN server, whether the clients were able to refresh their allocation. The STUN and TURN clients connected a server with a time interval of 3 seconds.

The results match well with the results from the previous measurements. A P2PSIP peer that acts as a TURN server was not able to relay even a single voice data connection. With one voice data connection the drop rate already grew to 0.15. The TURN server was able to relay one connection less than in the previous measurements, that is 9 connections, since it sent and received its own overlay maintenance data. With only keepalives, a TURN server was able to keep up connections of 14 clients (all need relaying). The corresponding number of clients for a STUN server is 1018. The reason for such a high number of clients a STUN server is capable of serving is that the server only has to reply to Binding requests received every 3 seconds, otherwise it can just ignore all the received messages. Besides, all the messages (except its own signaling data) are received over a single socket that is handled by a single thread. The traffic model that a mobile phone supports seems to be better suited for receiving than sending messages. With 1018 STUN clients, the STUN server receives Binding indications every 14.7 milliseconds. Moreover, hardly any retransmissions are required for the sent Binding requests.

4.4.5 Impact of NAT Traversal on Delays in P2PSIP

This measurement was performed in order to determine the delays that come with call setup in a P2PSIP overlay. Our main interest is in establishing a call between two mobile nodes acting as P2PSIP clients in a 500-peer overlay running in Planetlab. However, to be able to determine how much of the delay depends on the fact that the clients were implemented on wireless nodes the same measurements were done also for wired endpoints. In the measurements, we consider the call setup delay to be the time between initiating the call and finishing ICE connectivity checks for RTP. We examined four different scenarios for both wireless and wired clients:

1. No ICE: client nodes are publicly reachable and do not use ICE.
2. No NATs: client nodes are publicly reachable and use ICE.
3. Good NATs: client nodes are behind good NATs.
4. Bad NATs: client nodes are behind bad NATs.

Our focus was not so much on knowing the exact types of the NATs used between the P2PSIP clients, but rather on the effect that a combination of NATs has on the path selected by ICE. Generally, relaying is needed if at least one of the peers is behind a NAT that uses address and port-dependent mapping and filtering, and neither of the peers is behind an endpoint-independent mapping and filtering NAT (or directly connected to the public network). This is why we refer to NATs using endpoint-independent mapping and filtering as good NATs, whereas bad NATs use address and port-dependent mapping and filtering.

For each of the test scenarios a call was setup 50 times. The average call setup delay and the 95th percentile are shown in Figure 4.25. The line segments on top of the bars represent the confidence intervals. By using a 95% confidence level we can estimate the reliability of the results. Based on that we can determine that the differences in the results between different test cases are statistically significant. From the same figure, we can clearly see that when using the same test scenarios the average delays for the wired are lower than for the mobile.

For both, mobile and wired call setup tests, the use of bad NATs resulted in the highest delay. The lowest delay was achieved by running the test without using ICE. For the mobile tests, when the nodes were behind bad NATs the average delay was 33.5 seconds. When the nodes were behind good NATs the delay was 17.8 seconds, when the nodes were not behind NATs but ICE was used the delay was 13.6 seconds. Without ICE the delay was

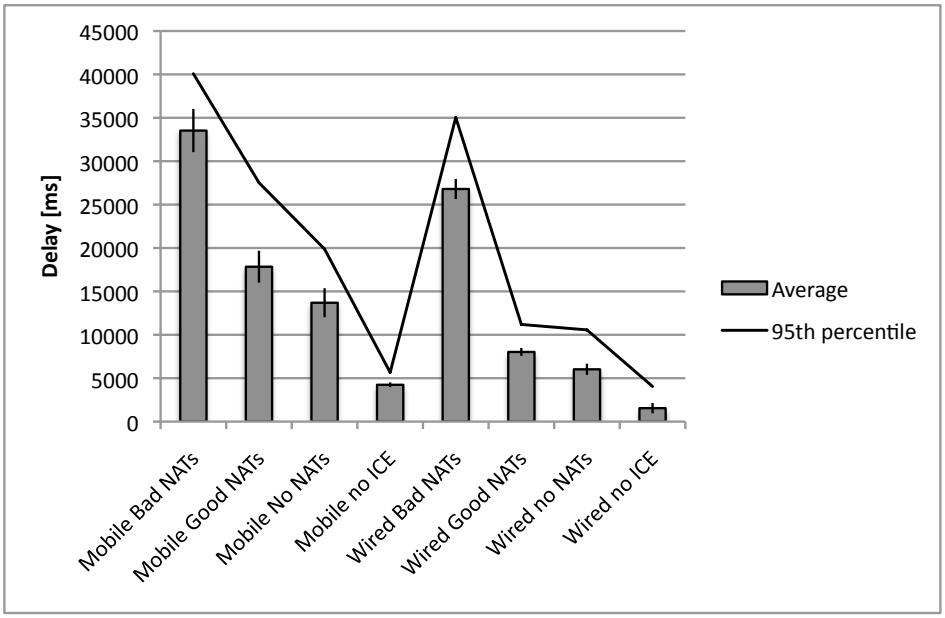

Figure 4.25: Call setup delays

4.2 seconds. The respective delays for the wired tests were 26.8 seconds, 8.0 seconds, 6.0 seconds and 1.5 seconds. The biggest difference in the delay between the mobile and wired setups was in the case where the nodes are behind good NATs, that is, 9.8 seconds. The smallest difference in the delay between the mobile and wired setups on the other hand was 2.7 seconds in the case where ICE was not used. The impact of enabling ICE in a case where the nodes are publicly reachable is substantial on the delay, in the mobile case the delay increases by a factor of 3.2 and in the wired case by a factor of 2.8. When looking at the delays of the wired setups in Figure 4.25, we can notice that the delay of the bad NAT case is notably long compared to the other wired cases. For the bad NAT scenarios the 95th percentiles are as high as 40.1 seconds for the mobile and 35.0 seconds for the wired. For the mobile scenario with the lowest delay when ICE is used (i.e., "no NAT" scenario) the 95th percentile is 19.9 seconds and for the corresponding wired scenario 10.6 seconds.

As described in Section 3.4.3 the justification for the chosen stopping criterion of the ICE connectivity checks is partly based on the ITU recommendations. ITU E.721 recommendation [53] states that the target values for mean delay for local, toll, and international connections should be 3.0, 5.0, and 8.0 seconds. Moreover, it recommends the 95th percentiles to be 6.0, 8.0, and 11.0 seconds, respectively. Regarding our measurements we are interested in the delays for the international calls. For the mobile scenarios with ICE, the

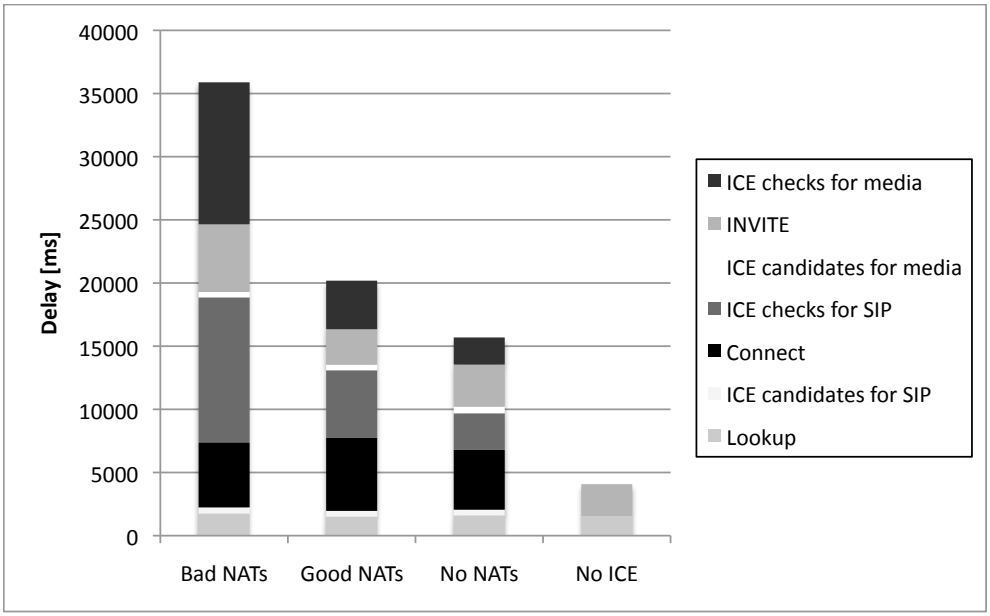

Figure 4.26: Components of call setup delay for mobile P2PSIP clients

delays never fall within the acceptable limits, since the lowest average delay is 13.7 seconds and the lowest 95th percentile delay is 19.9 seconds. For the wired scenarios using ICE, only the delays for the "bad NAT" scenario are clearly unacceptable. We use the word 'clearly', since the mean delay for the "good NAT" scenario was 8.0 seconds and the 95th percentile was slightly over the recommendation, that is, 11.2 seconds.

Figures 4.26 and 4.27 show the components of call setup delay for the different test cases. In cases where ICE is used, the delay is composed of a P2PP lookup for locating the other peer, ICE candidate gathering for SIP, a Connect transaction for exchanging the candidates, performing the ICE connectivity checks for SIP, gathering the candidates for media, exchanging the candidates in a SIP INVITE transaction, and performing ICE connectivity checks for the media. In the no ICE scenarios the delay is composed of only the Lookup and the INVITE transactions.

Figure 4.26 shows the components of call setup delay for the mobile cases. As shown in the figure for the cases using ICE, the delays for the Connect and INVITE transactions are higher than for the P2PP lookup. This is because the Connect and INVITE transactions include the candidate gathering at the called party. Moreover, the delay of the INVITE transaction is lower than the delay of the Connect transaction, since the INVITE is sent using the direct connection created for SIP, whereas the Connect message is forwarded over

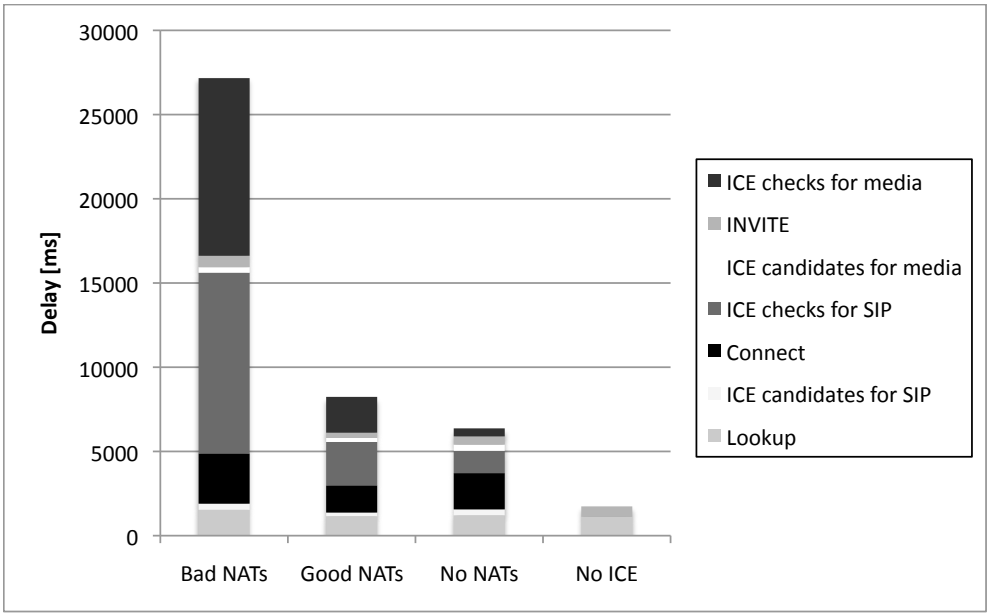

Figure 4.27: Components of call setup delay for wired P2PSIP clients

several hops in the overlay. As depicted in the Figure 4.27, the situation is not the same for wired cases using ICE, since the delay of the INVITE transaction is lower than either of the P2PP lookup or the Connect transaction. Also, in the wired no ICE case the delay of the P2PP lookup is higher than for the INVITE transaction, which makes sense, since no candidates are gathered at the called party and a direct connection is used for the INVITE message. One possible explanation why this is not the case in the mobile "no ICE" scenario could be that INVITE is much larger (number of bytes) than lookup. Moreover, the overlay is quite small and the non-wireless hops are fast. Lookup is also not terminated by a mobile phone. The delay of the INVITE transaction is the highest in scenario of bad NATs (both mobile and wired), since the message is relayed via a TURN server, and relaying naturally induces extra delay.

These results make it also possible to examine the average time it takes to create a new P2PSIP connection. The average setup delays of a P2PP or SIP connection is the sum of delays of lookup, ICE candidate gathering, Connect transaction, and ICE connectivity checks. For mobile P2PSIP, the delays are 1.7s, 9.7s, 13.1s and 18.9s in the "no ICE", "no NATs", "good NATs", and "bad NATs" scenarios, respectively. For wired P2PSIP the corresponding delays are 1.2s, 5.0s, 5.6s, and 15.6s. In the P2PSIP connection setup delay, the difference between between "no ICE" and "no NAT" scenarios is even more crucial

compared to the P2PSIP call setup delays: in the mobile case the delay is 5.8-fold and in the wired case 4.1-fold, when ICE is used.

From the Figures 4.26 and 4.27 we can additionally see that the duration of the ICE connectivity checks is always shorter for the media than for SIP. As an example, in the mobile "good NATs" scenario it takes 1.39 times longer for the ICE checks to finish for SIP than for the media. This is because of the different pacing of the ICE connectivity checks for SIP and RTP. The transmission rate of media determines the transmission rate of RTP, that is, every 20 milliseconds in our case. For SIP we used the default value of 500 milliseconds given by the ICE specification. A faster pacing would make it possible to finish checks earlier. However, a lower default value is defined by the ICE specification due to possible bandwidth limitations.

Finally, let us look at the delays caused solely by the ICE connectivity checks in view of the stopping criterion. As we know, the time before meeting the stopping criterion is not the same as delay of the ICE checks. The delay of the ICE connectivity checks is the total of meeting the stopping criterion and nominating the selected pair. The delay for nominating the pair is at minimum the RTT (Round-Trip Time) between the peers. This means that the minimum delay for the ICE checks of the "no NATs" scenarios should be $2 * RTT$, since the highest priority candidate pair gets nominated as soon as it succeeds. Moreover, for the "good NATs" and "bad NATs" scenarios the minimum delays should be 2 seconds + RTT and 10 seconds + RTT, respectively. Depending on the mutual timing of the checks, the pacing of the checks and/or possible packet losses it can be that the minimum delay is not attained. All other cases, except the mobile "good NATs" cases, follow the minimum delays with a reasonable additional delay (33-309 milliseconds), when taking an approximation for the RTT from the "no NATs" case (i.e., "no NATs" ICE check delay / 2). The difference between the minimum delay and the actual delay for the mobile "good NATs" scenario is 1893 milliseconds for SIP and 767 milliseconds for RTP. Based on this it seems that in the mobile "good NATs" scenario the ICE checks have not yet generated a valid pair for SIP or media when the 2 seconds deadline is met.

Number of ICE Messages

Figures 4.28 and 4.29 show the number of STUN messages sent by the caller during different ICE phases of the call setup. The message count includes also the retransmissions of the messages. The bars also show the confidence intervals for both, requests and retransmissions, when using a 95% confidence level. Let us first discuss the candidate gathering

phases of a call setup. The progression of the candidate gathering phase is the same regardless of the NAT scenario, since it is a simple client-server transaction. Additionally, since we are only using one TURN server during the candidate gathering, it does not either make a difference regarding whether the candidates are gathered for SIP or RTP. If there were multiple servers, the pacing between the candidate gathering would be different for SIP and RTP. This would not really affect the number of messages sent to a server but rather the the duration of the gathering. By looking at the figures, we can notice that retransmissions to the TURN server are much more likely to be needed for the mobile than for the wired test scenarios. The difference in retransmissions between mobile and wired is especially clear in the "no NAT" scenario.

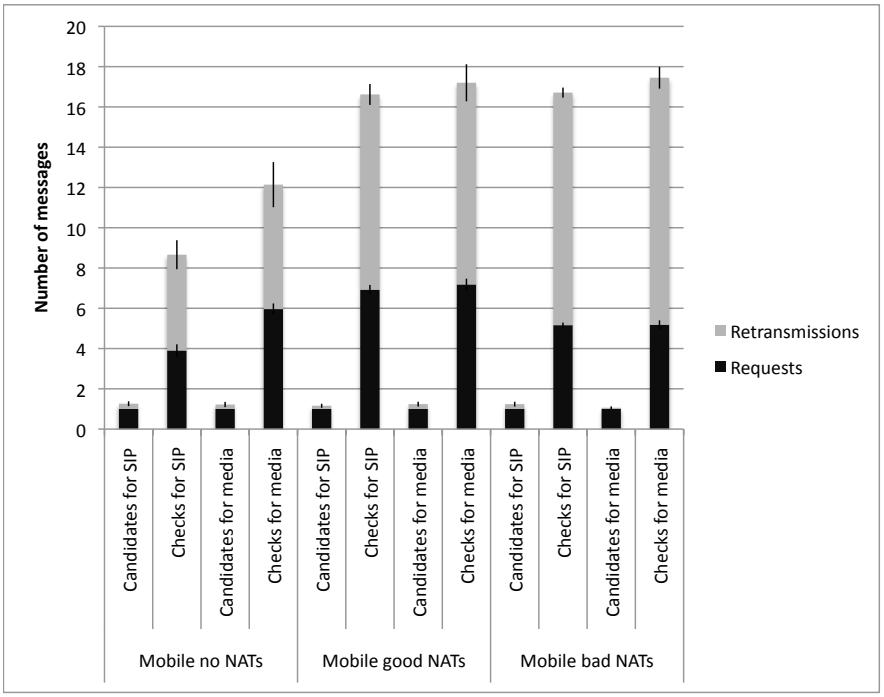

Figure 4.28: Number of STUN messages sent during call setup by a mobile P2PSIP client

As mentioned in the above section, the faster pacing of the checks for media allows the checks to finish earlier. However, when comparing the message counts of the checks for SIP and RTP, we can perceive that this is done at the cost of more messages sent. Especially clear this is in the wired "good NAT" case where the checks for RTP require on the average 5.67 messages more than the checks for SIP. Even for the wired "no NAT" case the amount of transmitted messages more than doubles. Yet, since the wired "no NAT" scenario has the lowest delay, it still only sends on average 4.83 messages during the checks for media.

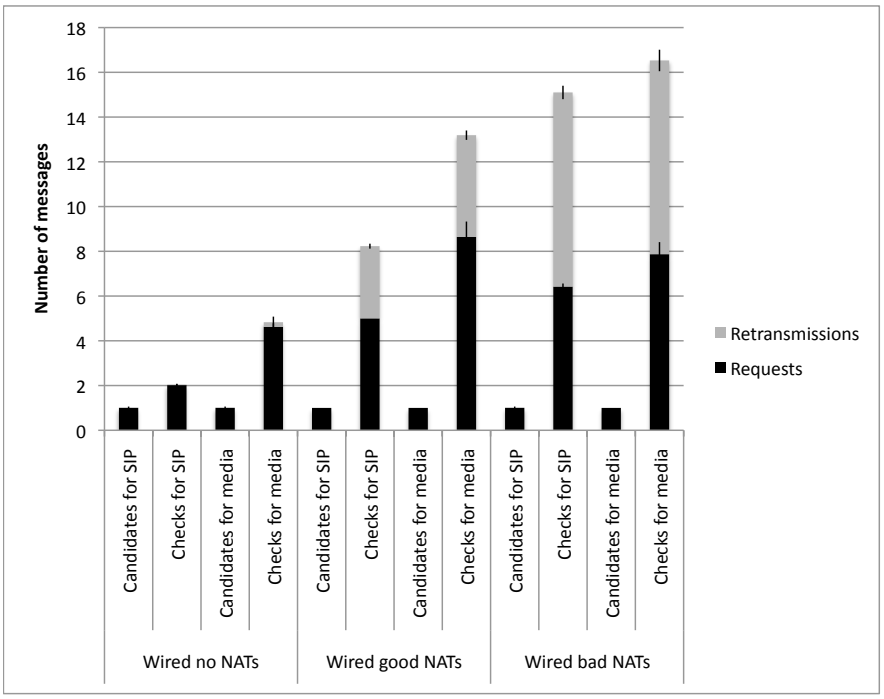

Figure 4.29: Number of STUN messages sent during call setup by a wired P2PSIP client

The amount of the all the messages sent during an ICE connectivity check is almost equal in the mobile "good NAT" and "bad NAT" scenarios, for both SIP and RTP. The difference between the respective wired ICE checks is much more noticeable. The highest number of messages sent during connectivity checks were in the case of mobile "bad NAT", that is, 17.45 messages on average. In general, the number of sent messages was higher for the mobile than for the equivalent wired scenarios.

During the checks for media in the "good NAT" scenario and during both of the checks in the "bad NAT" scenario, more requests get transmitted in the wired scenario. This is most likely due to the mutual timing that the peers perform their checks in the wired case: more request transactions get restarted due to the triggering of checks. Moreover, in the wired case more requests get transmitted in the wired "good NAT" than in the "bad NAT" scenario. The same reason applies for this: there are more request transactions to restart due to more working pairs with good NATs.

4.5 Measurement Analysis

This section makes some further analysis based on the measurement results presented above. The analysis of the results of mobile phone as a STUN or TURN server, and mobile phone as a STUN or TURN client is made from the point of view of battery consumption, CPU load, memory consumption, and drop rate. Finally, the impact of NAT traversal on delays in P2PSIP is analyzed separately from the other results.

4.5.1 Battery consumption

As we are to assume, the battery consumption results are correlated with the bandwidth consumption, as well as the transmission interval. The relation between the battery consumption and the bandwidth consumption is shown in Figure 4.30, and the relation between the battery consumption and the transmission interval is shown in Figure 4.31. With the bandwidth, we mean the combined bandwidth usage of the uplink and downlink directions, and with the transmission interval, we mean the average time between receiving or sending of a message. The figures sum up the battery consumption measurement results for one client connection presented under the results earlier. For example, the "TURN server signaling" scenario means the battery consumption is measured on a mobile TURN server that relays signaling data of one client, whereas the "TURN server keepalives" scenario means that only keepalives of one client (between the client and its peer) are relayed. The "TURN client keepalives" scenario then again measures the battery consumption of a mobile TURN

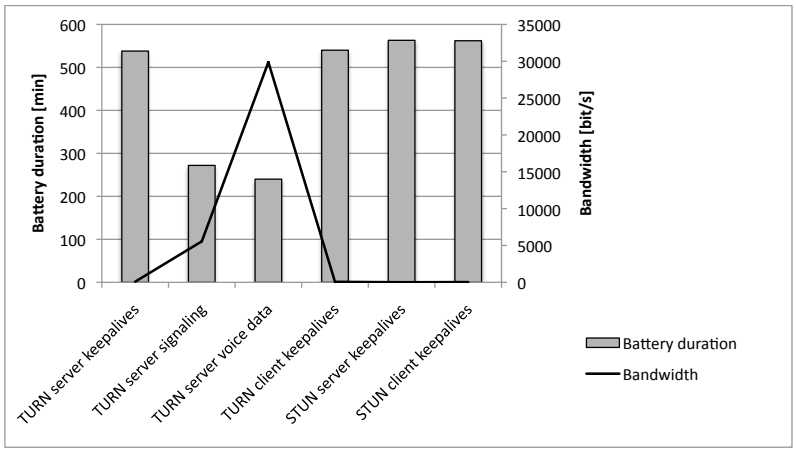

Figure 4.30: Battery duration and bandwidth

client that exchanges keepalives with its peer through a TURN server. We will first take a look at the results of a mobile phone acting as a STUN or TURN client. When the impact of keepalives only was observed, the battery duration was at a relatively acceptable level considering that messages were sent regularly over a radio channel. However, it is worth noticing that the consumed bandwidth was very low, that is, 32 bit/s on the STUN client and 66.9 bit/s on the TURN client.

If we consider that a peer implementing the ICE functionality is behind a NAT and part of a P2PSIP overlay, it also has to exchange signaling messages with its peers. We can assume that in such a case the battery duration is less than 5 hours. Naturally, the battery consumption can be even higher depending on the type of data being exchanged between the client and its peers over the connection created using ICE.

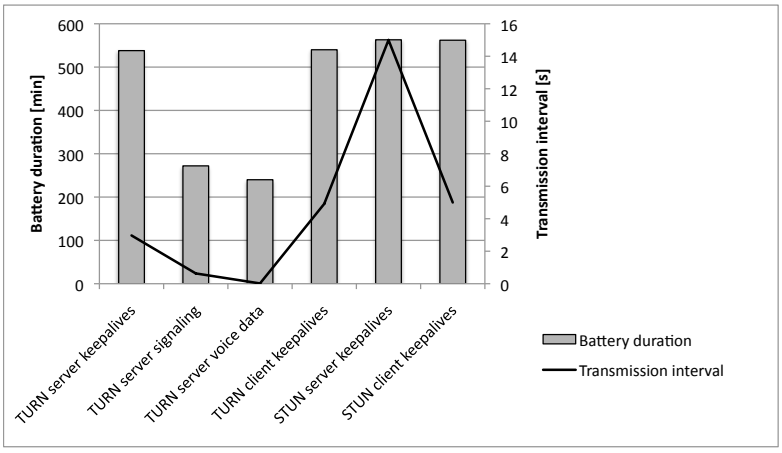

Figure 4.31: Battery duration and transmission interval

The mobile phones are by default much more suitable to work as STUN servers rather than TURN servers. This is because TURN servers implement all the same functionality as STUN servers, and additionally they take care of relaying messages between the client and its peer. A STUN server simply has to respond to incoming requests and ignore received indications. Yet, the difference of being a STUN or TURN server for one client is very subtle if only keepalives are being transmitted, even if the TURN server is relaying the keepalives of the client and its peer.

Next, let us consider a case where a publicly reachable peer is part of a P2PSIP overlay and is capable of working as a STUN and TURN server. The impact of the peer's own overlay maintenance signaling drains the battery so fast that the battery consumption that a few

STUN or TURN clients' keepalives cause on the peer is very little. Of course, the situation changes if the number of clients grows, or if some type of data is being relayed in addition to keepalives.

In a 3G Wideband Code Division Multiple Access (WCDMA) network, there are three different Radio Resource Control (RRC) states to match the power consumption level to the required traffic level [23]. For the highest power consumption state the typical power consumption is 200-400 mA [25]. According to [23], when there is not much data to transmit the battery consumption roughly halves. In the lowest connected state, where the phone can be paged but cannot transmit data, only 1-2 percent of the highest state's power is consumed. In our measurements, the mobile phone acting as a STUN or TURN client sending keepalives, or as a STUN or TURN server relaying keepalives from a single client consumed on average 99-104 mA knowing the phone's battery capacity to be 930 mAh. For a TURN server relaying overlay maintenance signaling or voice data the average power consumption was 205-233 mA. STUN client with signaling consumed on average 192 mA.

Based on this, we can conclude that the mobile phone in our keepalive measurements used mostly the middle RRC state, since it did not send or receive messages that frequently, whereas the signaling data and voice data utilize mostly the highest RRC state. The fact that HSDPA was used for the downlink should not make a big difference, as reported in [18] which measures the difference in power consumption between plain 3G and HSDPA in a similar setup. Compared to the standby time, which is 350 hours, the mobile phone drains the battery 37.4 times faster by sending keepalives (STUN client with one connection). In the case of voice data relaying (TURN server relaying one connection), the mobile phone's battery lasts half an hour longer than the talk time is specified to last.

4.5.2 Memory Consumption

Regardless of the measurement scenario, the memory curve follows a sawtooth waveform, meaning that the memory consumption grows upward and then sharply drops. The wavelength depends on the transmission or reception rate: the more frequent the transmission or reception rate, the shorter the wavelength. The used java heap size grows until it nearly reaches the initially allocated 1049 kilobytes and then drops to a size of 765-783 kilobytes depending on the test case. The drop is caused by the garbage collector that is run by the Java Virtual Machine (JVM) after the java memory heap reaches a certain heap size. Objects that are no longer used are subject to garbage collection. This is done to save memory.

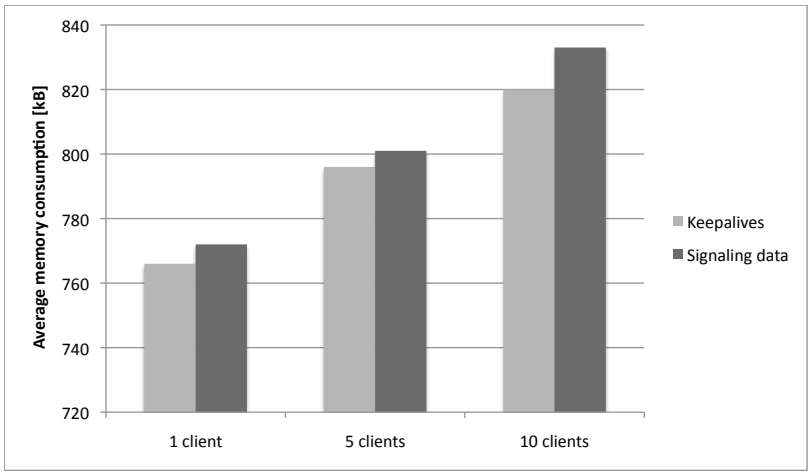

Figure 4.32: Average memory consumption of a TURN server

As already mentioned under the Section 4.4.3, there is no clear difference in the memory consumption between a STUN or TURN client with one connection. Next, let us discuss the memory consumption of STUN and TURN servers. For the STUN server, we could not notice any correlation between the number of clients and the amount of average memory used. This makes sense, since the STUN server does not keep any state information for its clients, it simply either replies to requests or ignores the indications it receives. However, a TURN server maintains information for each of its allocations, such as the expiry time of the allocation and the required transport protocol. Additionally, it creates a new thread per allocation. This means that the average, as well as the minimum used memory increases as the number of clients grows. Figure 4.32 depicts how the used average memory usage on a TURN server depends on the number of clients in case of keepalives and signaling.

4.5.3 CPU Load

To help analyzing the results on the CPU load for different test cases we have summarized the results in the Figures 4.33 and 4.34. The first Figure 4.33 shows the average CPU loads of STUN and TURN servers with different number of clients when keepalives and signaling data is being transmitted. As one would expect, the CPU load increases as the number of clients on the TURN server grows or the number of STUN sessions on a STUN client grows. One would also assume that the CPU load on a TURN server would be higher with signaling data due to the more frequent transmissions and bigger packet sizes. However, this happens only when the number of clients gets high enough, that is at least more than 5

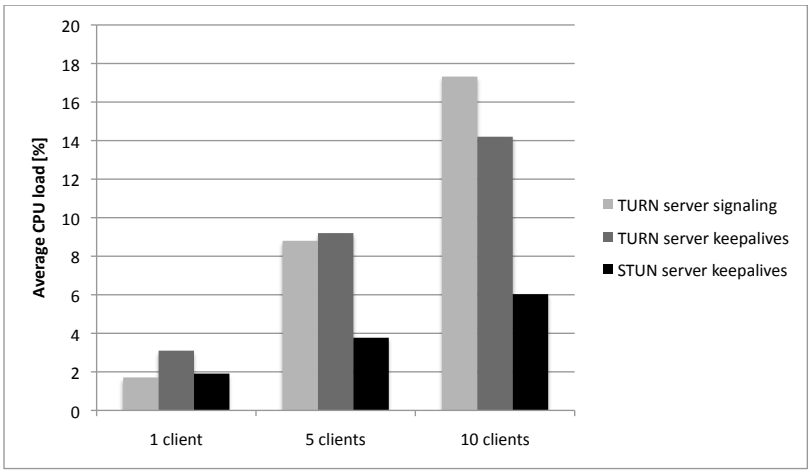

Figure 4.33: Average CPU load of a server with different number of clients

based on the figure. Moreover, the CPU load of a TURN server, with either keepalives or with signaling, is higher than the CPU load of a STUN server. This makes also sense, since the TURN server has to take care of relaying (assuming that the relayed path gets chosen). Even in case of only keepalives relaying is needed for relaying the keepalives between the client and its peer. However, there is also an exception when the number of clients is one.

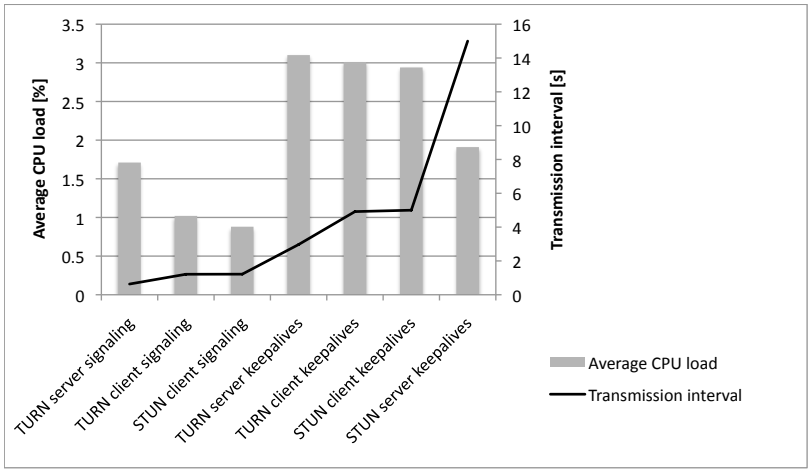

Figure 4.34: Average CPU load of different scenarios with one connection

Let us try to explain this irrational behavior by taking a look at the figure 4.34 that sums the results for all the scenarios with either a client with a single session or a server relaying messages of a single client. The figure also shows the average time interval between mes-

sages received or sent over the radio air interface for a given scenario. The figure seems to support the remark on the somewhat illogical behavior, that is, the signaling data causes consistently a lower CPU load than keepalives. This happens regardless whether it is the client or the server being observed.

We have presented the different scenarios on the x-axis in an ascendant order based on the time interval between received and sent messages. It seems that the CPU load decreases as the transmission and reception interval increases. However, as the average time interval grows over a certain threshold (around 2 seconds) the CPU load ramps slightly up again and decreases again from there. It seems that if no packets have been sent or received over the air interface for a while, the reception of a packet has a bigger impact on the growth of the CPU load. A possible explanation could be that if a radio channel gets released, the reallocation of a channel is heavy on the CPU.

For all other scenarios except for the scenario where the TURN server is relaying data, the CPU load is low, that is, on average 3% or less. If a mobile phone relays a data connection compared to keepalives only, the CPU load grows 24-fold. To put this into the context of P2PSIP networks: utilizing 57.8% of the mobile phone's CPU capacity for relaying someone else's voice call is a very high percentage.

4.5.4 Overhead Bandwidth and Drop Rate

Let us discuss the overhead bandwidth caused by relaying via a TURN server by summing up and analyzing the results of the scenarios where the TURN server was relaying 10 signaling connections or one voice data connection. One should take into account that data flowed in both directions over a single connection through the relay. Due to the frequent transmissions in both cases, there is no need to send keepalives between the client and its peer.

As Table 4.11 shows, 10 signaling connections achieve a much better throughput than one voice connection with approximately the same drop rate. This is because, in case of a voice connection, the TURN server receives packets 4.4 times more frequently ("Time between data packets" in the table). By sending smaller packets the proportion of overhead increases. By overhead we refer to the additional bytes transmitted due to the encapsulation in a STUN Send or Data indication. In fact, the proportion of overhead in a voice data packet is as high as 0.44. We have not taken the overhead of lower layers in the protocol stack, such as UDP and IP, into account.

Table 4.11: Comparison of voice and signaling data relaying

	1 x voice data	**10 x signaling data**
Packet size	40 bytes	819 bytes
STUN overhead of a single packet	32 bytes	32 bytes
Time between data packets	30 ms	133 ms
Drop rate	0.091	0.095
Data bandwidth uplink	10.67 kbit/s	49.45 kbit/s
Data bandwidth downlink	10.67 kbit/s	49.45 kbit/s
Overhead bandwidth uplink	4.27 kbit/s	0.97 kbit/s
Overhead bandwidth downlink	4.27 kbit/s	0.97 kbit/s

We can in general state that to achieve a higher throughput over a relayed connection one should send bigger packets (not too frequently, though). Of course this applies only up to a certain upper limit. Actually, the TURN specification defines as a guideline that a maximum of 500 bytes of application data in a single TURN message should not be exceeded to avoid IP fragmentation [43]. The signaling data exceeded this value, but since the packet size worked well for our measurements we did not see a reason for fragmenting the data into different packets. Moreover, as described in Section 2.3.2, if it is necessary to send large volumes of data one should use the the channel binding instead of Send and Data indications to reduce the overhead. In case of a ChannelData message the overhead for a single data packet would have been 4 bytes.

However, as the Figure 4.8 with small transmission or reception intervals showed, there are exceptions to the 'bigger the better' -rule of packet sizes on small transmission intervals. Packets with 40-bytes of application data sent every 60 ms make the packet drop rate fall below 0.1 whereas for packet sizes of 50 and 60 bytes the drop rate was above 0.1. This could be related to the way that the air interface allocates radio channel resources. Since the received capacity in the network is based on statistical traffic models, it might be that with certain packet sizes and/or transmission rates the phone receives a more reliable channel than on others. It might also depend on some of the phone's own radio interface optimization. Further analysis to explain the behavior is left for future work.

4.5.5 Call establishment in P2PSIP

In the measurements where the delays of calls established between nodes acting as clients in the P2PSIP overlay were studied, we could see that only in the cases where mobile or wired endpoints were publicly reachable (i.e., "no ICE" and "no NATs" scenarios) the delays were acceptable. The delays for the mobile P2PSIP clients in the "no ICE", "no NATs", "good NATs", and "bad NATs" scenarios were 2.7, 2.3, 2.2, and 1.3 times higher than for the respective wired scenarios. There is no specific component in the mobile call setup that would cause the delays to be higher. Instead the wireless access network causes all of the components to be about 1-9 times higher.

The additional delays when communicating over the radio interface might be caused by the (re)allocation of a dedicated channel, since in a 3G Wideband Code Division Multiple Access (WCDMA) network a dedicated channel gets released if it has not been used for several seconds. A dedicated channel enables a maximum throughput and minimum delay. An other reason could be that since the data amounts sent during the call setup are quite small, a dedicated channel is not even given to the mobile terminal. In such a case, the delay comes from sharing a common channel with other terminals. Nonetheless, since we more or less think about the network that the peers and clients are connected to as a black box, we have to think of ways how the interworking of peers in the P2PSIP overlay could reduce the delay. Or additionally, what the caller and callee (either mobile or wired) themselves could do to reduce the delay. One possible way for making the cost smaller would be by utilizing a service discovery mechanism that would let a peer or client use a TURN server geographically close to it. Naturally, given that the load on that closest server is reasonable. However, this has only a small impact on the candidate gathering phase. It might also slightly affect the duration of the ICE checks depending on the stopping criterion and candidates exchanged. First and foremost, it reduces the RTT for media once the connection is setup. Another optimization would be to use the same pacing of ICE connectivity checks for SIP as is used for RTP. As our measurement showed, a faster pacing of the checks reduce the delay for both mobile and wired nodes.

The reason why we have to repeat the ICE negotiation for both SIP and RTP between the same clients during a single phone call setup is because that is the way P2PSIP specifies it. Another reason is the possibly non-deterministic behavior of NATs, which basically means that a NAT can change its behavior due to a conflict. Anyhow, the impact of these optimizations is somewhat marginal. To more substantially cut off the delay, it would be necessary to adapt the way how SIP works. First of all, by looking at the components causing the delay, we can notice that the biggest cause of delay is the combined duration for perform-

ing the ICE connectivity checks for SIP and RTP. It would be very tempting to perform the ICE checks for SIP and RTP in a common ICE check by utilizing the frozen algorithm optimization provided by ICE when establishing a connection for more than one transport layer port (i.e., one for SIP and one for RTP in our optimization). A single ICE connectivity check for two transport layer ports lasts slightly longer than for one transport layer port, but it would totally remove the delay caused by a second connectivity check. Additionally, this would mean that the candidates would be gathered during a single candidate gathering phase rather than two separate, which would further reduce the delay. Of course, this optimization has the drawback that if the SIP INVITE gets rejected (e.g., because the called user is engaged in another call), we have created the connection for RTP in vain. However, the optimization breaks the way things are specified, not only by SIP but by ICE and RELOAD as well. According to SIP [45] the offer/answer model is used to agree the media streams, codecs, ports, IP addresses, quality of service (QoS), etc. ICE [42] states that the frozen algorithm should be used when multiple transport layer ports are required for components of a stream, such as voice and video. Due to the restrictions concerning RELOAD only the exchange of candidates for a single component ID is supported. However, as presented in [10] the problem with separate negotiations can be solved by using Host Identity Protocol (HIP) based overlays, since HIP allows re-using a path created with ICE.

Yet another optimization could be used, if we consider the P2PSIP application to be such that in addition to making VoIP calls the applications supports other type of features such as a friend list, showing the presence information of the friends. When the user would start the application, the application would create a direct SIP connection to all of the friends currently using the application. Using the SIP connection the user's terminal would send a SIP SUBSCRIBE message to the friends to receive their status and possible status updates later. Since it is most likely that a user making a call would be making it to one of his friends, he could use the existing SIP connection for sending a SIP INVITE message. Meaning that the delay for the call setup would be more than halved (including the ICE candidate gathering for media, SIP INVITE and ICE checks for media).

4.5.6 Generality of the Measurement Results

Since most of our measurements use mobile phones, we are dependent on the prevailing conditions in the wireless access networks, such as how many clients a base station is currently serving. Also, we used only one mobile phone model for carrying out the measurements, so we cannot know how much the results are affected by the given phone's features or optimizations.

In the P2PSIP delay measurements, it is very likely that the Planetlab nodes are more congested than some other nodes in the Internet joining a P2PSIP network would be. This might have affected the time it took to transmit messages via the overlay, or the number of retransmissions required. Also, the used stopping criterion has a considerable impact on the duration of the ICE connectivity checks and the amount of traffic generated, especially in scenarios were a relayed candidate pair gets selected. By implementing a different stopping criterion the outcome could have been different.

In the measurements where the performance of a mobile phone acting as a TURN server or client was tested, the test setup was such that the relayed path always got selected. We did not test the use of a TURN server without using the relaying servive because in that case we considered the results to be very similar to the results of using a STUN server. Additionally, the performance of a mobile phone as a STUN server is likely to be considerable worse if the keepalives were to be implemented as Binding requests and not as Binding indications that need no further handling.

Moreover, we should take into consideration the impact that the implementation itself has on the results. Implementation specific choices, such as the optimality of the chosen algorithms, may reflect on the results.

4.5.7 Measurement Observations

As we mentioned in Section 3.4.2, in J2ME, the address of the sender of a message gets returned as a host name, if one exists. This already caused problems during the implementation phase of the ICE prototype; three additional attributes were needed to be implemented. While doing the measurements we could, however, notice that seldom, yet frequently enough to make things complicated, the IP address gets returned, even if a host name corresponding to the IP address exists.

There are multiple situations, especially in connection with the TURN prototype, for which this kind of inconsistent behavior causes problems. For example, the permission of a peer allowed to send data for a TURN client via the TURN server, is tied to the host name or IP address of the peer that the TURN client has communicated to the TURN server (in a PEER-HOST-NAME-ADDRESS or PEER-ADDRESS attribute). So, if a TURN client has requested a permission for a peer based on its host name, but the TURN server would every now and then interpret the received messages from the peer based on their IP address, those messages would simply get dropped at the server. The same applies for the TURN client, if the TURN server has interpreted the address of the TURN client as a host name as the

TURN client made an allocation on the server. Now if the TURN server ever interprets the address of a Send indication or a Refresh request as an IP address, the server thinks that it does not have a corresponding allocation and simply drops the received messages. An allocation is among others identified based on the address and port of the TURN client.

To work around this problem, we implemented a simple DNS server. Our ICE implementation was able to make reverse DNS queries on the server. The DNS server was actually a piece of J2SE code that could easily solve the host name corresponding to an IP address by using the existing J2SE libraries. We used an informal protocol for communication between our ICE implementation and the simple DNS server. Of course, a DNS query caused additional delay. But we used a cache (implemented as a hash table) for the already resolved IP addresses that was maintained for the duration of a program run. We decided to resolve IP addresses to host names, and not the other way around, since it was clearly more likely that a mobile phone interpreted an address as a host name than IP address.

4.6 Summary

We used our ICE prototype for determine how well a mobile phone can act as a STUN or TURN client or server. We examined the performance with keepalives, P2PSIP overlay maintenance data and voice data, and measured the battery and memory consumption, the CPU load, and the drop rate. We also measured the call establishment delay in a P2PSIP overlay for both mobile and wired clients. In order for the P2PSIP prototype to work in the presence of NATs, it was integrated with the ICE prototype. The P2PSIP and ICE prototypes are provided in two versions, one for PCs and one for mobile phones.

From the measurements we found out that when a mobile phone acting as a STUN client sends only keepalives, it drains the battery 37.4 times faster than if it would in a standby mode. So, the battery consumption is a crucial limit on performance. For all other scenarios, except for the scenarios with voice data, the CPU load of a single connection (either as a client or relay) is rather low, that is, on average 3% or less. By sending smaller packets via a TURN server, the proportion of STUN overhead increases. In fact, the proportion of STUN overhead in a voice data packet is as high as 44%. However, when a mobile phone is also acting as a P2PSIP peer it is not able to relay even a single voice data connection. The TURN server was able to relay up to 9 P2PSIP overlay maintenance data connections with a drop rate of 0.1.

The lowest delay in the P2PSIP call setup measurements was achieved without using ICE. In the mobile tests, When the nodes were not behind NATs but ICE was used, the delay was 13.6 seconds. Without ICE the delay was 4.2 seconds. When the nodes were behind good NATs the delay was 17.8 seconds, and when they were behind bad NATs the delay was as high as 33.5 seconds. The delays for the wired scenarios were on average 2.7-9.8 seconds less, depending on the scenario.

Chapter 5

Discussion

This Chapter discusses how NAT traversal could actually work in a P2PSIP network that is run entirely on mobile phones acting as peers. Then we go through some possible future work that could follow up from the findings of this thesis. Finally, we shortly sum up the discussion of this chapter.

5.1 NAT Traversal on Mobile P2PSIP Peers

Under this section, we will not draw any definite performance thresholds on the working of a P2PSIP network that consists entirely of mobile phones. This is because the mobile phones acting as STUN and TURN clients and servers in most of our measurements were not actually part of a mobile P2PSIP overlay, and in the P2PSIP delay measurements we used the mobile phones as P2PSIP clients. Besides, the performance of a P2PSIP network (without NATs though) has already been examined for instance in [30]. However, we should be able to make some outlining.

As derived from the baseline measurements on our test mobile phone in Section 4.3, the upper limit on the number of simultaneous open sockets on the Sony Ericsson phone is 50. Based on our measurements, this should be more than enough in most cases, since it is very likely that the performance restricts the number of sockets even before the threshold is encountered. However, in the context of a low-bandwidth P2PSIP application, such as an instant messaging and presence application, it could be that a peer behind a NAT would need to have as many STUN or TURN sessions as it has friends online. That is because the SIP subscription to each friend's presence status requires its own ICE established connection.

Even though the mobile phone's air interface is not really designed to work with VoIP traffic, it was an interesting measurement to make. It let us better understand the kind of traffic suitable for a mobile phone acting as a P2PSIP peer with NAT traversal capabilities. We were especially interested in the different traffic scenarios in the context of a TURN server. From the point of view of a STUN or TURN client, if we have frequently (transmission interval < keepalive timer Tr) application data to send, the amount of sent STUN messages actually decreases, since no keepalives to the peer are required. Moreover, when a mobile phone uses a relay the overhead of the sent and received messages increases. However, as shown in our measurements, the transmission interval is more crucial for the performance than the proportion of the overhead.

As shown by the measurements in [30] the sending and receiving of overlay maintenance data is especially consuming on the battery. This is something that the mobile users will also most certainly be aware of. Not to mention the scenarios where a peer in the overlay, is additionally relaying overlay maintenance data of other peers behind bad NATs. From this we could reason that if a user is not currently using the application, he or she will most probably close the P2PSIP application before using it again. This will increase the amount of joins and leaves to the network compared to a P2PSIP application running on PCs. Behavior like this will make things problematic for a peer behind a bad NAT, since its connections are likely to break if a peer acting as its TURN server leaves the network. Another natural reason for a connection to break is the mobile running out of battery. This is why it might be sensible to take into account the remaining battery life of a TURN server already in the server selection phase. Further, since a mobile phone is aware of its own battery state, a "TURN handover" might be a desirable feature for mobile phones to implement. However, such handover mechanisms are not specified by TURN. Yet, it could be realized by modifying the ALTERNATE-SERVER mechanism [44] to be used even after a connection through a TURN relay has been established.

Moreover, we can assume that the users of mobile phones (with a public address) acting as TURN servers are not happy about having to relay the connections of others. This kind of turns the whole issue with NATs being problematic for P2P communications upside down: users might actually benefit performance-wise from being behind (endpoint-independent mapping and filtering supporting) NATs, since then they would never have to relay connections of other peers.

Since we have no information on the existence and proportions of different NAT types in mobile phone networks, we will have to depend on the results in wired networks. One nice possibility would be that since the mobile networks have only more recently (compared to

wired networks) started to have NATed addresses, the NATs would follow the recommendation for NATs more faithfully. The Requirements for Unicast UDP [3] state that a NAT must have an endpoint-independent mapping behavior. Additionally, it is recommended that P2P-friendly NATs would use endpoint-independent or address-dependent filtering [3]. However, since we do not know that, we will use the values from Section 2.4 that presented the existence of different NAT types in wired networks. So, we can consider that the percentage of publicly reachable peers is 20%. This would mean in a 1000-peer P2PSIP network that 200 peers would be capable of acting as TURN servers. Moreover, Table 2.6 showed that approximately 6% of peers need relaying, that is, 60 peers in our example network. Each peer is likely to need multiple connections to be relayed. Based on this, in a P2PSIP network running on mobile phones it is more important than ever to have the cost of relaying equally distributed among the TURN servers.

Let us next focus on discussing in more detail TURN servers that are run on mobile phones acting as P2PSIP peers. Firstly, the fact how many clients a TURN server is capable of serving depends on the type of data relayed through the server. Moreover, the peer acting as a TURN server can have one or more connections for its own purposes. This naturally also affects the number of allocations it can have and means that a TURN server needs to be able to determine when it is no longer able to make more allocations than it already has. It would be also preferable if the client could tell the bandwidth requirements it has for the connection that needs to be relayed. Unfortunately, the TURN specification has removed the BANDWIDTH attribute, which still used to exist in version 7 of the specification. The same version also included an error code 507 “Insufficient Bandwidth Capacity”, that the TURN server could have simply used to denote that the TURN server in its current state cannot fulfill the requested bandwidth requirements. Nonetheless, the specification states that the removed properties could be easily added back later. To express that the server has no more relayed transport addresses available at the moment is denoted by an existing error code 508 “Insufficient Capacity”.

A TURN server without proper security mechanisms is an easy target for a denial of service attack since an attacker only needs to make a maximum of 49 allocations on a server to make it unavailable for other clients. This is why it is important to impose limits on the number of allocations that a client with a given username can have at once. The error code 486 “Allocation Quota Reached” exists for this purpose.

5.2 Future Work

By looking at the results in Section 4.4.5 on the delay of a P2PSIP call setup, we can clearly notice that some improvements are needed for the delay to stay below the acceptable limits. Even with small optimizations, such as adjusting the stopping criterion of the ICE connectivity checks, we could make a huge difference. For example, since the call setup repeats the ICE checks twice, we could let the results from the first check influence the stopping criterion used for the second check. Naturally, we cannot trust the outcome of the first ICE check if only due to the deterministic behavior of NATs. However, if a check on the same type of candidate pair that got selected in the first ICE checks would succeed in the second, it could cause the stopping criterion to be met earlier than it otherwise would be. We did not implement any optimizations, but they could be an interesting topic for future work.

5.3 Summary

Since we have no information on the existence and proportions of different NAT types in mobile phone networks, it is hard to make any definite statements about how NAT traversal would actually work in a real mobile P2PSIP environment. However, based on our measurements presented in the previous chapter, mobile P2PSIP peers running ICE could relay signaling traffic but not media. Therefore, they would be most suitable for a narrowband application, such as instant messaging and presence information.

The fact that running a P2PSIP prototype is very battery consuming even without ICE [30], could lead to a growth in the frequency of joins and leaves to the network. This could, then again, make it more likely that connections via TURN server get broken. In a P2PSIP network implemented on mobile phones, it is even more important that the cost of relaying is equally distributed among the TURN servers. Moreover, the servers should limit the number of allocations and the type of data being relayed.

Due to the high delays in P2PSIP when ICE is used, optimizations for the call setup are required, such as adjusting the stopping criterion. The optimizations are left for future work.

Chapter 6

Conclusions

The need for network address translators existed even before more than a billion wireless devices, such as mobile phones, needed access to the Internet. Now that the peer-to-peer applications are spreading to mobile phones, the phones are confronted with the same challenges as computers are with NATs. This is why additional tricks, commonly referred to as NAT traversal techniques, are required also on mobile phones. ICE provides a complete NAT traversal solution. It is not interested in figuring out the types of the NATs between the peers; it simply tries to find the most optimal working path between the peers. We implemented an ICE prototype using J2ME for examining the applicability of NAT traversal mechanisms for mobile phones in the context of P2PSIP. The prototype was implemented according to the ICE specification, except that it was necessary to add three non-standard attributes to our implementation due to the incapability of the J2ME library to resolve a host name into an IP address.

Among other things, we examined the impact of NAT traversal on a mobile phone from the view point of CPU load, memory consumption, and battery consumption. In these measurements the mobile phone running the ICE prototype was not actually part of a P2PSIP overlay to better distinguish the effects of NAT traversal only. A STUN client that has established a connection to its peer drains the battery in 9 hours 22 minutes when no data but only keepalives are transmitted over the connection. In contrast, for a TURN server that is relaying P2PSIP overlay maintenance data between a single client and its peer, the battery duration is 4 hours 32 minutes. Due to the high battery consumption, users may have low incentives to allow their phones to act as TURN servers. For all the measured scenarios with one connection, the used momentary java heap size varies between 765 and 1049 kilobytes. The average memory usage of a TURN server is clearly dependent on the number of clients

it has; the average memory consumption grows approximately 5 kilobytes per allocation. If either keepalives or P2PSIP overlay maintenance data are being transmitted between a STUN or TURN client and its peer or relayed by a TURN server, the CPU load is rather low, that is, on average 3% or less. However, if a mobile phone relays a voice data connection compared to keepalives only, the CPU load grows 24-fold, which means utilizing on average almost 60% of the entire CPU power. The downside of relaying is, in addition to the increased delay, the increased proportion of overhead due to the encapsulation in a STUN Send or Data indication.

We also examined how well a mobile phone being part of a P2PSIP overlay can act as a STUN or TURN server. A P2PSIP peer acting as a TURN server cannot relay even a single media connection, even if a narrow band voice codec was used. However, a mobile phone can relay infrequent signaling and and can act as a STUN server. A TURN server running on a phone was able to relay 14 connections with only keepalives and 9 connections with P2PSIP overlay maintenance data, assuming it did not have any ongoing data connections of its own. If more connections were added, the drop rate became unacceptably high. A mobile P2PSIP peer worked surprisingly well as a STUN server, especially if the keepalives to the server are implemented as Binding indications, which is the standard way (no reply is needed). This means that in a P2PSIP network with mobile phones only it is preferable for peers to be publicly reachable or behind NATs that use endpoint-independent mapping and filtering since then relays are not needed.

The ICE prototype was integrated to a P2PSIP prototype for the purpose of measuring the delays that NAT traversal causes in a call establishment between mobile and wired P2PSIP clients. As expected, the average delays and the average amount of STUN messages sent for the wired scenarios were lower than for the corresponding mobile scenarios, that is 2.7-9.5 seconds depending on the scenario. In a mobile scenario, where the nodes are publicly reachable, the use of ICE increases the delay by a factor of 3.2. ITU E.721 recommendation states an average delay of 8.0 seconds and a 95th percentile of 11.0 seconds for international calls. In the mobile scenarios with ICE, the delays never fell within the acceptable limits, since the lowest average delay was 13.7 seconds and the lowest 95th percentile delay was 19.9 seconds. In the wired scenarios using ICE, only the delays for the publicly reachable clients fell within the limits. During a P2PSIP call establishment, the ICE procedures are repeated twice, once for SIP and once for RTP. The ICE connectivity checks finished faster for RTP than SIP due to a different pacing of the checks. However, a faster pacing comes with the cost of more messages being sent. Since the delays, especially in the mobile scenarios using ICE, were clearly unacceptable, we presented some possible suggestions to speed up the P2PSIP call setup.

When starting with the thesis, we envisioned a P2PSIP overlay that would be run entirely on mobile phones. Despite the fact that mobile phones are able to perform both the client and server side functinality of STUN and TURN, the limitations of mobile phones make it unfeasible to have the mobile phones act as a server, especially in a P2PSIP overlay. Whenever possible, it is better for the mobile phones to act as P2PSIP clients, or if acting as P2PSIP peers, have e.g. a desktop computer perform the STUN or TURN server functionality. Some of the delay caused by ICE when creating a connection follows from the way ICE is specified (in addition to the locally optimizable stopping criterion): ICE focuses a lot on ensuring that the most optimal path is found even in rare scenarios (checks are sent from relayed addresses to host addresses with a relatively high priority). It follows that the way of finding the path might not be the most optimal itself in some more common scenarios, in terms of delay and the traffic generated.

Bibliography

[1] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, April 1995.

[2] P. Almquist. Type of Service in the Internet Protocol Suite. RFC 1349 (Proposed Standard), July 1992. Obsoleted by RFC 2474.

[3] S. Asadullah, A. Ahmed, C. Popoviciu, P. Savola, and J. Palet. ISP IPv6 Deployment Scenarios in Broadband Access Networks. RFC 4779 (Informational), January 2007.

[4] F. Audet and C. Jennings. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787 (Best Current Practice), January 2007.

[5] Finnish Communications Regulatory Authority. Network speed test. https://nettimittari.ficora.fi. Accessed, February 23, 2010.

[6] S. Baset, H. Schulzrinne, and M. Matuszewski. Peer-to-Peer Protocol (P2PP). Internet Draft (Standards Track), November 2007.

[7] L. Bo, S. Xie, G. Y. Keung, J. Liu, I. Stoica, H. Zhang, and X. Zhang. *An Empirical Study of the Coolstreaming+ System*, volume 25. IEEE Journal on Selected Areas in Communications, December 2007.

[8] D. Bryan, P. Matthews, E. Shim, D. Willis, and S. Dawkins. Concepts and Terminology for Peer to Peer SIP. Internet-Draft (Informational), July 2008.

[9] D. Bryan, E. Shim, B. Lowekamp, and S. Dawkins. Application Scenarios for Peer-to-Peer Session Initiation Protocol (P2PSIP). Internet-Draft (Informational), November 2007.

[10] G. Camarillo, P. Nikander, J. Hautakorpi, and A. Keränen. HIP BONE: Host Identity Protocol (HIP) Based Overlay Networking Environment. experimental, October 2010.

[11] G. Camarillo and J. Rosenberg. The Alternative Network Address Types (ANAT) Semantics for the Session Description Protocol (SDP) Grouping Framework. RFC 4091 (Proposed Standard), June 2005.

[12] B. Carpenter. Architectural Principles of the Internet. RFC 1958 (Informational), June 1996. Updated by RFC 3439.

[13] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *SIGCOMM'03*, 2003.

[14] Oracle Corporation. Java ME: Mobile Media API (MMAPI); JSR 135. http://java.sun.com/products/mmapi/, 2010.

[15] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFC 5095.

[16] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631 (Informational), May 1994. Obsoleted by RFC 3022.

[17] Ericsson. P2PSIP Prototype.

[18] P. Eronen. *TCP Wake-Up: Reducing Keep-Alive Traffic in Mobile IPv4 and IPsec NAT Traversal*. January 2008.

[19] Google. Google talk. http://code.google.com/apis/talk/libjingle/important_concepts.html. Accessed, November 20, 2009.

[20] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. RFC 5382 (Best Current Practice), October 2008.

[21] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of The 5th International Workshop on Peer-to-Peer Systems*, 2006.

[22] T. Hain. Architectural Implications of NAT. RFC 2993 (Informational), November 2000.

[23] H. Haverinen, J. Siren, and P. Eronen. Energy consumption of always-on applications in wcdma networks. In *VTC Spring*, 2007.

[24] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. RFC 3027 (Informational), January 2001.

[25] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications*. John Wiley & Sons, 2004.

[26] C. Jennings, B. Lowekamp, E. Rescorla, S. Baset, and H. Schulzrinne. REsource LOcation And Discovery (RELOAD). Internet Draft (Standards Track), July 2008.

[27] C. Jennings, J. Rosenberg, and E. Rescorla. Address Settlement by Peer to Peer. Internet Draft (Standards Track), July 2007.

[28] A. Keränen. Host Identity Protocol-based Network Address Translator Traversal in Peer-to-Peer Environments. Master's thesis, Helsinki University of Technology, September 2008.

[29] S. Li and J.Knudsen. *Beginning J2ME^{TM} Platform: From Novice to Professional*. Apress, third edition, 2005.

[30] J. Mäenpää and J. J. Bolonio. Performance of resource location and discovery (reload) on mobile phones. In *Proc. of IEEE IPDPS*, 2009.

[31] J. Maenpaa and G. Camarillo. Study on maintenance operations in a peer-to-peer session initiation protocol overlay network. In *Proc. of IEEE IPDPS*, 2009.

[32] J. Mäenpää and G. Camarillo. Analysis of delays in a peer-to-peer session initiation protocol overlay network. In *Proc. of IEEE CCNC*, 2010.

[33] J. Mäenpää and G. Camarillo. Estimating operating conditions in session initiation overlay network. In *Proc. of IEEE IPDPS*, April 2010.

[34] Microsoft. Technical specification v20100218. Interactive Connectivity Establishment (ICE) Extensions, February 2010.

[35] Sun Microsystems and Motorola. Application Programming Interface: MID Profile. http://java.sun.com/javame/reference/apis/jsr118/, 2006.

[36] A. Müller, A. Klenk, and G. Carle. On the Applicability of Knowledge-based NAT Traversal for future Home Networks. In *Proceedings of IFIP Networking 2008*, 2008.

[37] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[38] P. Maymounkov and David Mazierès. Kademlia: A Peer-to-peer Information System Based on the XOR Metric, 2002.

[39] S. Perreault and J. Rosenberg. TCP Candidates with Interactive Connectivity Establishment (ICE). Internet-Draft (Standards Track), October 2010.

[40] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proc. of the 7th symposium on Operating systems design and implementation (OSDI '06)*, 2006.

[41] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM'01*, 2001.

[42] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Standards Track), February 2010.

[43] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Standards Track), February 2010.

[44] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[45] 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), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[47] Skype. http://www.skype.com.

[48] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[49] P. Srisuresh, B. Ford, and D. Kegel. State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs). RFC 5128 (Informational), March 2008.

[50] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), August 1999.

[51] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM'01*, 2001.

[52] R. Subramanian and B.D. Goodman. *Peer-to-peer computing: the evolution of a disruptive technology*. Idea Group Inc., 2005.

[53] International Telecommunication Union. Network grade of service parameters and target values for circuit-switched services in the evolving ISDN. ITU-T Recommendation E.721, May 1999.

[54] M. Westerlund and T. Zeng. The evaluation of different NAT traversal Techniques for media controlled by Real-time Streaming Protocol (RTSP). Internet-Draft (Informational), January 2010.