HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Electronics, Communications and Automation

Ari Keränen

Host Identity Protocol-based Network Address Translator Traversal in Peer-to-Peer Environments

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

Espoo, September 8, 2008

Supervisor: Professor Jörg Ott
Instructor: Jani Hautakorpi

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Ari Keränen
Name of the thesis:	Host Identity Protocol-based Network Address Translator Traversal in Peer-to-Peer Environments
Date:	September 8, 2008 **Number of pages:** 79 + 5
Faculty:	Electronics, Communications and Automation
Professorship:	S-38
Supervisor:	Prof. Jörg Ott
Instructor:	Jani Hautakorpi, M.Sc.

Network Address Translators (NATs) cause problems when peer-to-peer (P2P) connections are created between hosts. Also the Host Identity Protocol (HIP) has problems traversing NATs but, with suitable extensions, it can be used as a generic NAT traversal solution. The Interactive Connectivity Establishment (ICE) is a robust NAT traversal mechanism that can enable connectivity in various NAT scenarios. The goal of this thesis is to enable HIP-based NAT traversal using ICE and to evaluate the applicability of the approach by implementation and measurements.

We implemented an ICE prototype and tested it with different types of NATs. We used a network where two hosts were in different subnets and run ICE connectivity checks between them. The amount of messages and bytes sent during the process, and also how long the process took, was measured and analyzed. Based on the measurements, we calculated the overhead of using HIP with ICE for NAT traversal.

ICE was able to create a connection in all the scenarios, but sometimes using more messages and longer time than expected or necessary. We found reasons why too many messages are exchanged and presented solutions on how some of these redundant messages could be avoided. We also found out that while 4–5 connectivity check messages are enough in many scenarios, NATs with specific address mapping behavior can easily double the amount of needed checks. Still, the generated traffic bitrate is modest, and using shorter timeout values than what the ICE specification suggests can have a significant positive impact on performance. By using HIP with ICE, P2P programs can get an efficient NAT traversal solution that additionally supports security, mobility and multihoming.

Keywords: NAT, HIP, ICE, NAT traversal

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Ari Keränen
Työn nimi:	Koneen identiteetti protokolla -pohjainen osoitteenmuuntajien läpäisy vertaisympäristöissä
Päivämäärä:	8.9.2008 **Sivuja:** 79 + 5
Tiedekunta:	Elektroniikka, tietoliikenne ja automaatio
Professuuri:	S-38
Työn valvoja:	Prof. Jörg Ott
Työn ohjaaja:	DI Jani Hautakorpi

Osoitteenmuuntajat aiheuttavat ongelmia vertaisverkkojen yhteyksien luomiselle. Myös koneen identiteetti protokolla (HIP) kärsii osoitteenmuuntajien aiheuttamista ongelmista, mutta sopivilla laajennuksilla sitä voidaan käyttää yleisenä osoitteenmuuntajien läpäisymenetelmänä. Interaktiivinen yhteyden luominen (ICE) on tehokas osoitteenmuuntajien läpäisymenetelmä, joka toimii monissa erilaisissa tilanteissa. Tämän diplomityön tavoitteena on mahdollistaa HIP-pohjainen osoitteenmuuntajien läpäisy käyttämällä ICE-menetelmää, ja arvioida menetelmän toimivuutta implementoinnin ja mittausten avulla.

Implementoimme ICE-prototyypin ja testasimme sitä eri tyyppisten osoitteenmuuntajien kanssa. Käytimme mittauksissa verkkoa, jossa kaksi isäntäkonetta olivat eri aliverkoissa, ja suoritimme ICE-yhteystestejä näiden koneiden välillä. Mittasimme testeissä lähetettyjen viestien ja tavujen määrän sekä käytetyn ajan. Mittaustulosten perusteella laskimme myös arvion ICE:n ja HIP:in aiheuttamalle ylimääräisten viestien ja ajankäytön määrälle.

ICE onnistui luomaan yhteyden kaikissa testaamissamme tilanteissa, mutta käytti välillä enemmän viestejä ja aikaa kuin olisi tarpeen. Selvitimme työssä syyt ylimääräisille viesteille ja esitimme keinoja viestien määrän vähentämiselle. Saimme myös selville, että suuressa osassa tilanteista 4–5 yhteystestiviestiä riittää yhteyden luomiseksi, mutta tietynlaista osoitteenmuunnosta käyttävät osoitteenmuuntajat voivat helposti tuplata viestien määrän. Joka tapauksessa, yhteystestien luomat liikennemäärät ovat vähäisiä, ja käyttämällä lyhyempiä ajastinaikoja kuin mitä ICE spesifikaatio ehdottaa, voidaan ICE:n tehokkuutta kasvattaa merkittävästi. Käyttämällä HIP:iä ICE:n kanssa vertaisverkko-ohjelmat voivat saada käyttöönsä tehokkaan osoitteenmuuntajien läpäisymenetelmän, joka tukee myös yhteyden turvaominaisuuksia, mobiliteettia, sekä useita yhtäaikaisia verkkoliitäntöjä.

Avainsanat: NAT, HIP, ICE, osoitteenmuuntajien läpäisy

Acknowledgments

This Master's thesis has been done at Ericsson Research Finland, NomadicLab, for the Decentralized Inter-Service Communications (DECICOM) project.

Professor Jörg Ott has supervised my thesis and I want to thank him for all the valuable comments, inspirational ideas and well working communication. It has been, again, a joy to work with you.

I wish to thank my thesis instructor Jani Hautakorpi for the guidance and feedback during this process, for helping to keep the scope and time frame of the thesis manageable, and for all the interesting and enjoyable discussion on (and often off) topic.

I would also like to thank my other colleagues at the NomadicLab; especially Jan Mélen for helping with the Host Identity Protocol and clarifying numerous unexpected things ranging from kernel behavior to exotic features of C, Martti Kuparinen for helping setting up the prototyping network, Jukka Ylitalo and Kristian Slavov for generously helping me numerous times, and Gonzalo Camarillo for providing the resources for this work and managing a section where it has been fun to do research.

Finally, I want to express my deepest gratitude to Tanja, to my friends, and especially to my parents for all the support throughout my studies.

Otaniemi, September 8, 2008

Ari Keränen

Contents

Abbreviations

ALG	Application Layer Gateway
API	Application Programming Interface
BEET	Bound End-to-End Tunnel
DES	Data Encryption Standard
DoS	Denial of Service
ESP	Encapsulating Security Payload
GSM	Global System for Mobile communications
GPRS	General Packet Radio Service
HIP	Host Identity Protocol
HMAC	Hash Message Authentication Code
ICE	Interactive Connectivity Establishment
ICMP	Internet Control Message Protocol
ICV	Integrity Check Value
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	IP Security
IV	Intialization Vector
ITU	International Telecommunication Union
LAN	Local Area Network
MTU	Maximum Transmission Unit
NAPT	Network Address and Port Translation / Translator
NAT	Network Address Translation / Translator
P2P	Peer-to-Peer
PSTN	Public Switched Telephone Network

RFC	Request for Comments
RTO	Retransmission TimeOut
RTP	Real-time Transport Protocol
RTPC	RTP Control Protocol
RTT	Round-Trip Time
RVS	RendezVous Server
SDP	Session Description Protocol
SHA-1	Secure Hash Algorithm 1
SIMA	Simultaneous Multi Access
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
UA	(SIP) User Agent
UDP	User Datagram Protocol
VPN	Virtual Private Network
WLAN	Wireless LAN

List of Figures

List of Tables

Chapter 1

Introduction

When the Internet and the Internet Protocol (IP) version 4 [42] were devised in the late 1970's and early 1980's, an address space with 32 bits, resulting in more than 4 billion addresses, was assumed to be enough for giving all the hosts in the Internet a unique, globally routable address. This seemed more than enough at the time since the amount of computers was still quite limited and the Internet, or actually its predecessor ARPANET, was mainly used by universities and the military in the United States.

However, with the exponential growth of the Internet, already in the early 1990's it was becoming clear that there would not be enough IP addresses for all hosts that people would like to connect to the Internet. Since then, the problem has just become more severe because many homes today do not only have a personal computer that is used to connect to the Internet, but there may be multiple devices, such as laptops, game consoles, and cellular phones, that can be used to access the Internet, too. To fight the address depletion, work for a new version of the IP (version 6) with a larger address space was started, but also short term solutions were created. One of these solutions was Network Address Translation. [14]

With address translation, a private network can use certain blocks of IP addresses without consuming globally routable addresses. A device called Network Address Translator (NAT) is used to enable the hosts within a private network to communicate with hosts in the global Internet. To be able to conserve addresses, a NAT has only a small amount of public addresses to assign to the hosts in the private network it serves. For this reason, all the hosts in the private network may not be able to have a constant, globally routable (also known as public) address. Still, to be able to communicate with another host across the Internet, such an address is needed. NATs can provide addresses for the hosts behind them, but normally they do so only when a host in the NATed network initiates the connection. NATs were not an issue some years ago when majority of the Internet traffic was client-server based since dedicated servers are not usually behind a NAT. With the advent of peer-to-peer (P2P) mul-

timedia communication and file sharing programs, the focus is shifting from client-server to P2P communication. Since a peer, often working on a host that is in a private home or corporate network, is more likely to be behind a NAT, contacting it may no longer be trivial.

One example of a P2P communication system where NATs cause problems is the P2P Session Initiation Protocol (P2PSIP) [6]. In P2PSIP networks, the infrastructure is spread among the hosts (peers) participating in the network. Peers form an overlay network, a network on top of the IP network, and run together a distributed database algorithm using the overlay. The distributed infrastructure provides means for publishing e.g., contact information and querying it from the overlay. To be able to participate in the overlay, peers need to make connections to other peers in the network. Before such connections can be made between peers in different public and private networks, peers that are behind a NAT need to obtain a public address and make sure that the other peers can contact it. In addition to participating in the overlay, when the peers want to communicate directly with each other, they need to work their way through the NATs in between them.

For solving the connectivity problems that NATs cause, several different workarounds have been created and they are commonly called *NAT traversal* solutions. Many P2P applications have developed their own NAT traversal mechanisms for solving the problems but they usually work only in some NAT scenarios, may need relaying of all the traffic through a separate relay that all communicating hosts can connect to, or require manual configuration of the NAT from the user. Since the problem is so widespread, it would make sense to solve the issue in a lower layer in the protocol stack.

The Host Identity Protocol (HIP) provides an architecture [32] and a set of protocols [34, 26, 35] for creating secure connections that support mobility and multihoming between hosts. HIP is designed in a way that applications using the transport protocols provided by the operating system do not necessarily need to be even aware of it. If HIP is able to create a connection through the NATs between the peers, any application can use that path. This way, HIP can potentially decrease, or even completely remove, the need for application specific NAT traversal solutions and alleviate the problems that NATs pose to peer-to-peer applications desiring to communicate in the Internet.

1.1 Goal and Scope of the Thesis

Currently HIP has only limited NAT traversal capabilities [60]. The goal of this thesis is to extend these capabilities so that HIP can be used successfully in different NAT scenarios and to evaluate the solution's effectiveness by implementation and measurements. For this purpose, necessary extensions for HIP are presented and a suitable NAT traversal library is implemented. The performance of the library is evaluated and the applicability of the

approach to peer-to-peer environments is discussed. The main performance metrics that we are interested are: is our solution able to create an optimal path between two hosts, how long does this process take, and how much overhead in terms of sent messages and bytes does this require?

This thesis is focused on solving NAT traversal problems, so even if the same methods can be used to traverse also other types of middleboxes, such as stateful firewalls, generic middlebox traversal is out of the scope. Specifically, we focus on the traversal of legacy NATs which do not support HIP specific NAT traversal solutions. Also, the main focus of the implementation and evaluation is on the NAT traversal library and its integration to a HIP implementation is left for future work.

1.2 Structure of the Thesis

In this chapter we briefly introduced the problem area and defined the goal and scope of the thesis. Chapter 2 offers an overview of the background of the work and introduces related work. Chapter 3 presents the implemented NAT traversal solution prototype, implementation experiences, and how the implementation can work together with HIP. In Chapter 4 we present results of the measurements we performed on the prototype and analyze the results evaluating how well the prototype works in different scenarios. Chapter 5 contains discussion on using HIP with ICE as a NAT traversal solution for P2PSIP based on the results from the previous section and also introducing new ideas for improving ICE. Finally, in Chapter 6, we summarize our results and draw final conclusions from the work.

Chapter 2

Background

In this chapter we present relevant background and context for this thesis. We first introduce the concept of Network Address Translation and discuss some of the benefits and problems of it. Then, we present different NAT types and a way of classifying them by their behavior. Following sections introduce NAT traversal and different ways to do it. Some of the most relevant NAT traversal methods for this thesis are discussed in more detail. Finally, Host Identity Protocol and Peer-to-Peer Session Initiation Protocol are briefly introduced before summarizing the background of this work.

2.1 Network Address Translation

The networks connected to the Internet have different address realms. Within each realm a host has a unique address which can be used to route IP packets to it. In Network Address Translation, IP addresses are mapped from one address realm to another by changing the addresses of the packets passing by the realm border so that the addresses are valid within the address realm where they are routed into. [57]

The most common reason for performing address translation is the use of private range addresses [44] in a stub network. A NAT that resides on the border of the networks can assign public addresses for the hosts that need them, so they are able to communicate with other hosts having a public address.

The assignment and change of addresses performed by the NATs is meant to be transparent for the end hosts. For a host communicating with another host behind a NAT, the traffic seems to be coming from, and can be sent to, the address of the NAT. However, the host behind the NAT sees that the traffic was destined to it's private range address and can also use that as the source address.

In addition to the transparent routing and address assignment, a NAT should also change the contents of the Internet Control Message Protocol (ICMP) error messages that contain host addresses in their payload.

A host may end up behind a NAT for many reasons. Today, a common reason is that a host is in a home network that has only a single IP address provided by the Internet Service Provider (ISP). An example of this is the laptop and PC 2 in Figure 2.1. There can be also multiple layers of NATs as for the PC 3 in the same figure. Similarly, a mobile device connected to the Internet using a cellular network may have to connect through a NAT that is in the mobile provider's network. If any of these hosts want to communicate with the World Wide Web (WWW) server or PC 1, they need to obtain a public address on the NAT. Also, if PC 1 would like to connect to any of the devices behind a NAT, it would need to use an address that the NAT with a public address has assigned for the host. [15]

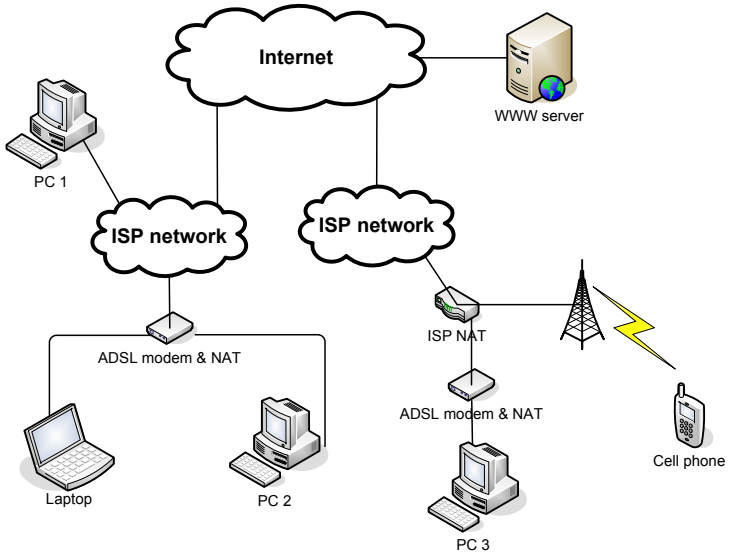

Figure 2.1: Example NAT scenarios in the Internet

There exists several different types of NATs, but we focus on the traditional NATs [55] which are the *basic NAT* and the *Network Address and Port Translator* (NAPT). These two types make up the vast majority of the deployed NATs in the Internet with NAPT being the most common one. Both of them are referred to as NAT unless there is a need to make a distinction between them.

Both type of NATs work in three phases. First, they need to assign addresses for the hosts in the private network they are serving. This can be done either in a static or dynamic way and is called address binding. In static assignment, the binding is pre-configured and stays

the same for the lifetime of the NAT operation or when it is changed by configuration. Dynamic binding happens when an outbound session is started by a host in the private network: when the NAT detects a new session, it chooses one of its currently free public addresses to be used for all packets belonging to this session. During the address lookup and translation phase, outgoing packets belonging to the same session will have their source address changed so that they appear to originate from the NAT, and incoming packets' destination address is changed so that they are properly routed to the originator of the session. Finally, if the session that created a dynamic binding is ended, the binding may be terminated to free the public address for re-use.

2.1.1 Basic Network Address Translator

A basic NAT has a block of public IP addresses that it can assign for the hosts in the private network. When a new IP packet is sent by a host (IP address 10.1.0.2 in Figure 2.2) in the private network to a host in the public network (address 138.76.29.7), the NAT takes one of the IP addresses reserved for NATing (198.76.28.1) and binds that for the host in the private network. Now, for the remote host, the connection looks like it's coming from the address of the NAT.

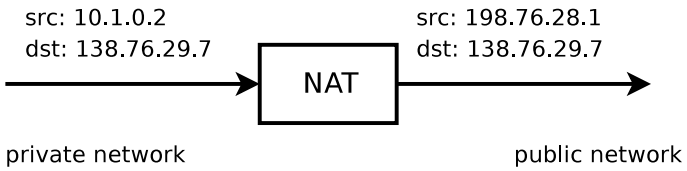

Figure 2.2: Basic NAT with outbound connection

When the remote host sends a packet back to the public address the NAT made the binding for, the NAT translates the destination address of the packet to point to the host in the private address as shown in Figure 2.3. For the host in the private network the source address of the packet is still the same where it sent the packet to in the first stage but the destination address is its private range address.

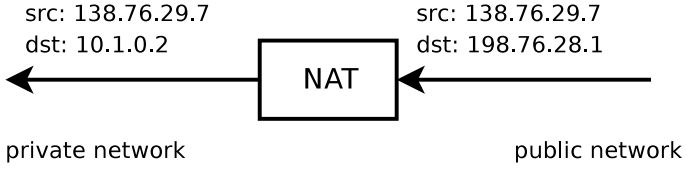

Figure 2.3: Basic NAT with return traffic

If another host in the private network wants to communicate with a host in the public network at same time, the NAT has to allocate another public IP address (in this case 198.76.28.2) for it as shown in Figure 2.4. Because each host initiating connections outside of the private network gets a unique public address, if all the hosts in the private network create such connections at the same time, basic NAT does not save IP addresses.

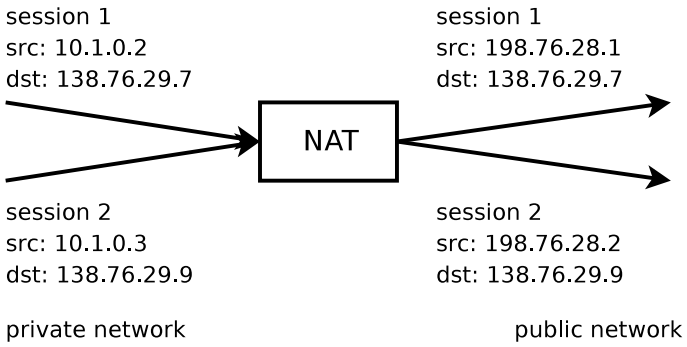

Figure 2.4: Basic NAT with two simultaneous outbound connections

2.1.2 Network Address and Port Translator

A NAPT works in a similar way as the basic NAT, but instead of using just IP addresses, it also uses transport layer, e.g., Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), ports to multiplex connections to a single public IP address. This way the NAPT needs just one public IP address for serving multiple hosts in the private network. A combination of an IP address and a transport layer port is called a *transport address*. In Figure 2.5 two host from addresses 10.1.0.2 and 10.1.0.3 want to start TCP connections using a local port 2000 to web servers (port 80) in addresses 138.76.29.7 and 138.76.29.9. In this example, the NAPT uses only a single public IP address, 198.76.28.1, for both of the sessions, but reserves the port 3000 for the first session and the port 3001 for the second session.

With incoming packets the NAPT translates the port numbers in addition to the addresses as shown in Figure 2.6. The incoming packet to port 3000 belongs to the first session and is translated to address 10.1.0.2 and port 2000, on the other hand, the packet to port 3001 belongs to the second session and is translated to address 10.1.0.3 and port 2000.

Because ICMP messages do not use any transport layer, a NAPT that allows ICMP queries from private network to public side must multiplex them by the query identifiers. When a response to a query arrives to the NAT, it checks if a query with that identifier was recently sent by a host in the private network and send the response to that host's IP address.

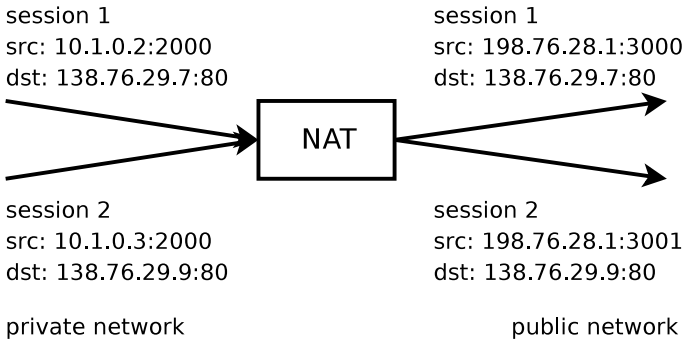

Figure 2.5: NAPT with two simultaneous outbound connections

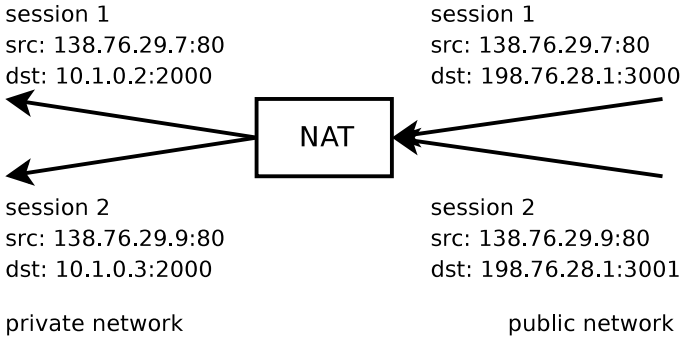

Figure 2.6: NAPT with two simultaneous sessions with incoming packets

2.1.3 Address Unbinding

To be able to re-use the public addresses, a NAT has to remove the binding when a connection is no longer in use. A NAPT can observe the TCP packets it translates, and after a connection tear down or reset, it can remove the binding. TCP has special flags in all the packets that tell if a packet is starting a new connection or ending an existing one. However, plain IP (in case of basic NAT) and UDP do not have such a connection setup and tear down phases, so NATs have no easy way of knowing when it is safe to remove the binding. A NAT could keep the binding alive as long as it has free resources left, but often a timer is used and if there is no traffic on the path for certain amount of time, the binding is discarded. Also, a TCP connection may end without proper connection tear down packets if one or both of the hosts, e.g., lose connection or reboot. Because of this, NATs also needs to remove TCP connections that have been idle for a long time. However, the time a TCP connection can be idle without losing the NAT state is often much longer than for UDP. [2, 16]

2.1.4 Benefits of Network Address Translation

The ability to conserve IP addresses is a clear benefit especially for the NAPT type of NATs, however, NATs provide also other benefits such as avoiding network renumbering, hiding network topology, and some level of security.

By using private address range addresses for the hosts, a network can hide topology changes outside of its domain because they will affect only the NAT's address. This makes, e.g., changing ISP easier, since normally the address block that was given by the previous ISP is not likely available from the other ISP due to IPv4 address assignment guidelines. The topology hiding also works the other way around since all the traffic from the network behind a NAT seems to originate from the NAT device. Because the hosts in the public network do not know the real address of the host initiating the connection, NATs can be seen to increase users' anonymity. In addition, because a NAT blocks inbound connections unless they are allowed by a static mapping, it works in a similar way as a stateful packet filtering firewall. Nevertheless, since NATs are made for address translation rather than packet filtering, relaying on such a behavior is not recommendable. For example, NAT bindings can change dynamically over time and a new outbound session may allow inbound connections to a neighboring host. [18, 14]

2.1.5 Problems Caused by Network Address Translation

Using NATs in the network is by no means free of problems and this was already mentioned by the authors of the first Request For Comments (RFC) defining NAT [14]. Egevang and Francis note especially that NATs break certain applications and that the appropriateness of NATs as a solution, even as short term, to the problem of address depletion should be determined by implementation and experimentation.

First of all, NATs break the Internet end-to-end model by introducing critical state in the network: if a NAT device fails, all the connections that were using it will fail too because, even if the NAT is replaced, the mappings it used for the connections are gone. Adding redundant NATs and replicating the state information between them can help to some extent, but it also creates new problems with communicating the state information. [18]

Another problem is that hosts will not have a consistent view on their address. A host in the private network thinks it is using the private range address whereas hosts communicating with it through the public network see the public address given by the NAT. Problems arise if the host communicates the private range address in an application layer protocol message. If the private address is communicated to other hosts outside of the private network, the address will not be routable or may even be routed to a wrong host in the other host's network. Examples of protocols that carry addresses in application layer messages are the File Transfer Protocol (FTP) [43] and the Session Initiation Protocol (SIP) [51].

The problem with application layer communicating private range addresses can be solved to some extent with an Application Layer Gateway (ALG) working together with the NAT [20]. The ALG can inspect the application layer messages and change occurrences of the private range address with the address that the NAT uses for the session. For this to work reliably, the ALG needs to know the details of the upper layer protocol and therefore ALGs will likely not work with less known protocols or protocols that were created after the ALG was programmed. A generic ALG can try to blindly scan any occurrences of the private range IP addresses, but this can lead to even more problems [49].

Some NATs are also known to perform badly with fragmented IP packets. Fragmentation may occur in the network if the Maximum Transmission Unit (MTU) of an on-path link is smaller than the size of the IP packet sent to that link. If a fragment arriving at a NAT does not have a complete TCP or UDP header, the NAT can not make the address translation and may choose to simply drop it [10]. Even if a NAT tries to assemble the fragments into a full IP packet, it may choose to ignore fragments coming out-of-order and fail assembling them. Also, fragmentation may even corrupt unrelated sessions: two hosts behind the same NAT may send data to the same host in the public network and both can have packets fragmented which happen to use the same fragmentation identifier. Because fragments after the first segment do not have the TCP/UDP port information, the target host sees just fragments with the same fragmentation identifier coming from the same (NAT's) host address and cannot determine to which session the fragments belong to. [55]

A whole different set of problems is introduced by NATs to IPsec connections. For example, if the IP addresses of the packets are secured against tampering with an IPsec Authentication Header, a NAT that changes the addresses results in packets being dropped as invalid by the receiver. Also, IPsec Encapsulating Security Payload (ESP) will not pass NATs properly since the TCP/UDP headers are encrypted and NATs need to change at least the header checksums due to change in the IP address. Since many NATs require transport layer identifiers for multiplexing the traffic, plain ESP over IP will not pass such NATs but it must be run over a transport protocol. In addition, even if ESP is run over a transport protocol, since it encrypts the contents, ALGs can not fix any private range addresses that are used by the application layer protocols. Also other protocols that secure connections, such as Transport Layer Security (TLS) [13], suffer from the same problems. [1]

Yet, maybe the biggest problem with NATs in peer-to-peer environments is that they often block incoming connection attempts altogether. As described in the previous sections, a NAT can only forward data to a host in the private network if it has an active mapping between the public and the private address (possibly including the transport layer identifiers). Even if there is a mapping for a public address to an address in the private network, the public address may be valid for only some hosts in the public network, or even only for

some transport layer ports. Whether an incoming packet is accepted or not in such a case depends on the behavior of the individual NAT. Unfortunately, a recommended behavior has not been standardized until recently [2], so there exists a lot of legacy NATs whose behavior can be unexpected, and often bad, for peer-to-peer traffic. [56]

Once IPv6 is widely deployed, there is no longer such a need to conserve addresses, but some of the features of NATs, or even IPv6 NATs, are likely to persist in the Internet. For example, for an IPv4 capable node to be able to communicate with IPv6 nodes, some address translation will be needed, and a NAT performing IPv6-IPv4 translation can be a solution for this [9]. Also, because of the benefits presented in Section 2.1.4, some networks may choose to use NATs even if they are not necessarily needed. However, there are also ways to perform many of these features without a NAT in an IPv6 networks and they are described in [11].

2.1.6 Ambiguity of Topology Caused by NATs

In addition to the problems described in the previous section, NATs can also cause ambiguity in the network topology. As shown in Figure 2.1, there may be multiple layers of NATs between the hosts. Even though we have referred to the other side of the NAT as the private and the other as the public side, the public side may not necessarily be the same for all hosts. Also, even if a host uses the same source address, it may appear to be at different addresses depending on the receiver of the data. Example of this is shown in Figure 2.7.

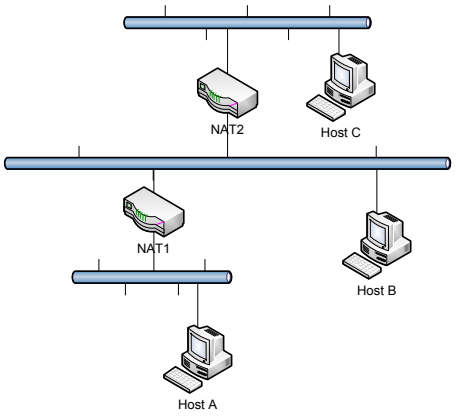

Figure 2.7: NAT scenario with multiple layers of NATs

In the figure, for host B, host A seems to come from the address of NAT1, which is in the private network for host B. Yet, for host C, both A and B appear be at the address of NAT2 [10]. Still, for the sake of simplicity, we continue to refer to the side of the NAT where the inbound connections are coming from, as the public side.

2.2 NAT Classification

The operation of traditional NATs seems quite straightforward: once there is an address (and possibly a transport layer port) reserved on the NAT's public side, the hosts on the public side can use that binding for sending data back to the host in the private network. However, NAT implementations differ in the way how other remote hosts than those to which the connection was initially made to, are able to use the binding. Also, there are differences on how NATs handle simultaneous connections started from the same private address and port but destined to a different remote host. These differences can be described with NAT mapping, filtering, and port assignment behavior which are defined in [2] and summarized in the following sections.

2.2.1 Mapping Behavior

The first outgoing packet through a NAT from a private address and port makes the NAT assign a public address and port that the traffic to the other direction can use by creating a mapping between the internal IP:port and external IP:port tuple. Depending on the mapping behavior of the NAT, this same mapping may or may not be used by other simultaneous sessions.

In case of *endpoint-independent mapping*, as long as the source address and port of the host in the internal side of the NAT do not change, the same external tuple will be used for all outbound connections, regardless of the destination address or port. An example of this is shown in Figure 2.8. Here, both sessions have the same source address and port (10.1.0.2:2000) but different destinations. The NAT uses the same external address and port (198.76.28.1:3000) for both of the connections.

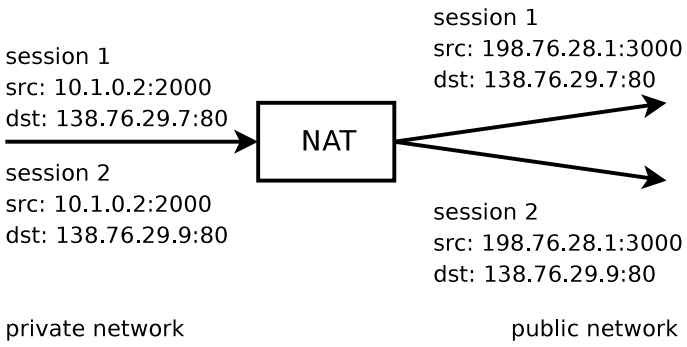

Figure 2.8: Example of endpoint-independent mapping

If the mapping behavior is *address-dependent*, the two sessions in the previous example will have a different external port given by the NAT. In Figure 2.9, the source and destination addresses and ports of the endpoints are the same as before, but the NAT uses port 3001 instead of 3000 for the second session. If the NAT has a pool of public IP addresses (in case of basic NAT), it may even give the second session a different public address. If the destination address of the second session would have been the same as for the first session, the NAT would have re-used the first session's mapping.

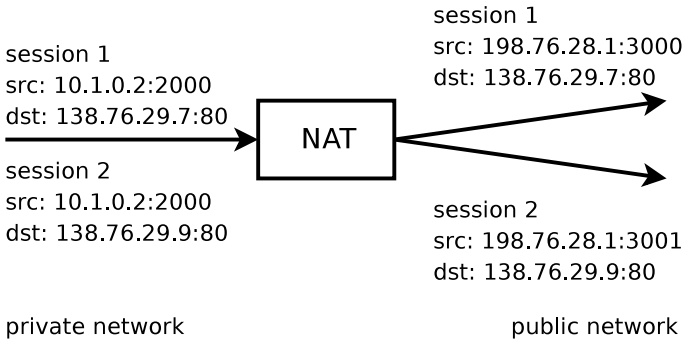

Figure 2.9: Example of address-dependent mapping

An even more fine grained version of mapping is the *address and port-dependent mapping*. Here, even if the destination address of the second session is the same as for the first session, but the destination port is different, the NAT will create a new mapping for the session. In Figure 2.10, the first session remains the same as in the previous examples, but the second session is now destined to the same host. However, since the NAT is now address and port-dependent, it maps it to a new port (3001) on the public side. Both previous types of mapping behavior would have re-used the first session's mapping.

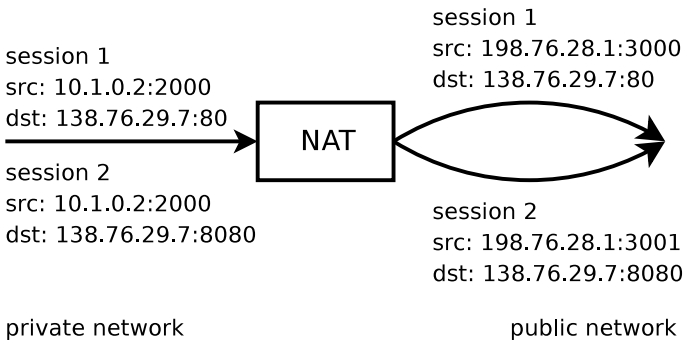

Figure 2.10: Example of address and port-dependent mapping

Out of these three, the endpoint-independent mapping behavior is recommended by [2] since it is the best for many of the NAT traversal techniques. Also, in case the NAT has a pool of public IP addresses, it should assign the same address for sessions originating from the same host in the private network. Otherwise protocols that assume all connections coming from a single host to be using the same source address will fail.

2.2.2 Filtering Behavior

The way how NATs create mappings does not necessarily correlate on how they allow incoming packets from different remote hosts through those mappings. Instead, this is decided by the NATs filtering behavior.

Once a mapping is created on the NAT by a host in the private network sending an outbound packet through it, a NAT that performs *endpoint-independent filtering* allows any host on the external side of the NAT to use the same mapping, i.e., public address and port on the NAT, for sending packets back to the host in the private network.

Address-dependent filtering is a more strict filtering behavior where a mapping can only be used by hosts in the public network that have received packet(s) from the host in the private network using that mapping. So, even if a mapping on the NAT exists, the NAT will drop all incoming packets that are coming from an unknown address. The host in the private network must first send a packet to that address using the same source port (and therefore the same NAT mapping) that initially created the mapping. The destination and source port of the remote host does not matter, so the remote host can send traffic back from any port as long as the return traffic uses the same mapping on the NAT.

The most strict filtering behavior, *address and port-dependent filtering*, behaves similarly to the address-dependent filtering, but now also the port of the remote host counts. That is, the remote host must send data back from the same port where the host in the private network has previously sent data to.

Endpoint-independent filtering is the recommended way by [2] to maximize application transparency and allow NAT traversal without the need for a relay.

2.2.3 Port Assignment Behavior

A NAT may assign arbitrary ports for sessions from its reserve of free ports, or it may make the decision based on the port the host in the private network used as the source port. A *port preserving* NAT tries to use the same port in the public side as the host in the private side selected. If the port is already used by a different host and session, the NAT may select a different IP address (if it has more than one) for the new session, use a different port for the new session, remove the old session, or even use the same port for both of the sessions.

Especially the last option, called *port overloading*, can lead to unexpected behavior and [2] recommends that is must not be used.

In addition to the port number, also the port number parity can be preserved by NATs. A NAT using *port parity preservation* maps even internal ports to even external ports and odd ports to odd ports. For example, specification of Real Time Protocol (RTP) and its control protocol, RTPC, encourage that RTP uses an even port number and RTPC an odd port number [53]. For this reason, [2] recommends to use port parity preservation.

Since the RTP specification also recommends that the port number of RTPC would be the next higher port, and that layered encoding application should use contiguous port numbers, a NAT employing *port contiguity* can attempt to preserve this assignment. This can be done either by port preservation or by *sequential assignment* in which public port numbers are given in sequence with hope that the connections are started in the order from smallest port number to largest. Because some ports may be already be preserved, and therefore unavailable for port preservation leading to non-deterministic behavior, and how well sequential assignment works depends on the behavior of the application, [2] makes no recommendations on this behavior.

2.2.4 Hairpinning Behavior

If a NAT supports *hairpinning*, two hosts behind the same NAT can communicate with each other using the external address the NAT has assigned for either one of them. This is illustrated in Figure 2.11 where host B has created a NAT binding for private address B:b and public address Bn:bn by sending a packet from B:b to any address on the public side of the NAT. Host A then sends a packet to host B's public address, which is routed to the NAT that works as a gateway for the network. The NAT should, if it supports hairpinning, create a binding with a public address (An:an) for host A and send the packet to host B so that it appears to come for host A's public address. [2] recommends that NATs support this behavior.

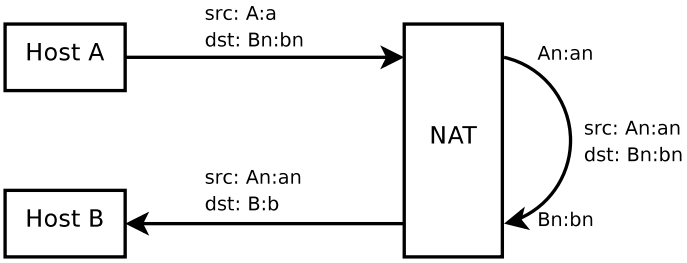

Figure 2.11: NAT hairpinning behavior

2.2.5 Mapping Refreshment

Another important feature of a NAT is how long it waits before discarding the state for an idle connection and how that mapping state can be refreshed.

The timer values for discarding the idle connections vary from implementation to another. For UDP, a minimum of 2 minutes and a recommended value of 5 minutes are given by [2]. For TCP, an established connections mapping should not be discarded before at least 2 hours and 4 minutes have passed since the last packet. However, especially for UDP, much smaller values have been discovered in practice and for example [46] recommends sending keepalive packets every 15 seconds unless there is data traffic. Also, with some NAT implementations, even idle TCP connections may lose the NAT state after just a few minutes [17].

The mapping state can be refreshed by sending packets that use the mapping through the NAT, but NATs differ on whether inbound traffic coming from the external side of the NAT can refresh the mapping state. Letting inbound traffic refresh the state allows a misbehaving application or attacker keep the mapping alive and send packets to the host in the private network. Also, doing this on many ports can potentially over time exhaust free ports on the NAT. For this reason, many NATs may choose to let only outbound traffic refresh the mapping state. [2]

2.2.6 Different Types of NATs in the Internet

According to measurements performed on TCP by Guha et al. [17], in 2005, 70.1% of the NATs had endpoint-independent mapping behavior and with 27.4% the behavior was address and port-dependent[1]. For filtering, the most common behavior was address and port-dependent (81.9%) with address-dependent (12.3%) and endpoint-independent (5.8%) behaviors being much less common.

As the authors of [17] point out, the situation has likely changed since the publication of the paper due to rapid replacement of NAT devices. The recommendations for NAT behavior for both UDP [2] and TCP [16] should make both endpoint-independent mapping and filtering more common. Also the NAT vendors have incentives to adhere to the recommendations since the peer-to-peer applications work with higher probability and require less configuration effort for the user.

[1]The paper presents 23.5% to have address and port-dependent behavior, but in addition, 3.9% of the NATs created a new mapping for all new connections, working essentially like the previous type

2.3 NAT Traversal

As described in Section 2.1.5, NATs prevent hosts from the public side of the network to connect hosts in the private network. To overcome this problem, several NAT traversal techniques have been developed. Most of them work by creating an address binding to the NAT so that the incoming connections can be forwarded to the right host. The binding can be created explicitly using a middlebox signaling protocol or implicitly by making the incoming traffic look like a reply to a request sent by the host in the private domain. Another option is to tunnel the traffic through the NAT and use different addressing, than the one the NAT provided, at the ends of the tunnel.

Middlebox signaling protocols such as UPnP [63], SOCKS [30] and MIDCOM [58] perform NAT traversal by communicating with the network's gateway that works as the NAT. They can be used to request the NAT to open and forward certain ports to the hosts in the private network. These methods require that the NAT supports the protocol, but they do not require any support from the peer that wants to contact the host behind the NAT.

On the other hand, Teredo [21] can provide hosts an IPv6 address that is able to traverse NATs using IPv4 and UDP encapsulation. Teredo first probes the type of the NAT that serves the private network with help from a Teredo server, and if the NAT is suitable for Teredo, peers can contact the host in the private network either directly or with help from the Teredo server.

Different methods of UDP and TCP hole punching [15, 17, 4] also work without explicit help from the NAT by creating a proper mapping on the NAT with normal UDP or TCP packets. Session Traversal Utilities for NAT (STUN) is one protocol that can be used for hole punching and learning the address NAT has given for the host.

One way to make the traffic look like client-server communication, and therefore work through NATs, is to relay the traffic through a server in the public network. When the hosts individually contact the relay, both of their NATs create the needed mapping and state for the return traffic. Then, the relay can use the same NAT mapping for sending the traffic coming from one host to the other. Traversal Using Relays around NAT (TURN) [48] provides a protocol that can be used for relaying the traffic this way.

Finally, interactive Connectivity Establishment (ICE) uses STUN and TURN to provide a complete NAT traversal solution. The NAT traversal methods that are most relevant for this thesis are presented in the following sections.

2.3.1 UDP Hole Punching

Ford et al. presented a UDP hole punching method in [15]. In this method, hosts need to connect a well-known rendezvous server which can relay messages between them. When

a host contacts the rendezvous server, it tells the server the transport layer address it sends the packet from. The server also registers the address where the host appears to come from by checking the source address of the incoming packet. If these two addresses are not the same, the host is behind a NAT.

When another host (host B) wants to create a connection to the first host (host A), it can ask for the private and public address of host A from the rendezvous server. When the server replies with this information, it can also tell A that B wants to contact it and tells B's private and public address to A. Now both hosts know each others' private and public address and can start sending UDP packets to both of them. If either one of the public or private addresses is able to deliver the packet to the other host, the other host can reply to it and hosts can use that address pair for further communication.

If both of the hosts are behind the same NAT, in the same subnet, the private address pair will work. If the NAT supports hairpinning, as described in Section 2.2.4, also the public address will work and hosts can choose to use either one of the paths. If the hosts are behind different NATs, the private addresses are either invalid or point to another (wrong) host in the same network and do not produce a successful response. Instead, an outgoing packet from host A to B creates a binding on the host A's NAT. If the host B has not send its own packet to host A's NATed address, host B's NAT will likely drop the packet. Instead, when host B eventually sends its own message to host A's NATed address, the host A's NAT already has a binding in place for the message, and it is delivered to host A.

Ford et al. state that as long as the NATs are well-behaving, i.e., map outbound connections from the same source address to the same public transport address, either one of the connection attempts on the public address eventually succeeds and they report a success rate of 82% for this approach from their measurements.

Similar technique can be used for TCP hole punching but it works with a lower success rate (64%). More advanced tricks can increase the probability for TCP hole punching, but they require e.g., address spoofing for the rendezvous server and access to raw sockets for the clients.

2.3.2 STUN

Session Traversal Utilities for NAT (STUN) is a protocol that can be used as a tool with other protocols for NAT traversal. The base specification [49] defines a generic packet format which the other protocols using STUN for NAT traversal, called STUN usages, can use and extend. With basic STUN a host can discover the binding that a NAT (if there is one or more) has created for it and also keep that binding alive. In addition, two authentication mechanisms: long-term and short-term, are defined.

STUN works as a client-server protocol with two different types of transactions. In a request-response transaction the client sends a request to the server and server sends back a response. Also, either client or server can send indication messages to the other party, but no response is generated for those.

All STUN messages start with a fixed, 20 byte header that contains a randomly selected 96 bits long transaction identifier used to correlate a response to a request. The header also indicates the method and class of the transaction. Class can be either a request, success response, failure response, or indication. The base specification defines only a single method: Binding. This method can be used for discovering the binding a NAT has allocated for the client (using the request/response class of messages) or keeping the binding alive (with the indication class of messages).

An example of STUN operation is shown in Figure 2.12. First, the STUN client sends a *Binding Request* to the STUN server. The client is in a private network and has the local address 10.1.0.2 and the request is sent from the port 2000. The NAT creates a new binding for the STUN client's address and port and uses the address 198.76.28.1 and port 3000 on the external side. When the STUN server receives the request, it sends back a response and includes the address where the request seems to come from in the response (the MAPPED attribute[2] in the figure). The NAT translates the destination address of the response packet and forwards the response to the STUN client. The STUN client can now learn the address the NAT used for it from the mapped address attribute. This address is called *reflexive transport address*. To keep the binding alive, the STUN client can also send periodically *Binding Indication* messages to the STUN server.

Some STUN usages require that the STUN messages are sent using the same address and port as some other protocol, so the STUN messages need to be demultiplexed from them. The STUN header has a fixed 32 bit long magic cookie value to help with demultiplexing. Also, the first and last two bits of the STUN header's first 32 bits are always zeroes. A STUN message may also have a FINGERPRINT attribute, with a CRC-32 checksum over the whole message, to make false positives less probable.

Because STUN messages can be sent over unreliable transport protocol, such as UDP, there is a need for reliability mechanism. If there is no answer to a STUN request after waiting a time of RTO (Retransmission TimeOut), the request is retransmitted. The RTO is an estimate of the round-trip time (RTT), but the specification recommends a minimum value of 500ms. For the next retransmission the time to wait is always doubled resulting in exponential back-off. It is recommended that at maximum 7 retransmits are tried before giving up. Since binding indications do not generate an answer, there is no similar reliability mechanism for them.

[2]The real full name of the attribute is XOR-MAPPED-ADDRESS

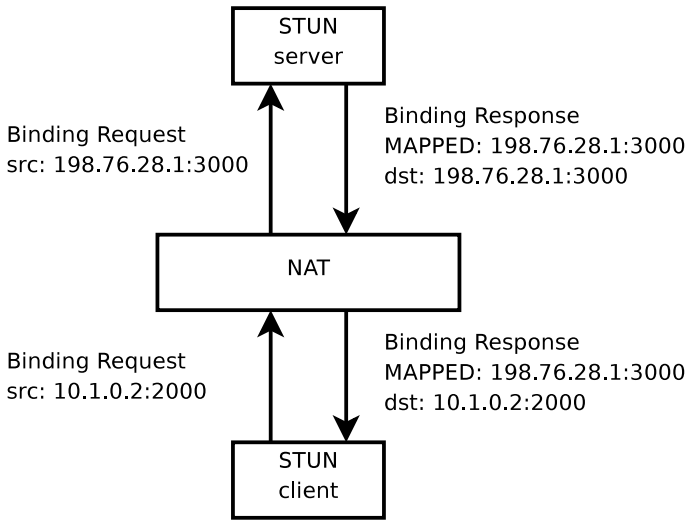

Figure 2.12: STUN binding request example

2.3.3 TURN

STUN's extension, Traversal Using Relays around NAT (TURN) [48] defines how STUN can be extended to provide data relaying functionality. If a host is behind a NAT whose mapping behavior is address or address and port-dependent (see Section 2.2.1 for definition), the address that was given by the NAT for communicating with the STUN server does not work with other hosts. To be reliably reachable by other hosts, a host behind such a NAT can register itself to a TURN server in the public network which can provide a transport layer address that is not behind a NAT. The TURN server can then forward all the traffic that is received on that address to the client using the NAT mapping client created when it first registered to the server. The address on the TURN server is called *relayed transport address*. The client can also send data to the TURN server and the server forwards the data from the relayed transport address to another host.

A typical deployment with a TURN server is shown in Figure 2.13. The TURN client is separated from the TURN server by a NAT and the client is on the private side. The client uses the transport address 10.1.0.2:2000 to communicate with the TURN server. The NAT between the client and the server mapped this address to 192.0.2.1:7000 on the public side. If the NAT's mapping behavior is not endpoint-independent, only the TURN server is able to use this mapping. The TURN server's address 192.0.2.15:3478 can be used for both sending commands to the TURN server and relaying data to other hosts. For the other hosts, the data sent by the TURN client via the TURN server seems to come from the address (192.0.2.15:9000) that the TURN server allocated for the client.

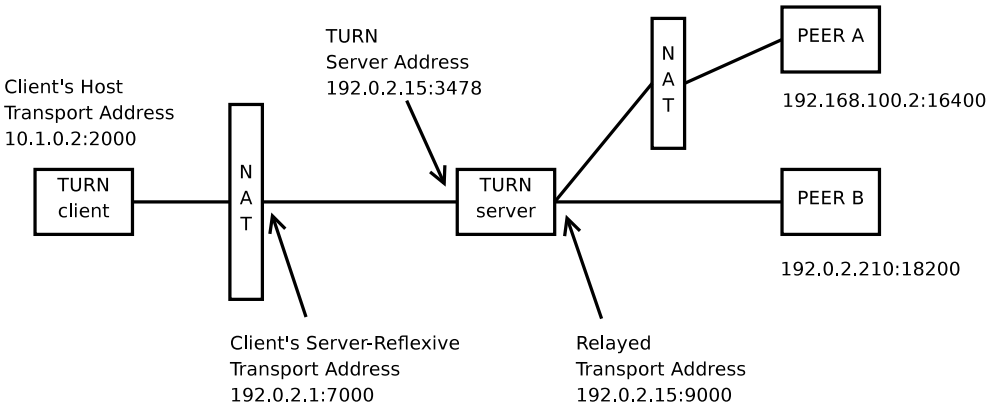

Figure 2.13: Example TURN deployment

An address allocation at the TURN server is obtained using the *Allocate request* STUN message. The client can ask in a request for certain transport protocol and lifetime for the allocation. To a successful request, the server replies with an *Allocate response* that contains the address the TURN server allocated, the server reflexive address of the client, and the lifetime it chose for the allocation. However, before any peer can send data using that allocation, a permission for that peer must be created. A client can create a permission for a peer by sending any data to that peer via the TURN server.[3] After this, the peer can send data back via the TURN server from the same address and using any source port.

There are two ways to exchange data between the TURN client and the peers using the TURN server. The client can wrap the data it wants to send in a *Data indication* message which also contains, in a STUN attribute, the destination transport address of the peer. When the server receives the Data indication from the client, it extracts the data and sends it to the address that was given by the client. When the server receives data back from a peer, it can similarly wrap it in a Data indication and attach a STUN attribute with the sender transport address to the message before sending it to the client. Because this approach can substantially increase the overhead due to the STUN wrapping, the client can also reserve a channel for communicating with a certain peer.

A channel is reserved using a *ChannelBind request* which contains the peer address where a new channel should be bound to and an unallocated channel number. The server bounds the channel number to the peer transport address and acknowledges the binding with a *ChannelBind response*. Then, the client can send data to that peer via the TURN server using *ChannelData* messages which contain just a simple 32 bit header before the actual data. The ChannelData header contains a 16 bit channel identifier and 16 bit field with

[3] An empty Data indication message to the TURN server is also enough to create the permission

length of the data. The server strips off the ChannelData header and forwards the data to the address where the channel was bound to. For the return traffic, the server inserts a similar header for the peers that have such a channel assigned and sends the incoming data, prefixed with a ChannelData header, to the client. A client can simultaneously communicate with other peers using Data Indication messages and/or other channels. In both methods the peer does not need to use any TURN encapsulation, or even be aware that TURN is used, but only the data between the TURN server and TURN client is encapsulated in Data Indication messages or TURN Channels.

An allocation expires when the lifetime that the server used in the Allocation response has passed. To be able to use the allocation longer, the client must refresh the allocation using *Refresh requests*. A client can also destroy an existing allocation by requesting a zero lifetime for the allocation in the Refresh request.

2.4 Interactive Connectivity Establishment

Interactive Connectivity Establishment (ICE) [46] is a robust NAT traversal solution that combines hole punching and relaying with a whole set of methodology and optimizations. It has been under work since 2003 [45] and will likely become an RFC during 2008. ICE is mainly specified as a NAT traversal solution for SIP-initiated sessions, but it is also applicable to other session oriented protocols that need connectivity in networks with NATs [47, 52].

2.4.1 Basic Operation

The main idea of ICE is quite simple: when a host wants to communicate with another host, it gathers a set of transport addresses that it thinks it might be reachable from. These address-port pairs, called candidates, are communicated to the other host using a signaling protocol like SIP. When the other host is informed about the incoming connection attempt, it also gathers a set of candidates, announces them to the first host and then both hosts try to connect to each others' candidates. When these connectivity checks succeed on some of the candidate pairs, those pairs are marked as working. After the checks, the best working candidate pair is selected and further communication can use the path between those transport addresses.

The local addresses and ports that are gathered can be from the physical network interfaces (like Ethernet or WLAN) or, e.g., virtual interfaces (like Virtual Private Network (VPN) tunnels). These local candidates are called *host candidates*. If some of the host's network interfaces are behind a NAT, hosts on the other side of the NAT see the NAT's address when communicating with the host. ICE uses STUN to determine addresses and

ports the hosts in the globally routable Internet see. This is done by sending binding request messages from the hosts candidates to a STUN server that is known to be on the other side of the NATs (if there are any), asking for the address it sees the request is coming from. The STUN server can reply using the same path and the requesting node learns the addresses and ports the NATs allocated to it. Candidates that are learned this way from an external server are *server reflexive candidates*.

As described in Section 2.2, some NATs do not allow a reply to come from a different address and/or port than where the request was sent to. If the host that is doing the address gathering is behind such a NAT, only the STUN server can use the discovered return path and packets from all other source addresses (and ports) are dropped. Therefore, in some cases, a relay is needed. For this purpose, TURN can be used to obtain relayed addresses from a TURN server. If the TURN server is reachable from both of the hosts, they can communicate using it even if the NATs prevent direct communication. These *relayed candidates* that the TURN server provides are also announced to the other party for the connectivity checks.

Before announcing the candidates, they are locally prioritized. The prioritization algorithm works in a way that similar candidates get similar priorities and candidates containing less hops, i.e., less NATs and/or relays, are preferred. Within those limits the hosts can substantially effect the priorities using, e.g., local policies. After prioritization, the candidates are coded in a way that is suitable for the signaling protocol (e.g., using Session Description Protocol [19] attributes in case of SIP) and they are sent to the other host. When both hosts have exchanged their prioritized candidates, the priorities of the local candidates are combined with the priorities of the remote candidates and a checklist is formed with higher priority candidate pairs on top of the list.

Connectivity checks are done sequentially in the order the candidate pairs are in the checklist. In addition, if a check is received on some candidate pair that is not validated yet, a check for that pair is scheduled to happen before other candidate pairs in the checklist. This is called a *triggered check* and it is done to speed up the connectivity check process. If more than one such check is received on different candidate pairs before the triggered checks are sent, they are stored in the *triggered check queue* which is processed before the normal checklist.

During the connectivity checks STUN messages are sent between the candidates, and if a request-response exchange succeeds for some pair, a working candidate pair has been found. If hosts are behind NATs that do address-dependent filtering, the first request is likely dropped by the peer's NAT as depicted in Figure 2.14. However, since the other host is performing the same check roughly at the same time, the request sent by it looks to the NAT like a response to the request the first host sent. In this case, unless the NATs have

address-dependent mapping behavior, the NATs on the path have a binding ready for it and the second request is delivered [20, 15]. This delivered request causes a triggered check to be performed by the host that received the request and if the triggered check succeeds, a working path has been created and is ready for use. If the NATs do not allow creating such direct paths, the relayed candidates are used as the last resort. It is possible that multiple working paths are discovered during the connectivity tests but only the one with the highest priority is selected for use.

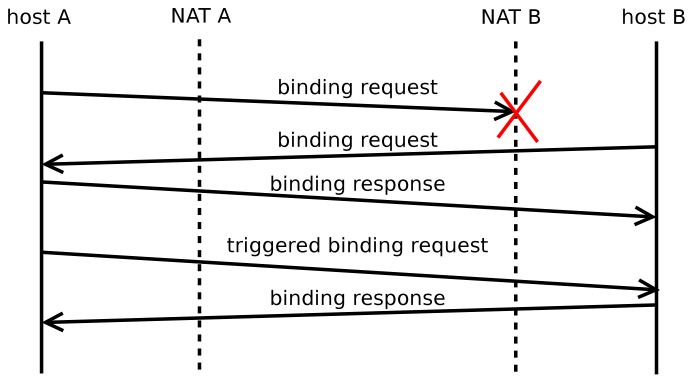

Figure 2.14: ICE connectivity checks in a simple NAT scenario

After an operational path is selected, it must also be kept alive. As mentioned in Section 2.1.3, a NAT may remove the binding if it is not used for a while. Normal data traffic is enough to keep the path alive, but if there is no data traffic for a while, keepalive traffic must be sent instead. ICE recommends using STUN *Binding Indication* messages for this purpose. Both hosts must send either data or periodic keepalives to each other since, as discussed in Section 2.2.5, due to security reasons, inbound traffic may not be enough to refresh the binding.

2.4.2 Advanced Features

While the basic operation of ICE is fairly simple, there are some nuances and optimizations in its operation that make it a bit more complex. Some of these more advanced features are introduced below.

Transaction Pacing and Retransmission Timers

The STUN and TURN transactions that ICE performs during the gathering and connectivity check phases are not started as fast as possible, but the request messages are sent only once every T_a milliseconds. The value for T_a is calculated in a way that the checks consume

roughly the same bandwidth that the RTP traffic ICE is making a path for would take. If ICE is used for setting up a path for non-RTP traffic, there is usually no way to know the bandwidth that will be used beforehand, so the draft mandates using a more conservative minimum value of 500ms for T_a.

The RTO value, i.e., the time to wait for an answer to a STUN request before sending the request again, for connectivity checks also relates to the T_a. For RTP sessions it is $MAX(100ms, T_a \times P)$ where P is the number of candidate pairs which will be tested in the connectivity check phase. A similar formula is used for non-RTP sessions but the minimum value in that case is 500ms.

Candidate Pair States and Triggered Checks

ICE candidate pairs that wait for their turn to start the connectivity check are in the WAITING state. When the connectivity check process starts, a check on the highest priority candidate pair in the WAITING state is sent and that pair is moved to the IN-PROGRESS state. A successful check moves the pair to the SUCCEEDED state whereas a failed check changes the state to FAILED.

If a check is received any time on a pair that is not already in the SUCCEEDED state, a triggered check to the other direction is scheduled for that pair into the triggered check queue. If the state of the pair was IN-PROGRESS, the current check transaction is cancelled and a new check is started. Otherwise the pair moves to the IN-PROGRESS state when the triggered check is performed. When it's time for the next check, if there are no checks in the triggered check queue, a check on the next highest pair in the WAITING state is started.

Creating Multiple Paths

ICE can be used to create simultaneously multiple paths for different *media streams*. A media stream is a single media instance that can be, e.g., an audio or a video stream [50]. A single stream may require multiple transport layer ports, called *components*. ICE uses information across different components to perform the tests faster.

In case of multiple components, all the candidate pairs are initially in a FROZEN state. ICE starts checks only on pairs that have unique *foundations*. Two candidates have the same foundation if they have the same type (e.g., server reflexive), they have the same local IP address, transport protocol, and the same STUN or TURN server was used at the gathering phase. Two candidate pairs with identical foundations have matching foundations for local and remote candidates. Because the network characteristic are likely similar for candidate pairs with matching foundations, a check on one of such pairs is also likely indicative for the other pairs. Thus, testing first only unique foundations can speed up the process.

Nominating the Candidates

There are two different ways for selecting the path that is used for the data traffic: aggressive and regular nomination. In the regular nomination, the controlling ICE endpoint decides when the connectivity checks should be stopped and sends a new connectivity check with a special flag on the highest priority candidate pair that has performed a successful connectivity check. When the peer receives the check and detects the flag, it nominates that pair for use. If the reply for the request arrives at the controlling host, it also nominates that candidate pair as the one to use for data traffic and the traffic is free to flow between the hosts. In case of aggressive nomination, the controlling host inserts the flag on every connectivity check binding request and when the first check succeeds, the associated candidate pair is automatically selected for use.

The aggressive approach can speed up the selection process, but it can also result in suboptimal path if the first successful check was not on the optimal path. Also, aggressive nomination may lead to a situation where more than one path is nominated and a higher priority pair replaces a lower priority pair as the pair to be selected for data traffic.

When a pair is successfully nominated, or if ICE was used to create more than one path and every path has a nominated pair, the ICE process is stopped. If the regular nomination is used, the controlling peer needs to decide when to nominate the best pair that is found so far. The ICE specification does not recommend any algorithm for this, but it is a matter of local optimization.

Discovering Peer Reflexive Candidates

When an ICE endpoint starts a connectivity test, it acts as a STUN client. It sends a STUN binding request from the local candidate towards the remote candidate, i.e., to the peer. All binding requests must contain a *priority attribute*. The attribute contains a value that is calculated in a similar way as the priority of the local candidates, but the type preference that is used in the calculation is *peer reflexive*. If the NAT, that is between the sender of the binding request and the public Internet, does not have endpoint-independent mapping behavior, it will create a different mapping for this binding request and allocate a different port than what was used for the binding request towards the STUN server. If the other host is not behind a NAT, or the NAT does not do port-dependent filtering, this connectivity check is delivered to the peer. Because the peer is aware of only the address and port that was seen by the STUN server during the address gathering and signaled to it in the answer/offer, the connectivity check comes from an unknown address. This address then presents a new *peer reflexive remote candidate* for the peer. The priority that was delivered with the binding request attribute is used as the priority of the candidate and the new candidate is added to the

list of known remote candidates; however, it is not paired with all local candidates. Instead, just one candidate pair is constructed by the peer from the local candidate which received the request and from the newly discovered peer reflexive candidate. The new candidate pair is added to the triggered check queue to wait for connectivity checks to the other direction.

The peer also answers to the binding request normally with a binding response that contains the peer reflexive address in the mapped address attribute. When the host that initiated the connectivity test receives the response, it checks the mapped address attribute and notices that it is an address that was not known to it before, and presents a new *peer reflexive local candidate*. The type of the candidate is, of course, peer reflexive, but the priority is the one that was used in the binding request's priority attribute. This way, both hosts can discover the new candidate pair and use the same priority for the new candidate.

2.5 Peer-to-Peer Session Initiation Protocol

Protocols which are used for building multimedia communication systems need to discover the locations of the hosts where the users who need to be contacted are. The Session Initiation Protocol (SIP) [51] solves this problem with centralized servers where a user can register his current location and which other users can then use to forward their session initiation requests to the right host. An alternative approach is to use a peer-to-peer overlay network for storing and retrieving this contact information and forwarding the session initiation requests in a distributed manner. The overlay network is formed by creating connections between the peers on top of the IP network. Instead of relying on centralized servers, the peers in the overlay collectively provide the required service with a distributed database algorithm. An example of this approach is the Peer-to-Peer Session Initiation Protocol (P2PSIP) which is currently being defined by the P2PSIP working group in the Internet Engineering Task Force (IETF). [6]

Figure 2.15 shows an example of a P2PSIP overlay. The dashed lines are the overlay connections between the peers. The topology of the network can be e.g., a ring or a mesh. In the figure there are 3 peers, titled *UA peer*, which are paired with a SIP User Agent (UA). They all participate in the overlay by storing the data and answering to queries but they can also use the overlay by themselves to query for registrations and start sessions.

The node *proxy peer* is coupled with a SIP proxy, the *redirector peer* contains a SIP redirector and the *gateway peer* serves as a gateway towards the Public Switched Telephone Network (PSTN). These three peers work as adapters between the P2PSIP overlay and other networks or devices. For example, the proxy and redirector peers accept standard SIP requests and use the overlay for resolving the needed next-hop. The plain SIP UA at the bottom of the picture does not participate in the overlay and it does not need to understand

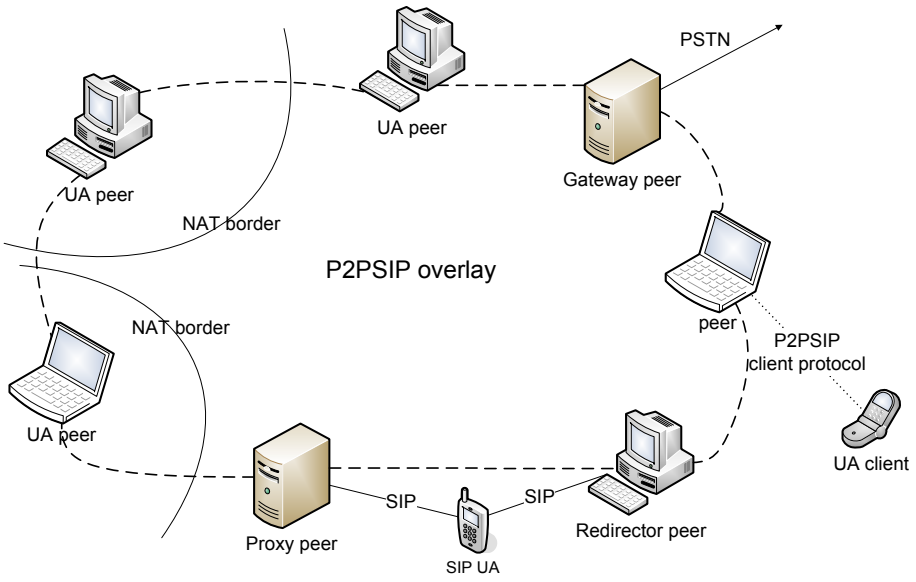

Figure 2.15: P2PSIP reference architecture

the protocol in the P2PSIP network, i.e., the peer protocol, but it can interwork with the overlay using the proxy and redirector peers.

The *UA client* on the right hand side of the picture, connected with a dotted line, understands only a subset of the peer protocol but can still perform some of the operations but may not participate in the network as fully as the other peers[4].

The two UA peers on the left hand side are both behind NATs in different address realms but they still work as fully featured peers. This kind of a distributed system which may have some of the nodes behind NATs needs a NAT traversal solutions such as the ones described in the previous sections.

2.6 Host Identity Protocol

The current TCP/IP model uses the IP address both as the *locator* and *identifier* for the hosts: the IP address is needed to route and forward the packets to the right destination in the network, but the address also names the host's networking interface. This dual role of IP addresses presents problems especially for mobile and multi-homed hosts. These problems and a solution to them, called Host Identity Protocol (HIP), are presented in [38, 32, 34].

[4]The role of the P2PSIP client is still under discussion at the P2PSIP working group

2.6.1 Mobility, Multihoming and Security

A mobile host may change its point of attachment in the network while being used for network communications. A multi-homed host has multiple network interfaces, with different IP addresses, that it can use either simultaneously or one at a time. Also, even a stationary host with single network interface may end up using different IP addresses due to dynamic configuration. In all of these cases, the IP address of the host will change even though the host that is connected to the network remains the same. Traditionally, transport layer connections are bound to an IP address of the host, and if the address changes, the connection does not survive. Therefore, dynamically changing IP addresses are a problem for the hosts.

Even if the transport layer connections survived from a change of IP address, there are security problems with multihoming and mobility. Since there must be a way to change the IP address where the traffic is delivered, a malicious host can try to pretend to be the rightful owner of the traffic and divert the traffic to itself for inspection or for performing a man-in-the-middle attack. Similarly, an attacker can try divert the traffic to a victim host which will receive unsolicited traffic resulting in Denial-of-Service (DoS) attack.

The Host Identity Protocol aims to solve these problems by inserting a new host identity layer and namespace between the transport and internetworking layers of the IP stack as depicted in Figure 2.16. Instead of using the IP address as the identifier for the host, with HIP, the identifier is the public key of an asymmetric cryptographic key pair. Also, instead of binding connections to the IP address, transport layer protocols can now bind to a presentation of the host identifier: Host Identity Tag (HIT). The host identifier can remain the same regardless of the IP address that is currently used for it. Because the binding between the host identity and the IP address is not fixed, the IP address can change without breaking the transport layer connections.

With this solution, mobility and multi-homing can be handled in a more natural way since multiple IP address can be bound to a host identifier and the underlying IP address can be changed dynamically. Also, because the host identifier is an asymmetric cryptographic key, the host can prove that it is allowed to use the identifier with the private key. This solves the problem with malicious hosts trying to change the destination of the traffic.

2.6.2 Creating a HIP Connection

A HIP session starts with a four-way handshake, called the HIP base exchange, depicted in Figure 2.17. The endpoint initiating a connection is called the *Initiator* and the other party the *Responder*. First, the Initiator sends a simple connection initiation packet, I1, to the Responder that contains only the HITs of the hosts. The Responder replies with an R1 packet that contains the Responder's public key and a cryptographic puzzle the Initiator must solve

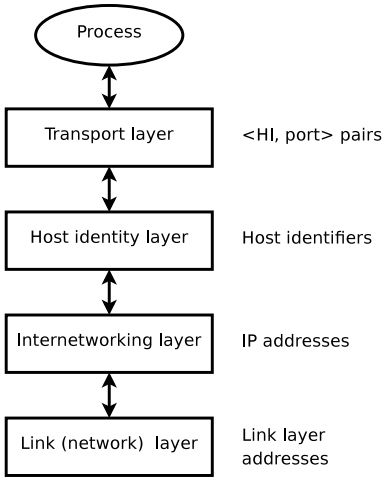

Figure 2.16: TCP/IP stack with HIP

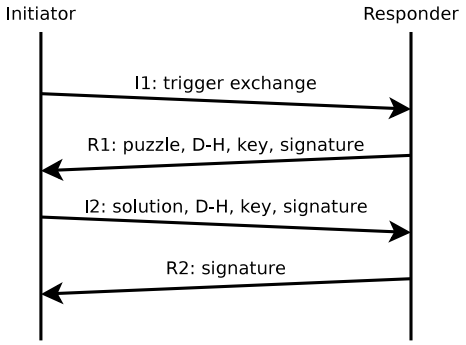

Figure 2.17: The HIP base exchange

before the Responder accepts the connection. No connection state at the Responder is created at this stage. This helps protecting the Responder from resource depletion DoS attacks since the Initiator has to perform more processing than the Responder before the Responder has to create any local state for the connection. When the Initiator has solved the puzzle, it sends the solution and its own public key to the Responder in an I2 packet. Both R1 and I2 are cryptographically signed and the signature is added to the packets. Using the public keys and signatures in R1 and I2, both of the hosts can verify the identity of the peer, i.e., they can check that peer has a private key that matches to the public key. The Responder verifies the puzzle solution and if it wishes to accept the connection, it replies with a signed R2 packet.

The I2 and R2 packets contain also Diffie-Hellman key exchange parameters (denoted with D-H in Figure 2.17) which allow the hosts to derive cryptographic keys to set up a secured connection between them. IP security (IPsec) using ESP is defined for HIP in [26] and future specifications can define other methods for securing the connections.

If a host is mobile, its IP address may change so often that it is not feasible to make it known by conventional means such as using DNS [36]. For this kind of situations, a host may use a Rendezvous Server (RVS) [29]. The host registers its current location with the RVS and tells the initiators to send the I1 packets to the RVS (e.g., by registering the RVS's IP address in the DNS). When the RVS receives an I1 packet, it makes a lookup on the HIT in the I1 and if there is a registered host for that HIT, forwards the I1 to the registered hosts. The RVS also includes the Initiator's address (where the I1 was received from) to the I1. When the Responder receives the I1, it can respond directly to the Initiator's address and the base exchange proceeds between the hosts without the RVS.

The HIT, where the connections are bound to, looks like an IPv6 address and is therefore compatible with the standard socket Application Programming Interface (API). This allows legacy programs that are not HIP-aware to use HIP to some extent transparently: when a host initiates a connection to a HIT of the other host, the HIP implementation can intercept this connection setup, run the HIP base exchange between the hosts, and hand over a socket to the application. All the traffic that is exchanged using that socket is then IPsec secured and supports mobility and multihoming.

Since HIP works below the transport layer protocols, it is normally run directly on top of IP. It can be used either as an extension header of IPv6 or as payload of IPv4. Also the IPsec ESP tunnel that is setup using HIP is transported on top of plain IP. Both HIP and ESP have therefore problems traversing NATs and other middleboxes and need NAT traversal mechanisms to cope with the problems. [60]

2.6.3 Proposed NAT Traversal Solutions

HIP NAT traversal can be made efficient and secure with a NAT that is HIP-aware [64]. Unfortunately, the legacy NATs that are currently deployed in the Internet do not support such features and need other ways for NAT traversal.

Ideas for legacy NAT traversal were proposed already at 2003 by R. Moskowitz et al. in [33]. This early version of the HIP specification used UDP encapsulation of HIP packets in IPv4 environments. NAT traversal mechanisms for HIP were further investigated by M. Stiemerling and J. Quittek in [59].

L. Silvennoinen explored and implemented in his thesis [54] a UDP hole punching mechanism (discussed in Section 2.3.1) for HIP NAT traversal. In this proposal, HIP control packets and ESP are both encapsulated in UDP. Endpoints use an external way (such as STUN) to detect whether they are behind a NAT, and if only the Initiator is behind a NAT, the simple UDP encapsulation is enough. However, if the Responder, or both of the hosts, are behind a NAT, an extended Rendezvous Server is used as the rendezvous service needed by the UDP hole punching. As noted in Section 2.3.1, this approach is able to traverse NATs as long as their mapping behavior is endpoint-independent.

ICE-based connectivity checks for HIP were proposed in [62] while concurrently discussed early (00 and 01) versions of [28] used simpler hole punching mechanisms. Ideas from the previous drafts evolved in to the current (04) version of [28] which suggests using ICE for solving the NAT traversal problem.

2.7 Summary

Network Address Translators allow connecting networks using private range addresses to other networks by transparently changing the addresses of the packets that cross the network border. Different types of NATs exist and they can be classified by their filtering and mapping behavior. NATs help, among other things, with IPv4 address depletion but they also create problems especially for peer-to-peer connections.

Various NAT traversal mechanisms have been developed to solve these problems. ICE is a robust NAT traversal solution that uses UDP hole punching and STUN/TURN protocols for creating optimal paths between hosts in NATed environments. Basically ICE performs UDP hole punching by probing for connectivity on all different paths between the hosts, but this simple approach is enhanced with a set of optimizations.

Peer-to-peer SIP uses a distributed P2P network architecture for communication and is an example of a P2P application that needs a NAT traversal solution. The Host Identity Protocol solves various problems with host mobility, multihoming, and security by introducing a new host identity namespace and a layer to the TCP/IP stack. HIP has problems with legacy NATs and different NAT traversal solutions have been proposed for it. In the next chapter we present a design and implementation for HIP NAT traversal and in the later chapters we discuss how it can be used for enabling NAT traversal for P2PSIP.

Chapter 3

NAT Traversal Using HIP with ICE

In this chapter, we first briefly discuss the need for NAT traversal and present some of the benefits and drawbacks of HIP-based NAT traversal. For solving the NAT traversal problem using HIP, an ICE library was created that could be integrated into a HIP implementation. The following sections discuss how ICE can interwork with HIP and describe the architecture of the library. Also design decision and some implementation experiences from implementing the ICE library are presented.

3.1 Need for NAT traversal

As mentioned in Section 2.1.2, NATs commonly multiplex connections using the transport layer identifiers. If the packets do not contain a transport protocol that is known to the NAT, such as UDP or TCP, they simply drop the them. This alone is enough to block HIP connections that are normally run on top of IP. Besides, HIP should work for both client-server and peer-to-peer connections, so the problems with connecting a host behind a NAT also apply to HIP. Therefore, HIP needs a NAT traversal solution before it can be successfully used in the Internet.

NAT traversal can be performed in many ways and also at different layers of the protocol stack. Because, today, NATs are so widespread, many applications that need to create peer-to-peer connections must deal with NAT traversal. For example, the Internet telephony program Skype is well known for its capabilities for creating connections even in environments with NATs and firewalls [3]. Also, peer-to-peer file sharing programs, such as BitTorrent client BitLord [5], have developed their own NAT traversal techniques.

Instead of developing NAT traversal mechanisms for each protocol or application separately, all applications could benefit from a lower layer solution. The IPsec tunnel created with HIP looks like a normal IP-based path to the application layer and the applications can

run any protocol on top of it. With a proper NAT traversal solution, HIP is able to create an IPsec tunnel between two HIP-enabled hosts, through the NATs. This way, there is no need for protocol specific NAT traversal but any protocol can be run on top of HIP without worrying about NATs.

3.1.1 Benefits and Drawbacks of Using HIP

Many applications also need more than one concurrent connection between the hosts. For example, a signaling part of the protocol may use different ports than the actual data. The data may also need more than one UDP/TCP port: e.g., audio and video can be transmitted using different ports. Because all upper layer connections between two hosts can be multiplexed into a single UDP encapsulated IPsec connection, the NAT traversal must be done only once between the hosts and all subsequent connections can use the same path.

The downside of this is that a NAT or a firewall in a gateway can no longer protect the host based on TCP/UDP port numbers because all the traffic appears to use the same port. However, even without NAT traversal a HIP host will need to use a firewall at the host since a firewall at the gateway does not know what is transferred in the encrypted ESP connection.[1]

As noted in Section 2.6, HIP can handle host mobility and multihoming securely and efficiently, which can be especially useful for P2PSIP. A personal communication device that uses P2PSIP, whether it is e.g., a mobile phone or a laptop computer, is likely to be carried around with the user. When the user moves, the terminal may switch to a different network and get a new IP address. A connection that does not support mobility would break, but HIP is able to restore connectivity without breaking upper layer connections. Also, both laptops and mobile phones have today often more than one network interface: laptop computers have commonly both Ethernet and WLAN interfaces and many high-end mobile phones have a WLAN interface in addition to the cellular interface. HIP's multihoming support can utilize multiple interfaces and switch from an interface to another without disturbing the upper layer connections.

3.2 Integrating ICE into HIP

We chose to use ICE as the NAT traversal mechanism for HIP since it provides a robust mechanism that should work in various different NAT scenarios. Also, in the HIP working group, there was a rough consensus that ICE, or some ICE like mechanism, would be the best way to implement the NAT traversal for HIP.

[1] A firewall that knows the public-private key pair of the host would actually be able to decrypt the ESP connection and protect the host, but this is outside of the scope of this thesis

For defining the protocol extensions and how ICE should fit into the picture, a design team was formed in the IETF HIP working group. The output of the design team was a HIP NAT traversal specification draft called *Basic HIP Extensions for Traversal of Network Address Translators* [28]. The draft defines how HIP control and data packets can be encapsulated in UDP, how the signaling path needed by ICE can be set up using HIP and used for conveying the ICE offer and answer, and how the path created by ICE can be used for data traffic and kept alive with keepalive signaling. The following sections discuss these issues in more detail.

3.2.1 UDP Encapsulation

Since NATs do not often allow any other protocol than UDP or TCP pass them, plain HIP over IP cannot usually be used if there are NATs between the hosts. The need for UDP encapsulation of HIP control messages and ESP data traffic is mentioned in RFC 5207 [60], and defined for ESP at RFC 3948 [22]. The HIP NAT traversal draft defines similar UDP encapsulation for HIP control packets as [22] uses for Internet Key Exchange (IKE) protocol: the standard UDP header is followed by a marker with 32 bits of zeroes, as shown in Figure 3.1, to distinguish HIP control packets from UDP encapsulated ESP packets.

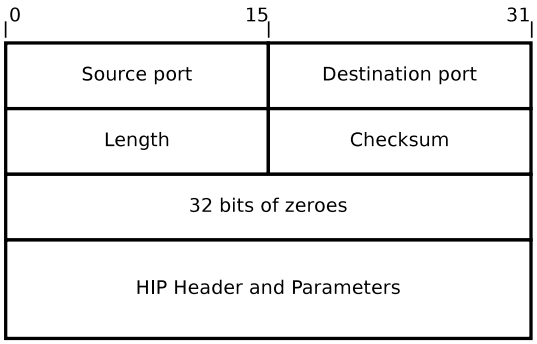

Figure 3.1: UDP encapsulated HIP control packet

3.2.2 HIP Signaling Path

The signaling path needed by ICE for exchanging the candidate pairs is created using a HIP Relay. A HIP-enabled host can register to the HIP Relay using the registration extensions defined in [28]. Basically, at registration, a UDP encapsulated HIP base exchange, shown in Figure 3.2, is performed with the relay and the relay announces its capability of working as a HIP Relay in a registration information parameter contained in R1. The Initiator then requests for relaying service in I2 and if the relay accepts the request, it acknowledges the

request with a registration response in R2. The R2 message also includes a REG_FROM parameter that contains the address where the Initiator appears to send packets from; this address can be used as the server reflexive candidate for the address where the base exchange was started from.

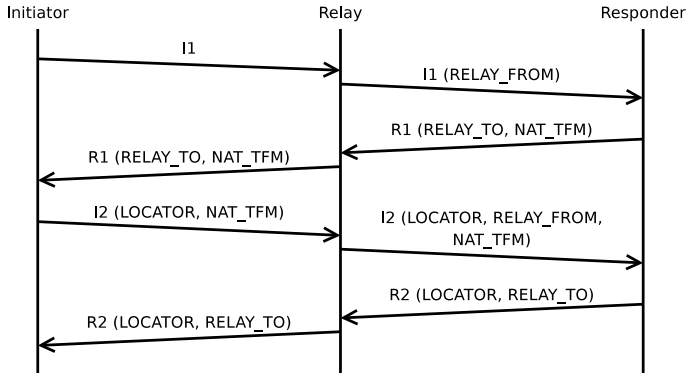

Figure 3.2: HIP base exchange via a HIP Relay

If the registration succeeded, all well formed I1 packets sent to the registered host's, i.e., Responder's, HIT (and to the relay's IP address) are forwarded to the Responder. The relay's behavior with I1 is similar to one with HIP RVS, but whereas an RVS forwards only the I1 packet, a HIP Relay can also forward the rest of the base exchange. This is needed because a NAT with endpoint-dependent filtering behavior will not let through an incoming packet (R1) from a different address than where a packet (the I1 in this case) has been sent recently. To be able to receive the incoming connection attempts, the Responder must keep the NAT bindings on the way to the HIP Relay alive by sending periodic keepalive HIP notify messages to the relay.

When an Initiator sends I1 trough the relay to the Responder, the relay adds a RELAY_FROM parameter to I1 before sending it to the Responder. This parameter tells the Responder where a response should be sent to. The Responder then responses through the relay with an R1 message. The R1 message contains the address from the RELAY_FROM parameter in a RELAY_TO parameter which the relay uses for forwarding R1 to the right address without needing to remember where the I1 message came from. R1 also includes a NAT transformation negotiation parameter (NAT_TFM) which contains a list of NAT traversal modes supported by the Responder. Currently, only one mode with ICE-based connectivity checks and UDP encapsulated control and data traffic is defined.

If the Responder does not wish to use a HIP Relay and do the ICE connectivity checks, but wants to use just UDP encapsulation for the HIP control and data traffic, it can omit the NAT transformation parameter from R1. When the Initiator does not find a NAT transform

and RELAY_TO parameters in R1, it can continue the base exchange and exchange data traffic as it would normally do but just use UDP encapsulation.

It is also possible for the Initiator to try a concurrent base exchange without UDP encapsulation, i.e., plain HIP over IP. If the Initiator receives a valid, non-encapsulated R1 for the I1 message it sent without the UDP encapsulation, the UDP encapsulated version of the base exchange is ended and the non-encapsulated base exchange is finished. Because, in this case, the HIP control packets did not need UDP encapsulation, also ESP traffic without UDP encapsulation should work between the hosts and normal HIP procedures without NAT traversal can be followed.

If the hosts selected a NAT transform with ICE connectivity checks during the base exchange, they exchange ICE candidates in I2 and R2 using LOCATOR HIP parameters. The candidates can be gathered any time before the connectivity checks, but must obviously be gathered by latest before I2 and R2 are exchanged if ICE is used.

3.2.3 ICE Connectivity Checks

After a successful base exchange, in which the Initiator and the Responder agreed to use ICE for the connectivity checks, the checks can be started. The checks are performed as described in Section 2.4 but what is specific to HIP is that only a single media stream and component are tested, so, for example, frozen candidates part of the ICE algorithm or candidate foundations are not needed. The HITs of the Initiator and Responder are used as username fragments for the connectivity checks and both hosts derive STUN password from their keying material in a similar way as they do for the HIP and IPsec ESP keys.

After ICE connectivity checks are finished, the best working path is selected and ready for use for UDP encapsulated ESP. However, if the path is idle longer than 15 seconds, keepalives are needed to keep the NAT binding alive. ICE with HIP uses normal STUN binding indications on the data path for this purpose. If the hosts did not agree to use the ICE connectivity checks, HIP NOTIFY messages are used instead.

3.3 Implementation Architecture

An implementation of ICE is needed for the ICE candidate gathering and connectivity check phases that were discussed in the previous sections. For this purpose, we implemented an ICE library for the Linux platform with the C programming language. The library consists of 7 modules and roughly 6000 lines of code.

The library's division into modules is depicted in Figure 3.3. The socket IO functionality needed to send and receive STUN messages efficiently is implemented in the ICE_IO module. It also contains utility functions for comparing and copying socket address structures

independently of the IP address family. The ICE_TIMER and ICE_TIMER_IO modules implement timer and event multiplexing. Other modules can register callbacks to be executed when either a socket receives data or a certain amount of time is passed using the functionality provided by the timer modules. If the program where the ICE library is integrated (such as the HIP implementation) already has its own event multiplexing functionality, only the ICE_TIMER_IO module has to be changed to make it possible to multiplex ICE events with other events provided by the program without using multiple threads.

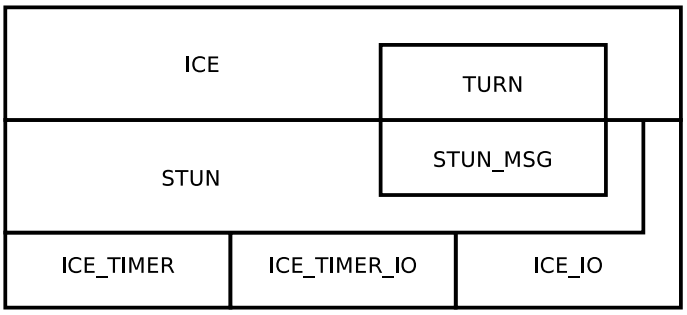

Figure 3.3: ICE library architecture

The STUN_MSG module implements STUN message parsing and construction. It provides functionality for creating STUN messages, adding Type-Length-Value tuples (TLVs) and also parsing the TLVs that are defined in the STUN draft [49]. The STUN module contains the application logic for handling incoming STUN messages and also for sending STUN messages. It provides a set of callback structures that higher level modules can use for setting up STUN usage specific behavior for different types of STUN messages and application data.

The TURN module contains an implementation of a TURN client that can create an allocation on the TURN server, send data using binding indication messages, and keep the allocation alive by refreshing it. It uses the STUN and STUN_MSG modules for creating the STUN messages and transactions.

The ICE module uses the TURN, STUN and STUN_MSG modules for all processing phases and provides all the rest of the logic that is needed for candidate gathering and connectivity checks. This module also acts as the main interface for using the ICE library. For the candidate gathering phase ICE needs to perform low-level socket actions, such as binding to certain interfaces. For this purpose is also uses the ICE_IO module directly.

3.4 Implementing ICE

While the ICE specification was used as a guideline for the implementation, the goal of the implementation was not to have all the features of the specification, but rather the minimal set that would work in all NAT scenarios and interoperate with other implementations. The next subsection presents some of the differences from the specification. Also, implementation efforts revealed some features from both the Linux platform and ICE specification that were not immediately clear and deserve a few words of discussion in the following subsections.

3.4.1 Differences From the Specification

Some features of the ICE draft [46] were not implemented in the prototype. Many of those features are specific to SIP and therefore not needed for HIP NAT traversal. Some features could be useful, but would add a fair deal of complexity without much observed benefits, so they were left for future work. The goal was to make an implementation that would reasonably interoperate with any implementation conforming to the specification but still avoid excessive implementation effort. The SIP specific features left out of the prototype include:

- Everything related to the SDP offer/answer exchange; HIP does not use SDP, but HIP parameters, for encoding the offer and answer.
- Subsequent offer/answer exchange; there is no need to send the selected candidates through HIP signaling path.
- ICE-CONTROLLED attribute and role conflict repair; unlike with SIP, there can be no role conflicts with HIP: the Initiator is always the one in control.

Other features that were not implemented include:

- Frozen candidates, foundations and multiple checklists; HIP uses only one connection (one ICE component and stream) so there are no candidates to freeze nor is there a need for using foundations or multiple checklists.
- Transaction pacing calculation based on RTP features; since we are not setting up an RTP stream, we use only the formula for the non-RTP traffic.
- Triggered check queue; only the normal connectivity checks are paced and the triggered checks are sent immediately.

Only one of these features that is likely to have an impact on the ICE performance is the lack of the triggered check queue. Since now a triggered check is sent immediately after sending a response, our implementation is a bit more aggressive at sending the checks. One of the reasons why the checks should be paced is that some NATs are not able to create bindings very fast [46]. However, in case of triggered checks, there is already an existing binding which was used by the response message so this not that big problem.

In addition, our ICE implementation does not share STUN and TURN sockets, but uses a separate TURN socket for the relayed traffic. This made the implementation easier but requires an additional binding request that is made to the TURN or STUN server because the TURN socket's local address is different from the STUN socket's address. This will likely be fixed in the future versions and it does not affect the performance analysis section since we concentrate only on the connectivity check phase.

3.4.2 Sending Checks From Different Interfaces

A feature that was implemented in the prototype, and required some research and more tricks than expected, was sending the checks from a host that is multihomed.

A multihomed host has multiple IP addresses that are from multiple network interfaces. The connectivity checks should be sent from different interfaces so that different paths are really tested. Unfortunately, at least on Linux, it is not enough to bind a socket to a certain IP address to assure that a message sent from the socket use the related interface too. Instead, the kernel decides, based on the destination address and regardless of the address where the socket was bound to, which interface to use for outbound packets. To fix this, the proper interface must be set using the socket option *SO_BINDTODEVICE*. Unfortunately this requires administrator (root) rights for the program.

Our ICE implementation warns the user if it can not set the proper interface but still proceeds normally. However, in this case, if the host has an interface to a private network and another one to a public network, even paths from the private network's interface appear to work to the public network because the kernel uses the public network's interface for the connectivity checks towards a host with a public range IP address. Therefore, without root rights, the ICE library does not work reliably on multihomed hosts.

3.4.3 Stopping the Connectivity Checks

As mentioned in Section 2.4.2, the decision when to stop the connectivity checks is a matter of local optimization. Even though this issue received only little attention in the specification, it turned out to be crucial to the performance of ICE. If we use the simplest possible scheme and just wait for all the checks to finish (the non-working candidate pairs use up

to 6 retransmits), the procedure takes typically up to 40 seconds to finish. For example, if both hosts are in different private subnets, the highest priority pair, private to private address, does not ever work. Then, we use multiple attempts and exponential back-off for the retransmit timer as discussed in Section 2.3.2. Obviously, this time is too much for many applications and there is hardly any benefit for trying that long — especially if we have some less optimal working path before that.

We chose to use a simple check stopping algorithm with two different timeouts: a *hard deadline* that is the maximum time the checks will ever be running, and a *soft deadline* after which we stop the checks if we already have a path that does not use relays, i.e., is "good enough". If the checks on the highest priority candidate pair succeed, no further checks are necessary and that pair is nominated for use.

The hard deadline was set to 10 seconds, since even with a relatively long Round Trip Time (RTT) and some packet loss there is a good chance for finding a path, if such exists, during that time. Unless there are more than 10 candidate pairs, 10 seconds is enough for giving every candidate pair a chance for a retransmit even with a 500ms connectivity check pacing. Also, in case of interpersonal communication, the person who is trying to start communication is not likely to wait much more than 10 seconds for a path but will rather get a notification that there is no connectivity. However, this assumption can be wrong if the RTT is in the order of seconds, or even longer than our hard deadline. Such a situation can happen e.g., with connections created using a mobile phone network. In this case, a longer hard deadline may be necessary.

The soft deadline was set to 2 seconds. If both hosts are in different subnets and behind different NATs, the highest priority candidate pair will always fail. Therefore, it makes sense to accept server reflexive candidate fairly quickly. Checks within the same subnet would validate fast, if ever, because packet loss is not very likely in that case. The downside of accepting a server reflexive path fast is that we may end up using unnecessarily a NAT hairpinning path if a server reflexive candidate is validated before the host candidate and there is no working host candidate before the soft deadline.

The 2 second deadline is also in line with International Telecommunication Union (ITU) recommendation [24] for post-selection delay on local connections. The recommendation is (less than) 3 seconds under normal load, so ending the checks after 2 seconds gives the preceding and subsequent signaling, such as ICE candidate exchange and SIP negotiation, still one second to finish.

Basically, the soft deadline could be set even to a lower value, e.g., to 1 second or less, if connecting to a host in the same subnet is not likely or using the very best path is not crucial. Having a very low (or zero) value for the soft deadline approximates aggressive nomination of ICE with extra connectivity check for nominating the first working path.

This algorithm could be further optimized by checking that if there is no chance for a connectivity check to succeed before a deadline is passed, we would act as if the deadline had passed already. This kind of situation can occur if none of the pairs is in progress state and the time when the next pair moves to progress state is after a deadline.

3.5 Summary

HIP needs a NAT traversal solution, but such a solution will also benefit all upper layer protocols that use HIP for initiating connections. NAT traversal with HIP has many benefits compared to protocol specific solutions, but it can also make protecting hosts with a middlebox on the network border harder.

For NAT traversal, HIP control and data packets are encapsulated in UDP and the HIP base exchange is done through a HIP Relay. ICE can be used with HIP by exchanging the ICE candidates in the HIP base exchange and running connectivity checks after the base exchange. HIP's NAT traversal extensions can also be used for negotiating what kind of NAT traversal mechanism is used.

Our ICE library implementation consists of 7 modules which implement STUN, TURN and ICE functionality and also provide socket and timer multiplexing services. The implementation differs for some parts from the ICE specification since it does not contain, e.g., SIP specific functionality. For deciding when to stop the connectivity checks, our implementation uses two different deadlines: if a sufficiently good path is found after 2 seconds, the checks are stopped, but otherwise they continue until 10 seconds have passed.

The next chapter takes our implementation into use and presents how well it works in different NAT traversal scenarios.

Chapter 4

Measurements and Evaluation

In this chapter we first discuss how the NAT traversal using the ICE methodology should work in theory. We also introduce the prototyping environment that was used for testing the ICE library presented in the previous chapter. Then, we share some experiences on how some NAT implementations work in practice. Finally, we present our measurement data from the tests and based on the data analyze the ICE connectivity check process in different NAT traversal scenarios.

4.1 Theoretical NAT Traversal Using ICE

ICE should be able to traverse any combinations of NATs as long as either one of the hosts has a TURN relay that both communicating parties can connect to. However, use of relay servers is undesirable and hence it is important that ICE is able to traverse NATs in most of the scenarios without using a TURN relay. In this section we discuss what kind of scenarios should, in theory, work without relaying.

4.1.1 Impact of Mapping and Filtering Behavior

If only one of the hosts is behind a NAT, or even multiple tiers of NATs, the type of the NATs does not matter, because the connectivity checks from the NATed host to the non-NATed host behave like client-server communication which works with all the NAT types. However, if both of the hosts are behind a NAT, the type of the NATs counts.

If all the NATs have endpoint-independent mapping and filtering behavior, the mappings, that are created when the hosts query for the server reflexive address from the STUN server, accept also any packets sent by the other host. Then, already the first connectivity check sent to a server reflexive address passes all the NATs and confirms a working path. Nevertheless,

if there is at least one NAT between the two hosts whose filtering behavior is address, or address and port-dependent, the connectivity checks to that direction are initially dropped.

For the address-dependent filtering behavior, it is enough that the host(s) behind a NAT send packet to any port of the peer, whereas the address and port-dependent filtering requires that a connectivity check is sent on the same port where the peer originates (or seems to originate, if it is behind a NAT) his checks. If the mapping behavior of all the NATs is endpoint-independent, the connectivity checks towards a peer seem to originate from the same transport address that the STUN server saw the binding request coming from. This address was sent to the peer during the candidate exchange, and the peer should be sending checks to that address. Therefore, the peer's NAT should have a ready binding for a check that comes from the same address and is destined to peer's server reflexive address. For this reason, ICE works even with the strictest address and port-dependent filtering behavior.

Nevertheless, if any of the NATs have address or address and port-dependent mapping behavior, the peer's NAT's filtering behavior dictates whether a route without a relay is possible. If on the way to the public network, a NAT has an address or address and port-dependent mapping behavior, a check sent to the peer's address gets a different port on the NAT than what the STUN server saw. If the host A is behind such a NAT, and the peer, host B, has only endpoint-independent filtering NATs, the check is delivered to host B and it discovers a new peer reflexive address as described in Section 2.4.2. However, if any of the NATs in front of the host B have address or address and port-dependent filtering behavior, such a check is dropped. When the host B sends a check towards host A's server reflexive address, it creates filtering rules on the NATs that allow the next check by host A to pass if the filtering behavior is only address-dependent. If the behavior is also port-dependent, proper rule is never created and only a route through a relay works.

Also, if both of the hosts happen to have an address or address and port-dependent mapping NAT on the way towards the public network, they cannot discover the mapping the NAT creates for them towards the peer, unless they can use a STUN server that is on the same address as the peer's NAT. Even in this case, if any of the NATs towards the STUN server have port-dependent mapping behavior, the mapping will be different for the connectivity checks that are not sent to the STUN server's port.

To summarize, ICE methodology is able to create a working, direct path without a TURN server as long as there are no address (and port) dependent mapping NATs, or if one of the hosts is behind such a NAT, the other host must not have a NAT that has address and port-dependent filtering behavior.

If a host is multihomed, it may have more than one path to the peer. Because ICE tries all the different paths, the path that has the best NAT behavior counts. In other words, if any of the interfaces is free of NATs, or the NATs have endpoint-independent mapping behavior, no relay is needed regardless of the type of the NATs the peer has.

4.1.2 Multiple Layers of NATs

If hosts are behind more than one layer of NATs, things can get bit more complicated. As noted in the previous section, in case of more than one NAT, the NAT that has the strictest filtering behavior, or worst (i.e., not endpoint-independent) mapping behavior, dictates how well the NAT traversal works. However, multiple layers of NATs can also cause other types of problems.

As shown in Section 2.1.6 and Figure 2.7, the shortest path between hosts in different subnets is not necessarily through a public network. Still, since host B's private address is routable from the NAT host A is behind, ICE is able open and use the best path between the hosts. Yet, if we develop this scenario further and also place host B behind another NAT, as depicted in Figure 4.1, its private address is in different subnet than where the host A's NAT is, so it is no longer routable from the NAT1. If the STUN server is on the host C (or in the same public network), the binding both hosts learn is on NAT2.

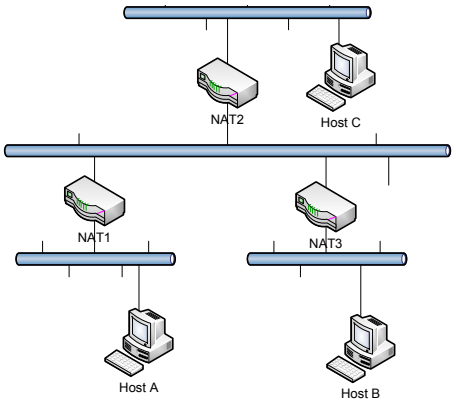

Figure 4.1: NAT scenario with two hosts behind multiple layers of NATs

If the last NAT (NAT2 in the figure) before the public network does not support hairpinning, it will not forward the traffic back between the hosts even if they had created proper bindings on all the NATs and all the NATs had P2P-friendly endpoint-independent filtering and mapping behavior.

A scenario like in Figure 4.1 can happen e.g., if an ISP uses a NAT to serve a group of home subscribers. If the home subscribers want to set up their own LANs, they may use a router with NAT functionality and all the hosts in the home LAN end up behind a second layer of NATs. For this reason, it is important that if ISPs deploy NATs in their network, those NATs must support hairpinning. Otherwise customers within that ISP that are using their own LANs can not communicate without the help of a relay.

Because the base problem is that the only routable address both hosts in the home LANs know of each other is on the NAT2, a way for hosts to learn addresses their own NATs have given them would also help. A STUN server deployed in the "private" network served by NAT2 could tell the hosts those bindings, but discovering such a server is problematic. A STUN server in the public, globally routable network may be pre-configured to an application since it is likely to be available from most of the private networks, but that is no longer the case with a STUN server in a private network. Also, it is not enough to use any STUN server, but the server has to be in the right place in the network topology. As a remedy to the problem, the DNS discovery mechanism suggested by [49] could perhaps be extended to provide not only STUN servers in the public Internet, but also servers located in ISP-NATed networks.

4.2 Prototyping Environment

To test how ICE works in practice, and to see whether the behavior matches the theoretical behavior discussed in the previous section, we created a prototyping network whose topology we could change easily and into which we could insert different types of NATs. The ICE prototype was tested in an environment consisting of 7 virtual Linux computers, with kernel version 2.6.18, running on VMware virtualization software. The network topology of the environment is depicted in Figure 4.2.

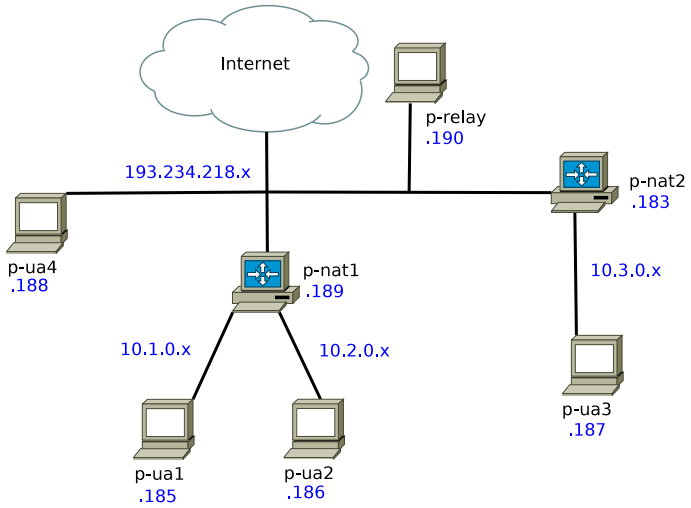

Figure 4.2: Prototyping environment's network topology

Two of the computers (p-nat1 and p-nat2) were used as NATs with the standard iptables [25] masquerading function. The version that was in use in the prototype environment

was 1.3.6, found in the Ubuntu Linux distribution. The iptables configuration is shown in Appendix A. For some tests we changed one or two of the Linux NATs to DLink DIR-635 wireless N[1] routers. This model was chosen because it can be configured to use different types of NAT filtering schemes: *endpoint-independent*, *address restricted*, and *port and address restricted*. These schemes are in line with RFC4787's [2] filtering behaviors. The DLink router has endpoint-independent mapping behavior.

The host p-ua4 worked as the STUN/TURN server running the PJSIP project's TURN server [41] version 0.8.0. The PJSIP project's server was used to reduce implementation effort but also to test interoperability of the prototype implementation.

For signaling the address candidates to each other the hosts use a simple, TCP-based, relay program at the host p-relay. The TCP relay therefore performs the same function that a HIP Relay would do. When a test is started, both hosts gather address candidates and connect to the TCP relay. The Initiator sends its own address candidates through the relay, and after the Responder has received the candidates, it sends its own candidates to the Initiator. Then, both endpoints start the connectivity checks.

The majority of the tests were run between the hosts p-ua1 and p-ua3. For the tests where the hosts were not behind a NAT, either p-ua1, p-ua3, or both, were connected to the public network by disabling the interface to the private network and enabling another interface to the public network. The host p-ua1 was always the Initiator of the connection, and therefore the controlling endpoint for the ICE checks. For the tests where both of the hosts were in the same subnet, the tests were run between the hosts p-ua1 and p-ua2.

For measurement purposes, we ran tcpdump [61] network traffic monitoring program on both of the hosts performing the connectivity tests. Also, we stored all the debug prints of the ICE library for later analysis. The tcpdump data was used for counting the number of connectivity check messages and the amount of sent bytes.

4.3 Observations on NAT Behavior

Even though NAT behavior is today standardized in [2], some of the NAT implementations do not follow the recommendations or do not always even fit into the defined categories.

From the tests we discovered that the Linux iptables NAT implementation has normally endpoint-independent mapping and address and port-dependent filtering behavior. That is, packets from the same private host's source address and port will use the same public binding regardless of the destination address, but traffic is allowed back only from the addresses and ports where it was originally sent to. However, it seems that if a Linux NAT receives a packet to the public address it has reserved for a binding, from an address on the public side,

[1] Hardware version B1, firmware version 2.21EU

where there has been no packets sent so far using that binding, the outbound packets to that public address will not use the same mapping but will create a new mapping, and therefore come from a different port on the NAT. This is demonstrated in a tcpdump capture of the beginning of the connectivity checks in Appendix B.

The mapping behavior of the Linux NATs makes ICE NAT traversal harder. The first connectivity check (sent by peer A, P_a, in Figure 4.3) made to the server reflexive address is always dropped. A check sent by the other peer, P_b, could pass both NATs since the dropped check created necessary bindings in the P_a side. However, if P_b also has a similar Linux NAT, the source port, at the NAT, of the connectivity check sent by P_b is now changed (from $b1$ to $b2$ in the figure), as described above. Then, also this connectivity check is dropped due to the endpoint-dependent filtering policy of NAT A (only check from port $b1$ would have passed). Because of this behavior, hosts need to fall back to using a relay even if the mapping behavior of the NATs is not (normally) address-dependent.

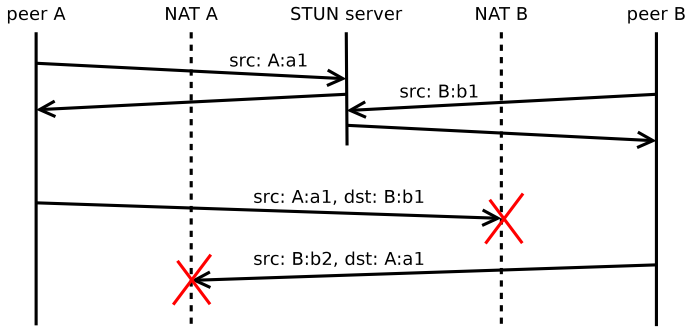

Figure 4.3: Failing connectivity checks between two hosts behind Linux NATs

Even if only one of the two NATs was a Linux (or similar) NAT as described above, depending on the timing of the checks, problems may emerge. In one possible scenario, P_a is still behind a Linux box, NAT_a, but P_b is behind another box, NAT_b, that does endpoint-independent mapping, but endpoint and port-dependent filtering[2]. If P_a sends a check to the server reflexive address of P_b, it creates a proper binding to NAT_a and the following check sent by P_b will use that binding and will reach P_a successfully. However, if a check sent by P_b happens to reach NAT_a before P_a has sent its first check, the mapping will be again different. Since NAT_b does port-dependent filtering, it will not allow a check coming from a different port to pass but will simply drop it. Even if the checks are repeated multiple times by retransmits, all of them are dropped.

This problem does not occur if the peer behind the badly behaving NAT is the one who sends the checks first, or the RTT between the peers is long enough and they start the checks

[2] As noted in Section 2.2.6, this is a really common NAT type

simultaneously so that the checks go pass each other in the network. Yet, since there are no guarantees that the endpoints start the checks exactly at the same time, sometimes the best path is created and sometimes not. A solution to this problem would be to delay the checks from the P_b side, so that the proper bindings were created on NAT$_a$ before the checks from P_b reach it. However, there is usually no way of knowing which one of the peers, if either one, is behind such a NAT. Both hosts could also double the amount of local candidates and always delay the checks for the other candidate, but this would multiply the amount of checks causing a lot of redundant traffic and increase the connection setup delay caused by the connectivity checks. Perhaps a better way to solve this would be to first run the normal ICE procedure, and if it seems that the best route is through a relay, a host could use the secondary candidates and checks between them as a second-to-last resort. This would require a change to the ICE protocol but it could increase the un-relayed success ratio a bit. The amount of increase in success ratio would depend on the amount of Linux-like NATs in the wild.

The behavior of the NAT in the D-Link Wireless router was more consistent and in line with [2] than the behavior of the Linux NATs. This was not surprising since that router model was chosen exactly for the reason that it had implementations of the different filtering schemes defined in [2].

4.4 Measurement Results

To asses how well the ICE methodology and our library works, we run a set of tests with it in different NAT settings. The prototype was also used with two different timer values to see how that affects the performance of the library.

We used 25 different NAT scenarios in our prototyping environment that was presented in Section 4.2. From all the scenarios we measured how long the whole process took and the amount of messages and bytes that was exchanged. The tests were repeated 5 times for each scenario to also survey variance in behavior. One of our goals was to get real numeric data from the ICE process and thus we give detailed descriptions of different results.

First, only one of the hosts was behind a NAT and we used four different NAT types. One NAT type was the Linux NAT and the rest were DLink routers with different filtering behaviors: endpoint-independent, address-dependent, and address and port-dependent. Then, we put both hosts behind a NAT in different subnets and varied the NAT type as in the previous test. Finally, we also tested how the prototype works if neither one of the hosts is behind a NAT. The notations used for different NAT types are summarized in Table 4.1. When both the Initiator's and Responder's NAT type is expressed, they are separated with a hyphen, and the Initiator's NAT type comes first.

Table 4.1: NAT scenario notations

Notation	Meaning
L	Linux NAT
EI	Endpoint-Independent filtering NAT
AD	Address-Dependent filtering NAT
PD	Address and Port-Dependent filtering NAT
N	No NAT

Quick Mode

As noted in Section 2.4.2, the ICE draft requires that the connectivity checks for non-RTP traffic are not sent faster than once every 500ms. However, to experiment with the timer values, we used also a lower value and call this version the *quick mode*. In the quick mode, the check pacing (T_a) is set to 100ms, so the connectivity checks are started 5 times faster than allowed by the ICE draft. Also, the soft deadline was changed to 1 second and the hard deadline to 5 seconds to better fit to the new check pacing. Because of the new T_a, also the RTO changed since it relates to T_a as described in Section 2.4.2. Essentially, all RTO values were five times smaller, but at least 500ms. Besides these new timer values, the implementation and stopping criteria were identical to the normal version.

4.4.1 Selected Path

Table 4.2 presents the paths that were chosen by ICE in different scenarios. The results were similar for both the normal and quick mode and they are combined in the table. The first column of the table presents the type of the p-nat1 (Initiator's NAT) and the first row contains the type of the p-nat2 (Responder's NAT). The middle section of the table presents the type of the candidate for both the Initiator and Responder that was nominated for use. The candidates can be host (H), server reflexive (S), peer reflexive (P), or relayed (R). For example, a path from the Initiator's host candidate to the Responder's server reflexive candidate is marked as H - S. If different tests of the same scenario resulted in different candidate, both of the candidate types are given, separated with a slash.

As can be seen from the table, in the scenarios where only one the hosts is behind a NAT, the host that does not have a NAT ends up using the host candidate and the other one a server reflexive candidate. Also, if there are no NATs between the hosts, the selected path is directly between the host candidates. In majority of the scenarios where both hosts were behind a NAT, both were able to use either the server reflexive or a peer reflexive candidate. However, if there was a Linux NAT and some other kind of NAT between the hosts, there

Table 4.2: Selected path in different NAT scenarios

NAT types	N	EI	AD	PD	L
N	H - H	H - S	H - S	H - S	H - S
EI	S - H	S - S	S - S	S - S	S - S/P
AD	S - H	S - S	S - S	S - S	S - S/P
PD	S - H	S - S	S - S	S - S	S - S/R
L	S - H	P/S - S	P/S - S	S/R - S	S - R

is some variation in the eventually selected path's type: depending on the test run, either server or peer-reflexive candidate was used for the host behind the Linux NAT.

In the scenarios EI-L and AD-L, the Initiator behind the EI NAT got once, both in the normal and in the quick mode, a peer reflexive address for the peer. Likewise, when the Initiator was behind a PD NAT, twice in the normal mode and once in the quick mode, the hosts ended up using a relay.

The same kind of variation in the selected path was happening in the NAT scenarios where the Initiator was behind the Linux NAT. Here, the Initiator used a peer reflexive candidate 3 times in normal mode and once in the quick mode for both EI and AD NATs. The PD NAT resulted in use of a relay 3 times in the normal mode and twice in the quick mode. When both hosts were behind a Linux NAT, the Responder's relay was always used.

4.4.2 Number of Messages

Different NAT scenarios required a different number of ICE connectivity check messages before the prototype was able to nominate a candidate pair and stop the ICE process. Also, there was some variation in the number of messages that was needed when the test was repeated multiple times using the same NAT setting. Figure 4.4 depicts the average number of messages sent during the ICE connectivity checks in scenarios where both of the hosts were behind a NAT. The x-axis contains the type of the NAT scenario where the Initiator's NAT type is before the hyphen and the Responder's NAT type is after the hyphen. On the y-axis, there is the average amount of messages that was sent by either the Initiator or the Responder. These figures do not include the candidate gathering phase but only the connectivity checks, including final the nomination message exchange. None of the scenarios had retransmissions of any of the messages.

In all the scenarios where both hosts were behind different NATs, after the gathering phase, both hosts announced 3 candidates to each other: host, server reflexive and relayed. When these are paired, they result in 9 candidate pairs. Then, because a host cannot really

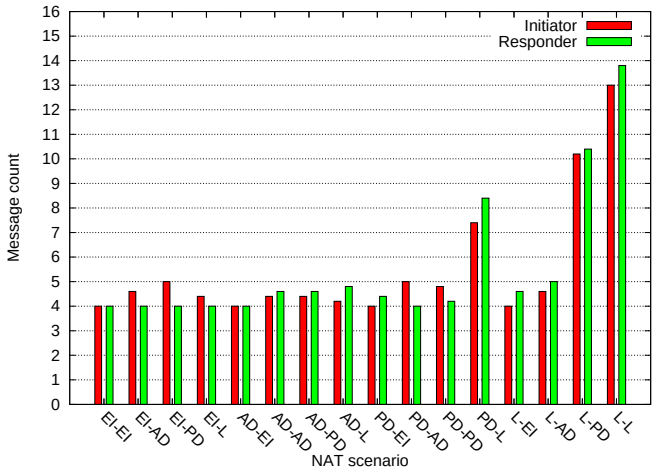

Figure 4.4: Average number of messages sent during the ICE connectivity checks

send anything from its server reflexive address, but the checks are in reality always sent from the host address, some of the candidate pairs are redundant and were removed from the list. The removed pairs are selected by finding candidate pairs whose remote candidate is the same but other pair has a host and the other pair a server reflexive local candidate. The server reflexive candidate is replaced by the host candidate and the lower priority pair of these two is removed. This process, called *pruning* [46], causes 3 candidate pairs (S-H, S-S, S-R) to be removed from both checklists, resulting in checklists where there were 6 candidate pairs for both of the hosts.

In the scenarios where only one of the hosts was behind a NAT, the hosts ended up with different size checklists. Because the host without a NAT had only 2 candidates (host and relayed) the host with a NAT had 4 entries in the checklist: from its host and relayed address to the host and relayed address of the peer. On the other hand, the host without a NAT paired the two candidates with host, server reflexive and relayed candidate of the peer resulting in 6 pairs. If neither one of the hosts had a NAT, there were no server reflexive candidates to prune and both had 4 entries in the checklists.

The scenario where both of the hosts are behind a NAT that has endpoint-independent filtering behavior needed constantly only 4 messages sent by both of the hosts. The other scenarios, where the filtering behavior of the NATs was more strict, needed sometimes one more message sent by either the Responder or the Initiator before a working path was created. An exception to this was the AD-EI scenario which, like the EI-EI scenario, needed in all the tests only 4 messages by both of the hosts. The scenarios where also the mapping behavior changed, i.e., the Linux NAT scenarios, resulted in more messages, especially if

the other host was behind a Linux NAT and the other either behind another Linux NAT or a NAT whose filtering behavior is port-dependent. On average, the PD-L scenario required 7 messages from the Initiator and 8 messages from the Responder, whereas L-PD scenario required 10 messages from both. The L-L scenario required on average 13 messages from the Initiator and 14 from the Responder.

Figure 4.5 contains the message count statistics for the scenarios where either only one or neither one of the hosts was behind a NAT. Unlike in the scenarios with NATs, the amount of messages was the same in all the test runs, so the values are not averages.

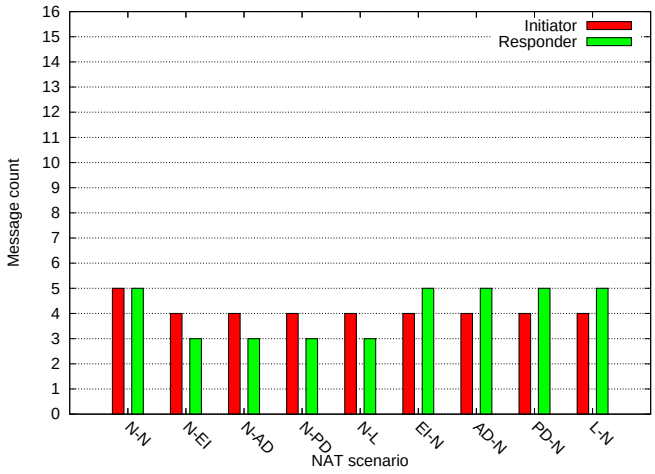

Figure 4.5: Number of messages sent in non-NATed scenarios

If neither one of the hosts was behind a NAT, both of the hosts ended up sending 5 connectivity check messages to each other. In all the other scenarios the Initiator of the connection sent 3 messages, but when the Responder was behind a NAT, it sent 3 messages, and when the Responder was not behind a NAT, it sent 5 messages during the connectivity check phase.

When the quick mode was used, the trend in the amount of messages was fairly similar to the normal mode, but instead using 4–5 messages, the average message counts were 5–6 in most of the scenarios with both hosts behind NATs. The PD-L scenario had the same average amount of messages as in the normal mode, whereas L-PD scenario used on average one more message from the Initiator and the Responder. The L-L scenario used on average 2 more messages from the Initiator and 4 more messages from the responder compared to the normal mode. If both of the hosts were in NAT free environment the quick mode used, like the normal mode, 5 messages from both of the hosts. If only one host was behind a NAT, the host without a NAT had to always send 5 messages whereas 3 messages

for the Responder and 4 messages for the Initiator was enough. The figures from the quick mode are shown in Appendix C.

4.4.3 Traffic Volume

Table 4.3 shows the average, minimum and maximum amount of bytes sent during the ICE connectivity checks in different scenarios, rounded to integer precision. The byte counts are almost directly proportional to the amount of messages sent during the connectivity checks, since a normal binding request is always 144 bytes and a response is 148 bytes. Also the last binding request message, which nominates the best pair, is 148 bytes.

Table 4.3: Amount of bytes sent during the ICE connectivity checks

Scenario	Initiator	min	max	Responder	min	max
EI-EI	584	584	584	584	584	584
EI-AD	670	584	728	584	584	584
EI-PD	728	728	728	584	584	584
EI-L	642	584	728	584	584	584
AD-EI	584	584	584	584	584	584
AD-AD	642	584	728	670	584	728
AD-PD	642	584	728	670	584	728
AD-L	613	584	728	699	584	728
PD-EI	584	584	584	642	584	728
PD-AD	728	728	728	584	584	584
PD-PD	699	584	728	613	584	728
PD-L	1096	584	1988	1314	728	2264
L-EI	584	584	584	670	584	728
L-AD	671	584	876	729	584	876
L-PD	1592	728	2216	1606	584	2348
L-L	1994	1808	2184	2215	2084	2412

The smallest amount of bytes was sent in the scenarios where both the Initiator and the Responder had to send only 4 messages. These scenarios resulted in 584 bytes sent by both of the hosts. On average, the highest amount of bytes was sent in the scenario with only Linux NATs: 1994 bytes for the Initiator and 2215 bytes for the Responder. However, the highest single amount of bytes sent for the Initiator was in the L-PD scenario with 2216 bytes, whereas for the Responder it was in the L-L scenario with 2412 bytes. If we do not take the Linux scenarios into account, the amount of bytes sent by either one of the hosts was always between 584 and 728 bytes.

Table 4.4 shows the amount of bytes sent in scenarios where both hosts were not behind a NAT. Because there was no variation in the amount of messages sent in these scenarios, the values are exact rather than averages.

Table 4.4: Amount of bytes sent in non-NATed scenarios

Scenario	Initiator	Responder
N-N	732	732
N-EI	584	440
N-AD	584	440
N-PD	584	440
N-L	584	440
EI-N	588	728
AD-N	588	728
PD-N	588	728
L-N	588	728

Also in these scenarios the amount of traffic both endpoints generate during the connectivity checks is between 584 and 728 bytes; except for one scenario. The highest amount of traffic is generated in the scenario without the NATs: this scenario exceeds, with 732 bytes sent by both of the hosts, even the highest amount of bytes in a scenario with two other than Linux NATs.

In the quick mode, scenarios where both of the hosts were behind NATs, excluding the PD-L, L-PD and L-L scenarios, the average amount of bytes sent was in the range of 728–843 for the Initiator and 728–872 for the Responder. The L-L scenario needed the most sent bytes with average of 2384 for the Initiator and 2904 for the Responder. In the non-NATed scenarios, the Initiator sent 728, 732 or 588 bytes and the Responder 440, 728 or 732 bytes depending on the amount of sent messages.

The candidate gathering phase, which should be done with a single message to the TURN server, required two messages in our implementation. If we disregard the redundant message, the gathering message exchange required sending a 28 byte request and receiving a 92 byte answer message from the TURN server.

4.4.4 Check Durations

Figure 4.6 shows the time it took in different NAT scenarios from the start of the connectivity checks before the first connectivity check by the Initiator succeeded. In other words, if aggressive nomination was used, this is the time it would take to nominate the first pair. The average time is shown with the cross and the error bars show the minimum and maximum times from the 5 test runs.

Almost in all the scenarios where both of the hosts are behind a NAT, it takes approximately 0.5 seconds before the first successful check is done. Only in three scenarios, where there was an address and port-dependent filtering NAT with a Linux NAT, or two Linux

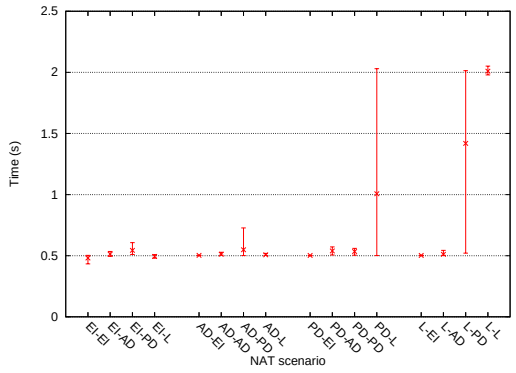

Figure 4.6: First successful test's time

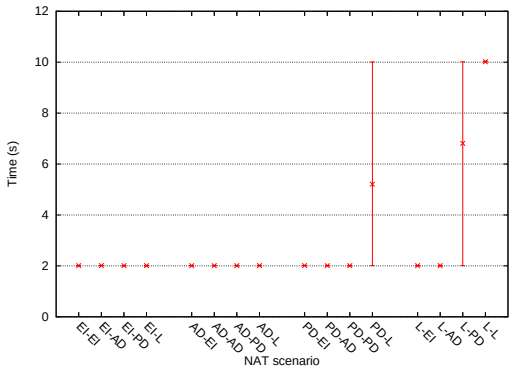

Figure 4.7: Time when all checks are done

NATs, it took on average much longer than 0.5 seconds. With the PD NAT, the fastest success was after 0.5 seconds, but in some test runs it took up to 2 seconds before the first success. With two Linux NATs it took constantly 2 seconds before the first successful check was done.

Figure 4.7 shows the total time the whole ICE connectivity check process took from the start of the connectivity checks to the reception of a response to the nomination message. Like in the previous figure, most of the scenarios resulted in same time, 2 seconds, whereas the PD-L, L-PD scenarios experienced some variance with some of the test runs taking 2 seconds to complete while for some it took up to 10 seconds. The L-L scenario needed always 10 seconds before the best pair was nominated.

As shown in Figure 4.8, in all the scenarios where only one of the hosts was behind a NAT, the first successful check was done in average during the first two milliseconds. The longest time was in the scenario where the other host was behind a Linux NAT with the check taking 8ms. Essentially, in all these scenarios the first successful check was done right after the start of the checks. However, in scenarios where the responder was behind a NAT, the checks continued until 2 seconds (Figure 4.9), whereas in other scenarios the checks were stopped right after nominating the first successful candidate pair.

The quick mode results were almost identical except that all the times were $1/5$ of the times in the normal mode. Only additional difference was that the final success times in the L-AD and L-L scenarios had slightly more variance in the quick mode: in the normal mode there was hardly any variance whereas in the quick mode L-AD scenario's first success time varied between 0.1 and 0.18 seconds and L-L scenario's time between 0.3 and 0.5 seconds.

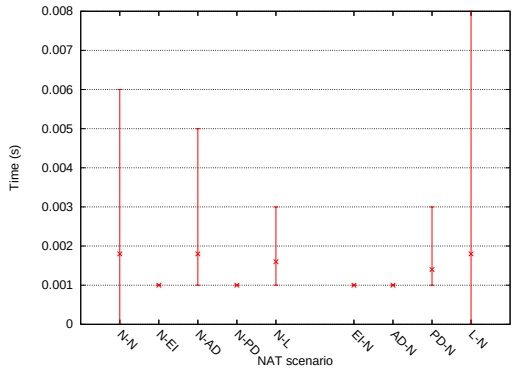

Figure 4.8: First successful test's time in non-NATed scenarios

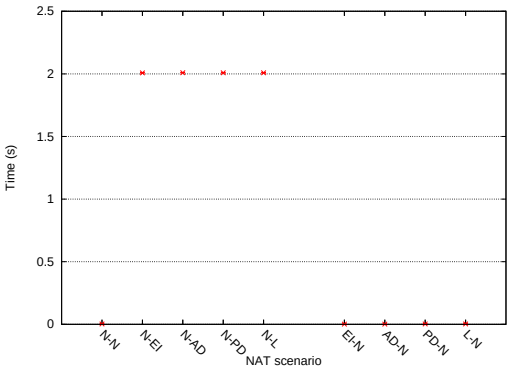

Figure 4.9: Time when all checks are done in non-NATed scenarios

4.4.5 Two Hosts in the Same Subnet

We also run tests with two hosts in the same subnet. When the hosts had only one interface to the subnet, the results were identical to the N-N scenario: both hosts had to send 5 messages (732 bytes) and the hosts-to-host candidate pair was nominated immediately.

However, when we changed the Initiator to be multihomed, i.e., enabled an interface towards the public network at it, the results changed. Now, both sent 4 connectivity check messages in the quick mode and the Initiator needed to always send 3 and the Responder 4 messages in the normal mode. Even if the message count was lower, finishing the checks took always roughly 2 seconds in the normal and 1 second in the quick mode.

4.5 Measurement Analysis

The results of Table 4.2 show that ICE is able to find the best path between two hosts if they have only endpoint-independent mapping NATs between them, regardless of their filtering behavior, as discussed in Section 4.1. The only variation is due to the use of Linux NAT which does not fit nicely into the filtering and mapping behavior categorization.

In the scenarios where one host was behind a Linux NAT and another behind an EI or AD NAT, the host behind the Linux NAT ended up sometimes using a peer reflexive candidate. This happened because a check sent by the host behind EI/AD NAT reached the Linux NAT before the other host had started its own check towards the server reflexive address at AD/EI NAT. As described in Section 4.3, in this case the Linux NAT uses a different mapping for the outgoing check resulting in new peer reflexive address. If the other NAT's filtering behavior had been address and port-dependent, this check would have failed and only a relayed path would have been possible as in the L-PD and PD-L scenarios.

If both of the hosts are behind a Linux NAT, it does not matter which one of the hosts starts the checks since the other host's NAT will switch to endpoint-dependent mapping behavior and the other Linux NAT's filtering behavior does not accept the check coming from the new port. Therefore, in this case the relayed path is the only one that will work as was seen with the L-L scenario.

4.5.1 Number of Messages

The number of messages sent during the ICE connectivity checks depends a lot on the stopping criteria, but even if the criteria were the same, some scenarios needed on average more than double the amount of messages before the checks were done. The least messages were exchanged in the scenario where both hosts were behind endpoint-independent filtering and mapping NATs. This sets the baseline for the amount of messages in scenarios where the hosts are in different subnets, behind different NATs.

NATs with endpoint-independent filtering behavior

In this scenario, both hosts first probe for connectivity on the host to host candidate pair (H-H). The check does not succeed, since the hosts are in different subnets. After waiting for the check pacing interval, T_a, they start the checks on the second highest candidate pair in the list: local host to peer's server reflexive (H-S). This check succeeds since the peer's NAT's filtering behavior allows the check message to pass. The peer that first receives the check, P_a, answers to it and starts a triggered check on the same candidate pair. When the answer to the check is received by the other peer, P_b, it notices from the mapped address attribute that the check came in fact from the server reflexive address and marks the server reflexive (S-S) candidate pair as validated. Also the H-S pair's state changes to SUCCEED since it produced a valid pair.

Because a check on the server reflexive candidate pair succeeded, all lower priority checks are cancelled and only the host candidate pair (H-H) check continues. When the triggered check arrives to the other host, it responses to it and also the peer P_a learns that the server reflexive pair works. Now both hosts have sent 3 messages.

Since the controlling host had initially 6 different candidate pairs in the checklist, it used an RTO value of 3 seconds ($6 \times 500ms$), and the soft deadline is reached before the retransmissions start. Therefore, it nominates the only validated pair by sending another check on the server reflexive candidate pair with the nominated flag. The controlled host responses to it and when the controlling host receives the response, ICE processing stops, resulting in 4 messages sent by both of the hosts.

NATs with other filtering/mapping behavior

If the NAT filtering behavior is endpoint-independent only for one host, the peer behind the endpoint-independent filtering NAT, P_{ei}, ends up sometimes sending one more message than in the EI-EI scenario. If P_{ei} sends the first check before the other host (P), unlike in the EI-EI scenario, the NAT in front of P does not let the check pass because of the filtering behavior. However, a check sent by P is delivered to P_{ei} and it causes P_{ei} to cancel and start a new check for that candidate. This new check passes because the check already sent by P created a proper binding in its NAT. P answers to the check and now P has sent 3 messages and P_{ei} 4 messages. The final nomination happens like in the EI-EI scenario.

In scenarios where the mapping behavior of P's NAT is not endpoint-independent, like in the L-EI and EI-L scenarios, the check sent by P comes from a different port than where the check by P_{ei} was sent. Because of the loose filtering behavior of P_{ei}'s NAT, this check passes and presents a new peer reflexive candidate for P_{ei}, but does not increase the amount of messages that need to be exchanged. If P sent the first check, everything happens like in the EI-EI scenario, and both hosts send only 4 messages. Because the order in which the hosts started the checks varied between the test runs, P_{ei} sent 4 or 5 messages depending on the exact timing.

In case both hosts had a NAT with either address or address and port-dependent filtering behavior, it was always the host who started the checks first who had to send the extra check. This happens because there is no binding for the faster host's connectivity check before the slower hosts sends its first check towards the peer.

The checks coming from a new peer reflexive address, due to Linux NAT mapping behavior, pass the other NAT as long as the NAT filtering behavior is not address and port-dependent. Otherwise, a relay is needed which results in more connectivity check messages. Because the relayed path was not accepted before 10 seconds had passed, the peers sent 10–15 messages while trying the higher priority candidate pairs with retransmits even after a successful check on the relayed candidate pair.

Hosts without a NAT

It was not surprising that the scenarios with more strict filtering or endpoint-dependent mapping behavior NATs required more messages from the endpoints than the EI-EI scenario. Instead, the fact that the scenario where there were no NATs at all (N-N), required more messages than any of the scenarios where only one of the host was behind a NAT, was rather unexpected.

An explanation to this behavior lies in the way how ICE handles candidate pairs which have started the connectivity checks but have not received an answer back yet, i.e., pairs

that are in the PROGRESS state. As mentioned in Section 2.4.2, an incoming check on a candidate pair in PROGRESS state causes ICE to cancel the ongoing transaction and create a new one. When two hosts without NATs start the checks, they send a binding request to each other's host candidates. When both of the hosts have send their own checks, they receive the check from their peer and note that it matches the candidate pair where they just started a check from. Therefore, they cancel the original check and send a new check on the same pair. After sending the check they process the next incoming packet which is answer to the check they just cancelled. This validates the pair and no further checks are needed on that pair. However, they both already sent an extra check to which the peer has to answer. So, both hosts send two checks, have to answer to two checks, and still need to nominate the only validated pair, resulting in 5 connectivity messages in total.

In this light it seems strange why a check in the PROGRESS state is cancelled in case of incoming check. Nevertheless, it makes sense since a common reason for the lack of a response before an incoming check is that the original request was stopped by the peer's NAT because of missing mapping or filtering rule. The check that was received on the pair in the PROGRESS state would have created such a rule and the new check should pass the NAT.

If our implementation had the triggered check queues, the new check would not have been sent immediately and a response to the original check would have removed the triggered check from the queue. Yet, if RTT is bigger than $2 \times (d + T)$, where d is the delay between the hosts for starting the checks and T is a value in the range of $[0 - T_a]$ (the time before the check sending timer fires), the response to the original message would not be received before the check from the triggered queue starts. This is shown in Figure 4.10. The response (R1) to the first check (C1) is received after a response (R2) to the check (C2) sent by the peer and a triggered check (TC3) due to the incoming check are sent. This happens if the time from the sending of the C1 to the time when TC3 is sent is shorter than the time from the sending of the C1 to the reception of R1 (see equations 4.1 – 4.3). The actual value for T depends on when the previous connectivity check message was sent: if the check was sent T_c ago, $T = T_a - T_c$.

$$d + RTT/2 + T < RTT/2 + RTT/2 \quad (4.1)$$

$$d + T < RTT/2 \quad (4.2)$$

$$RTT > 2 \times (d + T) \quad (4.3)$$

This behavior implies that beginning the checks simultaneously can actually hurt the performance. If the check pacing is implemented as the ICE specification suggests,[3] the

[3]This was not observed in our measurements since we do not pace the triggered checks

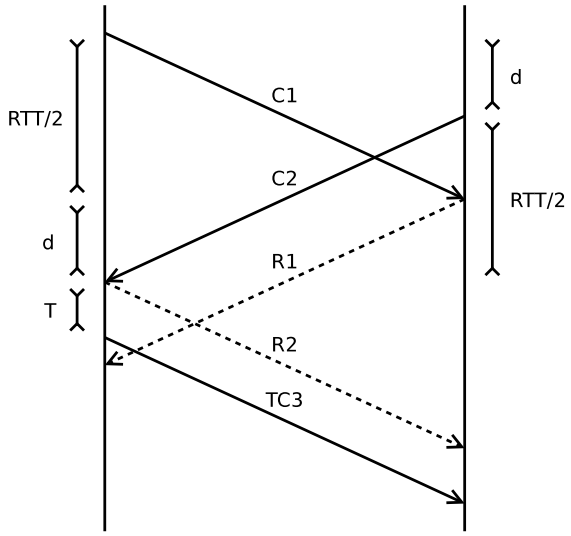

Figure 4.10: Situation resulting in an extra connectivity check

endpoints do not only need to send an extra check, but the controlling endpoint may also need to wait up to T longer than without the triggered check before it gets to nominate the best pair. With the recommended T_a value of 500ms, this means up to half a second delay.

A similar situation occurred in the scenarios where only the Initiator was behind a NAT. The only difference was that now only the Responder ended up sending 5 messages and the Initiator had to send only 4 messages.

4.5.2 Traffic Volume

The volume of the traffic generated by the ICE process naturally depends on the amount of messages. However, the size of a single message can vary a bit depending on the included STUN attributes.

Size of the connectivity check messages

A breakdown of a connectivity check message used in our implementation is shown in Figure 4.11. As noted in Section 2.3.2, all the STUN messages start with a 20 byte header containing the type and length of the message, the magic cookie, and a transaction identifier. In addition to the header, the connectivity check request messages contain also priority, username, integrity and fingerprint TLV attributes. All the attributes have a 4 byte header, followed by a value whose length depends on the type of the attribute. The priority attribute has a 4 byte priority value for peer reflexive candidates (see Section 2.4.2), the integrity at-

tribute has a 20 byte and the fingerprint a 4 byte checksum value [49]. Finally, the username attribute contains a variable length value presenting the usernames of both of the peers.

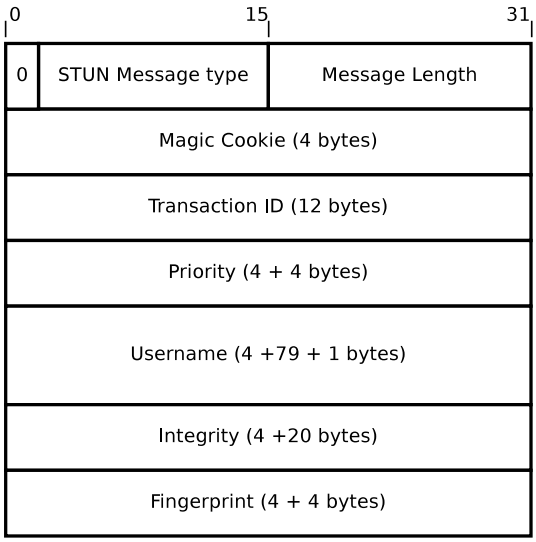

Figure 4.11: ICE connectivity check request message structure

The ICE specification [46] says that the connectivity check messages should also contain either ICE-CONTROLLED or ICE-CONTROLLING attribute that indicates the ICE role, controlled or controlling, the host believes it should be in. This attribute contains a 64 bit unsigned integer value that is used as a tie breaker if both endpoints believe they are in the same role. As noted in Section 3.4.1, this cannot happen with HIP, so these attributes are not included in the connectivity check messages. If they were, they would increase the size of all the check messages by 12 bytes.

In the tests we used the peer HITs as the usernames. A single HIT in the STUN username presentation consists of 8 groups of 4 hexadecimal characters which are delimited with a colon. For the STUN username, the two HIT presentations are concatenated and separated with a colon, resulting in 16 groups of hexadecimal characters and 15 colons. Since each character takes one byte, in total the username takes $16 \times 4 + 15 = 79$ bytes. In addition, the username TLV contains one byte of padding because the attribute must be aligned on a 32 bit boundary [49].

The only difference between a request and a response message is that the response message contains a mapped address attribute instead of the priority attribute. The mapped address attribute takes 12 bytes, i.e., 4 bytes more than a priority attribute. Also, the connectivity check message that nominates the best pair contains a nominating attribute which does not have a value and thus takes only 4 bytes. Hence, size of a normal connectivity

check request is $20 + 8 + 84 + 24 + 8 = 144$ bytes and a response or nominating request takes $144 + 4 = 148$ bytes.

Because of the rather long usernames, the username part takes roughly 60% of a whole message, hence using more than half of the total bytes exchanged during the connectivity check phase. If the byte overhead of the connectivity checks should be decreased, reducing the username length would be a good candidate for this. The HITs are natural usernames in the context of HIP, but e.g., a hash of the HIT would work equally well. Also, a more efficient coding of the HIT could be used: now every byte is presented with two bytes since one hexadecimal character can only present 4 bits of information. For example, if both of the usernames were presented with 5 character, the username attribute would take in total $4 + 5 + 1 + 5 + 1 = 16$ (header, username, colon, username and padding) bytes instead of 84 bytes. Then, a request would take only 76 bytes and a response 80 bytes.

Bitrate of the checks

Since most of the scenarios needed 2 seconds and 584–728 sent bytes to finish, the average upstream bitrate for these checks would be from $\frac{584 \times 8b}{2s} \approx 2.3\frac{kb}{s}$ to $\frac{728 \times 8b}{2s} \approx 2.9\frac{kb}{s}$. The L-L scenario needed the most bytes, but also 10 seconds, to finish resulting in bitrate of $\frac{2113 \times 8b}{10s} \approx 1.7\frac{kb}{s}$.

This bitrate is well below even the 9.6 kbps bandwidth of basic GSM (Global System for Mobile communications) data, and newer GPRS (General Packet Radio Service) connections can achieve orders of magnitude higher bitrates [39]. The quick mode sent packets 5 times faster, but since the amount of sent messages was not much higher, the consumed bitrate was approximately only from $5.8\frac{kb}{s}$ to $6.7\frac{kb}{s}$. This would be closer the maximum basic GSM bitrate, but would not still be an issue for faster connections. Also, if we used shorter usernames, as described before, the size of the connectivity check messages, and therefore the needed bitrate, could be almost halved and also the quick mode would be easily manageable even with the basic GSM bandwidth.

Taking these issues into consideration, the connectivity check traffic should not be too much for even modest range practical link speeds. Of course, if a link is already highly utilized, even this amount of traffic can cause it to become congested and start dropping traffic. Still, even the quick mode creates so little traffic that it should be safe to use it with most of the access technologies in use today.

For all the scenarios, except when there are no NATs at all, the used downstream bitrate is smaller than the upstream bitrate, since only some of the checks make it through the NATs or are even sent to a routable address. For example, if both of the hosts are behind a NAT, the first checks never make it to the peer's NAT, since the private range host address is not routed there. Also, since the first check that makes it through causes the lower priority checks to cancel, only couple of connectivity check messages are usually received.

4.5.3 Check Durations

In 9 out of the 12 scenarios, where both of the hosts were behind a NAT, the first successful check was in practice always done 0.5 seconds after the start of the checks. This check was always the second check in the checklist: from host to server reflexive candidate of the peer, and hence done 500ms (T_a) after the start of the checks. The first check between the host candidates was always doomed to fail since the hosts were in different subnets. Even if the first check to server reflexive address was in some scenarios dropped due to missing filtering rule in AD or PD NAT, the check to other direction would succeed and create a triggered check to other direction, keeping the success time close to 500ms.

In the PD-L and L-PD scenarios, depending on the timing, either one of the server reflexive candidate worked and the first successful test was after 0.5 seconds, or only the relayed candidate worked after 2 seconds as in the L-L scenario. The dependency on the check timing is due to the Linux NAT behavior discussed in Section 4.3.

However, things got more interesting when both of the host were behind a Linux NAT or only the Responder was behind a NAT.

Both hosts behind Linux NATs

Only in the scenario with two Linux NATs, it took constantly 2 seconds before the first successful check, because the fifth candidate pair was the first one to succeed. One would have expected already the third candidate pair of the Initiator, host to relayed candidate, to yield success, but this was not the case. If the Initiator happens to send the first check, the Responder's TURN server drops the message because there is no permission for it, as discussed in the section 2.3.3. Also, when the Responder performs its own connectivity check on the same pair, i.e., from its relayed candidate to the peer's host candidate, it actually asks the TURN server to forward the check to a private range host address of the peer, which naturally does not work. And to make things even worse, this check does not create a proper permission for the checks sent by the peer, since the peer's checks appear to come from the server reflexive address. Consequently, even if the Initiator sends the check after the Responder, the TURN server drops it.

For the fourth candidate pair both endpoints just use the Initiator's relay, and also these checks fail for the same reason. The fifth candidate pairs, relay to server reflexive address (from Initiator's point of view) work since now a permission for the right address is finally created.

Clearly, the problem here is the TURN server's address-dependent filtering behavior due to the need of permissions. This feature was included in the specification to ease concerns of enterprise network system administrators that TURN could be used to bypass their firewall

security measures [48]. However, in many scenarios, more important than the (somewhat limited) security benefit this feature brings, would be to finish the connectivity checks as fast as possible. Thus, it would make sense to be able to create a relayed candidate that accepts any incoming traffic without a need for explicit permission.

Only one host behind a NAT

In the scenarios where only one of the hosts was behind a NAT, the first check done by the host that is behind a NAT always succeeds since there is no NAT before the peer that would block the check. This check triggers also a check from the peer and both hosts perform successful checks right at the beginning of the connectivity checks. Still, the scenarios where the Initiator was not behind a NAT continued checks until the soft deadline, while the other scenarios with only one NAT, or no NATs at all, resulted in almost immediate nomination and ending the checks.

The reason for this behavior is that the highest priority candidate pair, host to host, moves to SUCCEEDED state for the peer behind a NAT when it performs the first check: the peer sees that the check actually came from the server or peer reflexive candidate, it creates a valid pair from that, but the original pair is still marked as SUCCEEDED. Yet, the peer that is not behind a NAT is unaware that the checks for this pair can be stopped, and it continues until the soft deadline.

This issue could be fixed by adding an attribute to the connectivity check messages which tells from which candidate the check was sent. Then, also the controlling host without the NAT would know which pair succeeds from the peer's point of view, and it would not continue trying to test a path that has no chance of ever working.

4.5.4 Two Hosts in the Same Subnet

The scenario where both of the hosts were in the same subnet with only one interface is fairly similar to the scenario where neither one of the hosts has a NAT since the best path between them is free of NATs. Therefore, it is natural that the results were identical to the N-N scenario. The only difference between these scenarios is that in the shared subnet scenario the hosts have also server reflexive candidates (from the same NAT), but they never get to try them since the host-to-host candidate pair is prioritized over pairs with the server reflexive candidate.

However, when the interface to the public network was enabled for the Initiator, the scenario changed. Because of local prioritization, the Initiator prefers the public interface over the private one. Consequently, the highest priority path would be from the host candidate of the preferred, public interface rather than from the private interface. Since a check sent

from the public interface to the peer's most preferred (host) candidate with a private range address is not delivered, this path never works. Because our check stopping algorithm tries the highest priority path until the soft deadline is passed, the checks are not stopped even while a check on another host-to-host candidate (from the Initiator's private interface) succeeds. Hence, the checks always take at least 2 seconds in the normal and 1 second in the quick mode.

This problem would not have occurred if the private interface was preferred over the public one. Still, if the peer had been accessible only through the public interface, we would have ended up to the same situation. Probably the best solution would be to use a more intelligent stopping criteria which immediately accepts a path from the less preferred interface too. However, there could be a reason for the user to prefer one interface over the other. For example, if there is a difference in the cost of using the interfaces or the other interface is using a more reliable or faster connection, it makes sense to keep trying checks on the preferred path. If this is not the case, it would be useful to have multiple candidates with the same preference, but this is not possible since, according to the ICE specification, all the candidates need to have unique priorities.

For this reason, the local prioritization of interfaces and candidate types is not always an adequate way for expressing what kind of candidate pair should be selected. This further increases the importance of the stopping criteria which should not make decisions based only on the priorities.

4.5.5 Quick Mode

The shorter value for T_a and check stopping deadlines of the quick mode essentially halved the time it takes to create a path between two hosts. As noted in Section 3.4.3 and in [24], this is often a rather important metric in case of interpersonal communication.

Since the first successful check with quick mode was made in most of the scenarios five times faster than with the normal mode, a more aggressive check stopping criteria would have been able to set up the path using roughly only one fifth of the time it takes for the normal mode. Yet, there was not much more traffic generated by the quick mode. Even the small increase in the amount of sent messages was due to faster retransmission, which in turn could be able to create a more optimal path in case of packet loss. Also, as discussed in Section 4.5.2, even the bandwidth usage of the quick mode is not considerably high.

The situation could be different in a network with longer RTT, since the retransmissions could start before the first checks have even made it to the peer's NAT. Still, it seems that the connectivity check pacing should be as small as reasonable, because the NATs where they should create bindings on are usually quite close to the peer sending the checks, and therefore the (possibly long) RTT between the peers is not relevant there. Also, the mini-

mum limit of RTO (500ms) makes sure that even with a small T_a value the retransmissions do not become too aggressive.

Sending checks really fast could also cause congestion in the network, but on the other hand, if the path created with ICE is not used by a protocol that tries to avoid congesting the network, the traffic after the checks will likely anyway cause much more congestion than the connectivity check messages. Of course, the congestion caused by sending the checks too fast may cause a connectivity check message being dropped and therefore a less direct path being selected.

Still, based on these observations, the timer values of the quick mode seem much more reasonable than the values suggested by the ICE specification for non-RTP traffic.

4.5.6 Generality of the Measurement Results

The measurements were performed in our prototyping network which has a very low average RTT (less than 1ms) compared to communication through the Internet. Therefore, some of the effects that are due to checks happening simultaneously will likely be different on the real Internet. The signaling path through the TCP relay is much faster than a HIP four-way base exchange through a relaying service and hence both endpoints start the connectivity checks almost simultaneously. On the other hand, since the checks reach the other host, or its NAT, much faster than they would do in the Internet, the probability that two checks cross in the network is lower. This way our prototyping network can make simultaneously happening connectivity checks both more and less probable.

Also, the prototyping network was fairly reliable and not congested, so we did not test the behavior of ICE when some of the connectivity check messages are not delivered due to packet loss. Since losing packets would lead to retransmissions, and possibly to selection of sub-optimal paths, many of the results could be different if our implementation was tested in a less reliable network.

Finally, since the stopping criteria of the checks plays a major role in the ICE performance, with different criteria, the amount of messages that are needed would be quite different. We implemented only one simple algorithm for this and a more advanced algorithm could be able to perform better. Similarly, different local prioritization of the candidates could decrease, or increase, the time it takes to create a path. For example, a host that knows that the peer is not in the same subnet could prioritize a server reflexive candidate over the host candidates and this way find a working path using the first pair in the checklist instead of trying to the host-to-host candidate pair first.

4.6 Summary

We analyzed ICE behavior theoretically, and concluded that no TURN relay is needed as long as there are no address-dependent mapping NATs between the hosts. Also, if one of the hosts is behind such a NAT, but the other host does not have a NAT that has address (and port) dependent filtering behavior, a path without a relay is possible. This behavior was also validated by measurements performed on the prototype. In addition, complications that may rise with multiple layers of NATs, and a remedy to the problem was presented. Section 4.3 contained an analysis of unexpected Linux NAT behavior, where the NAT may switch from endpoint-independent to address and port-dependent mapping behavior, depending on the connectivity check timing. The measurements showed that this behavior can hurt the connectivity check phase a lot.

Our ICE library was tested in 27 different NAT scenarios, the amount of messages and bytes sent during the checks was counted, and the time it took for the checks to succeed was measured. The results from different scenarios were presented and analyzed and some unexpected results were discussed in more detail. Especially, the scenario where the hosts are not behind NATs got special attention and we derived a relation between the RTT and the check timing that dictates whether a redundant connectivity check is performed. From the number of transferred bytes we derived bitrate of the connectivity check traffic and concluded that the bitrate is manageable even with modest bandwidth connections. Our analysis on the check duration revealed a feature of TURN that causes the checks between hosts with endpoint-dependent mapping NATs to take surprisingly long time. Also, if only the Responder was behind a NAT, we found out that ICE continues the checks longer than necessary. A fix to both of these problems was presented after the analysis. Finally, when two hosts were put on the same subnet, but the Initiator was multihomed, we discovered that hosts were able to create a path, but it took longer and the path was different than expected.

From the measurement we found out that the connectivity checks commonly required hosts to send 4–5 messages and 600–750 bytes, but NATs with endpoint-dependent mapping could double the amount of needed checks. A version of the implementation with shorter timer values, called quick mode, was able to create paths with similar amount of traffic, but using only from $1/5$ to $1/2$ the time the normal mode used. Quick mode had higher bandwidth requirements, but still the generated bitrate was not excessive. Also, from the connectivity check message structure's analysis we found out that the amount of bytes transferred during the checks, and thus also bandwidth requirements, could be halved by simply using shorter usernames.

Chapter 5

Discussion

In this chapter we discuss the applicability of HIP to P2PSIP NAT traversal: what is needed from the P2PSIP, how much overhead will be caused by HIP and ICE, and how HIP's features can be beneficial for P2PSIP. We also present ideas for future work on how ICE could be improved and also how we could make our evaluation of ICE more complete.

5.1 HIP as a NAT Traversal Solution for P2PSIP

As noted in Section 3.1, if HIP is used for creating connections between hosts, the applications do not need any other NAT traversal solutions, but they can use the path created by HIP for communication. For this reason alone, HIP appears to benefit P2PSIP considerably. However, this benefit does not come without a cost since HIP increases the communication overhead, i.e., the amount of time it takes to set up a path and also the amount of extra traffic that is generated. Also, while P2PSIP can benefit from HIP, it can also help HIP's NAT traversal work in situations where there are no HIP Relays available. These issues are discussed in the following subsections.

5.1.1 Using P2PSIP Overlay with HIP

HIP needs a signaling path for the base exchange to be able to traverse NATs. In Section 3.2 we introduced a HIP Relay for this purpose, but in many scenarios such a relay may not be available or known to the hosts. Instead, the relaying service can be provided by the P2P overlay. [8] proposes using hop-by-hop routing of the base exchange through the overlay for this purpose. This solution traverses NATs effortlessly, since the base exchange uses the same path as the overlay network traffic. A downside of this approach is that the RTT over the overlay can be relatively long since even a single hop in the overlay usually takes

multiple hops in the underlying network. Hence, concluding the base exchange may take considerably longer compared to using a single HIP Relay.

Another option would be to use a single overlay host that is not behind a NAT, or a host that is behind a NAT with endpoint-independent filtering and mapping behavior, as a HIP Relay. If such a host is not available, the hop-by-hop routing could be used as a fallback option. Using a single host would be faster and use less resources from the overlay but would also create a possible single point of failure. However, since a relay is needed only during the base exchange, the probability that a peer disappears or fails during that time is relatively low.

A third option would be to use a TURN server as the relay for also HIP signaling traffic. One motivation for this is that since TURN is used also by other protocols than HIP, finding a TURN relay could be in many scenarios more probable than finding a HIP Relay. However, a Responder behind a NAT cannot use a TURN server for receiving I1 packets because of the required TURN permissions. Fixing this would require a change or an extension to the TURN protocol, but it would also fix the TURN problem discussed in Section 4.5.3.

While HIP can benefit from the P2PSIP overlay, the P2PSIP overlay can benefit even much more from HIP. An overlay that is using HIP for setting up the connections does not need to worry about NAT traversal: the only requirement is that it should be able to carry the base exchange packets through the overlay. Even this requirement is not necessary if there are HIP Relays that can be used for the base exchange. Also, due to the use of IPsec tunnels, no additional encryption or integrity protection is needed for the overlay traffic. The strong authentication features of HIP can be used to help admission control of the overlay, the mobility extensions can be useful for mobile hosts, and multihoming benefits overlay peers with multiple network interfaces. Adding this kind of features, especially the mobility and multihoming, in some other way and still make it secure and efficient could be problematic.

Relaying Data Using Overlay Peers

In a P2PSIP overlay network, peers may work as media relays for nodes that are behind P2P unfriendly NATs. However, due to the nature of the peers, they may disappear without a warning from the network or stop providing reliable relaying service for some other reason. This is especially bad for a relay since it is the only working path between the communicating nodes. Running the whole ICE procedure after noticing that the path does not work anymore takes often too long time and thus the working path should be made more reliable.

One solution for this is to use more than one concurrent relaying nodes. If we send traffic all the time through two or more relays, we multiply the chances that at least one relay can deliver the traffic. Unfortunately, this also multiplies the amount of network traffic which is especially bad in the relayed scenario with already non-optimal paths. Yet, an even bigger

problem is that most of the applications that would be using the relayed data do not support receiving data using multiple simultaneous paths. And even if such support was integrated to the applications, doing it in a reliable way that does not open possibilities for attacks discussed in Section 2.6.1, is not by any means trivial.

HIP can provide support for multiple simultaneous paths using the Simultaneous Multi Access (SIMA) [40] extension. With SIMA, HIP can send packets belonging to different flows between two hosts on different routes. This feature could be extended to replicate the traffic at HIP layer, send it using two (or possibly more) different paths, and on the other end drop duplicate packets. This way, a considerably more reliable path, that is still transparent to upper layers, could be created between the hosts, with the expense of quite high overhead.

5.1.2 Cost of HIP-ICE in P2PSIP

Using HIP with ICE for NAT traversal generates overhead for both connection setup and data traffic. The HIP base exchange and the ICE connectivity checks must be done before any data can be exchanged and all the data must be encapsulated in UDP and ESP during the connection.

Overhead due to the base exchange

The HIP base exchange's four-way handshake requires four messages through a relay or an overlay. From a packet capture (see Appendix D for details) it can be seen that a normal HIP base exchange requires the Initiator to send $40 + 680 = 720$ bytes (I1 and I2) and the Responder to send $600 + 216 = 816$ bytes (R1 and R2).

In addition to the normal HIP parameters, in case of ICE-based NAT traversal, the hosts exchange NAT transformation and LOCATOR parameters as described in Section 3.2. If the only defined transformation is used, the NAT transformation parameter requires 8 bytes. The LOCATOR parameter takes 4 bytes for the TLV header and 36 bytes for each ICE candidate [28]. A host behind a NAT with a single interface has typically 3 local candidates: host, server reflexive and relayed. Therefore, a typical LOCATOR parameter would be $4 + 3 \times 36 = 112$ bytes. With UDP encapsulation and non-ESP marker for both packets, after taking into account the NAT transformation and LOCATOR parameters, this gives ICE overhead of $2 \times (8 + 4) + 8 + 112 = 144$ bytes to the base exchange. In total, this means that the Initiator has to send roughly $720 + 144 = 864$ bytes and the Responder $816 + 144 = 960$ bytes during the base exchange.

Multiple layers of encapsulation

When HIP has created a path between the hosts, all the traffic between them is IPsec protected and hence encapsulated in ESP. For NAT traversal purposes, the ESP itself is encapsulated in UDP. Therefore, a protocol that is run on top of NAT traversing ESP uses multiple layers of encapsulation on every data packet as depicted in Figure 5.1.

Figure 5.1: Encapsulation of data in HIP initiated connections

The UDP encapsulation requires 8 bytes and the ESP header takes another 8 bytes. Because HIP uses the ESP Bound End-to-End Tunnel (BEET) mode [37], no additional IP header is needed and the transport layer header comes directly after the ESP header. The length of the ESP trailer depends on the selected security services: if no integrity protection or encryption is used, it is only 2 bytes long. For encryption purposes, the trailer may contain padding so that the data length matches to the requirements of the used cipher algorithm. Also an Initialization Vector (IV), that is needed for some cipher algorithms, may be included in the data portion before the trailer.

For example, 3DES (Data Encryption Standard) [27] uses 8 byte IV and requires the data to be a multiple of 8 bytes, needing padding of 0–7 bytes. If the packets are integrity protected, the ICV field contains the Integrity Check Value. For example, with a 160 bit Secure Hash Algorithm 1 (SHA-1) Hash Message Authentication Code (HMAC) [31], padded with 4 bytes to match the IPv6 alignment conventions, ICV needs 24 bytes.

If the ESP security features are disabled, the per packet overhead of NAT traversing ESP is $8 + 8 + 2 = 18$ bytes (UDP header, ESP header and a minimal ESP trailer). If the data is encrypted with 3DES and integrity protected with the SHA-1 HMAC, this adds $8 + 24 + p = 32 + p$ bytes (IV and HMAC) where p is the amount of needed padding. Hence, the total per packet overhead with the example security features is from 50 to 57 bytes. Yet, if HIP is regarded only as a NAT traversal solution, that is still able to perform mobility and multihoming, the per packet overhead can be considered to be 18 bytes.

As a point of reference, this overhead could be compared to NAT traversal with Teredo [21] which uses UDP encapsulation like HIP, but instead of ESP, encapsulates the data in IPv6. A minimal IPv6 header is 40 bytes, resulting in per packet overhead of 48 bytes. Also, the overhead created by the security features of ESP is similar to other security protocols providing integrity and confidentiality, such as TLS [13].

A path for UDP traffic created with plain ICE does not have any per packet overhead, so it is naturally more efficient than the ESP encapsulated path created with HIP. Still, whether the ESP overhead is significant, depends on the traffic that is sent on the path: for average packet size of 1kB, the ESP overhead would be less than 2%, but for an interactive terminal session, sending just a few bytes at a time, the overhead can be multiple times bigger than the payload itself.

Total overhead

To get a rough figure of how much overhead the whole ICE and HIP process creates, we can calculate the overhead of different phases together. Then, we can calculate how many packets need to be sent before the amount of sent payload exceeds the overhead.

The candidate gathering phase has to be performed only once as long as the candidates are kept alive. A single request message to a TURN server from each of the interfaces is enough for getting relayed and server reflexive candidates. In case of single homed host, this means only one message exchange, with 28 sent and 92 received bytes as noted in Section 4.4.3. The next phase, HIP base exchange, requires both of the hosts to send two messages that contain the ICE candidates. As noted before, if both are single homed hosts behind a NAT, this requires 864 bytes to be sent by the Initiator and 960 bytes by the Responder. The amount of messages and bytes exchanged during the connectivity check phase depends a lot on the types of the NATs, but 4–5 messages (584 – 728 bytes) sent by both endpoints seems to be normal in a scenarios without misbehaving NATs. With Linux NATs, on average up to 14 messages (2215 bytes) might be needed. Thus, setting up a single connection between two hosts requires roughly from $1 + 2 + 4 = 7$ to $1 + 2 + 14 = 17$ sent messages and from $28 + 864 + 584 = 1476$ to $28 + 960 + 2215 = 3203$ sent bytes.

When a path is created, every data packet has at least 18 bytes of overhead due to the ESP and UDP encapsulation. If we send on average 1kB data packets, before the overhead is less than 50%, we need to send 2–4 packets, mainly depending on the amount of messages needed for the connectivity checks. If we send on average 100 byte packets, it takes 18 packets with total 1800 bytes of payload to exceed the overhead of $1464 + 18 \times 18 = 1788$ bytes. Yet, if also some possible alternative path has the same per packet overhead, the ESP encapsulation should be disregarded and with 15 packets the amount of payload passes the amount of connectivity check overhead.

This observation suggests that if only a few small messages are exchanged, an alternative path, such as using the overlay, would make sense. On the other hand, already a couple of bigger packets may justify the HIP and ICE overhead.

MTU issues

Because the HIP base exchange packets are relatively large, they may exceed the MTU limit of links between the hosts if multiple locators are included in them. This results in fragmentation which can be problematic with NATs as discussed in Section 2.1.5. A typical MTU for IPv4 run over Ethernet is 1500 bytes [23] and for IPv6 the minimum MTU is 1280 bytes [12]. Before the IPv6 MTU is reached for the biggest base exchange packet, I2 with 680 bytes, there can be up to $\lfloor (1280 - 680 - 4 - 8 - 8 - 4)/36 \rfloor = 16$ locators[1]. The common IPv4 MTU is reached after $\lfloor (1500 - 680 - 4 - 8 - 8 - 4)/36 \rfloor = 22$ locators.

Having that many locators would require multiple interfaces: e.g., 6 interfaces, with each behind a different NAT, could lead to 18 locators (host, server reflexive and relayed candidate for each). Then, all of them would not fit into an I2 sent over IPv6 minimum MTU without fragmentation. However, even if a host had that many interfaces, probably all of them would not be need for ICE, but even a subset would be sufficient for finding a working path. Also, the ICE draft recommends that the maximum connectivity check list size should be limited to 100 entries [46], so even if both hosts send more than 10 candidates, only the 100 highest priority candidate pairs would eventually be tested, limiting the usefulness of sending large amount of candidates.

Then again, if the base exchange is done over a P2P overlay, the overlay may set a lower limit for the packet size than what the IPv4 and IPv6 MTU allow. In this case, the number of locators sent by the hosts may need to be limited.

5.2 Future Work

Already in the measurement analysis part (Section 4.5) we mentioned a few possible improvements to ICE that were related to the observations on the connectivity check process. During the prototype implementation we also found out other ways to improve the performance of ICE in ways that are not found in the specification. The following sections discuss these improvement ideas. We have not implemented these ideas, but they are left for future work. Also, we present some ideas for making a more thorough analysis of ICE.

5.2.1 Enhancing ICE

ICE, and the hosts using it, could benefit from a self tuning RTO calculation that selects the value for the RTO based on the network characteristics, from a TURN server telling which networks are accessible through it, and from ability to use application layer hints as an input for the stopping criteria. All these ideas for enhancing ICE are shortly presented below.

[1] Taking into account the NAT transformation parameter, UDP encapsulation, and non-ESP marker

Self tuning retransmission timeout

Because of the low packet loss of our prototyping network, none of the tested scenarios included connectivity check packets that were dropped for some other reason than a missing NAT mapping or filtering rule. Hence, the retransmission timers were not tested in cases where another try would have resulted in a succeeded check.

Still, we can observe that the default value for ICE RTO timer, presented in Section 2.4.2, is relatively high. The high value prevents premature retransmits in networks with long RTT links, but in a high-speed network even a single dropped packet can cause substantial delay in completing the connectivity checks or may lead to un-optimal path selection. For example, with 6 candidate pairs and the standard 500ms RTO, the very first retransmit happens at 3 seconds after start of the connectivity checks — one second after the soft deadline used in our implementation.

To cope with this issue, we should use different value of RTO for candidates with different network characteristics. However, the characteristics of the links are not normally known to ICE. As a remedy to this problem, some hint of the candidate RTT can be discovered at the candidate gathering phase. If the STUN or TURN server used to discover server reflexive addresses is connected using the same network that will forward the traffic between the peers using that candidate, the time it takes for the response from the STUN/TURN server to arrive can be used as an estimate of this peer's part of total RTT. Both peers could then communicate, during the exchange of the candidates, also the RTT measurements and sum of both measurements could be used as the value for the initial RTO.

This enhancement would likely result in shorter than 500ms RTO with many high speed access network technologies, but would not still use unrealistically small RTO values for candidates that are from a network interface with long delays.

Private range destination addresses with TURN server

When all local and remote candidates are paired for the connectivity checks, a relayed address from the TURN server is also paired with all remote host addresses — even if they are from private address ranges. Since the TURN server is likely in the public Internet and has no interface to the private subnets, none of these pairs with private range destination will result in a working path. Still, they are tested and result in unnecessary increase in traffic. What makes the problem even worse is the fact that the host candidates have by default higher priority than the server reflexive candidate(s) (which would more likely work), so all pairs that are less likely to work are tested before the ones that are more likely to work using the high cost relayed path.

There is a possibility that the TURN server has an interface to the private network too, for example, if the TURN server is run on the same box that works as the NAT. In this case it makes sense to try also the private range addresses. Unfortunately, there is no way for the TURN client to know that.

To remedy this problem, an additional STUN attribute could be added to the TURN allocation response which states to which private range networks, if any, the TURN server has a connection to. The TURN client (i.e., ICE endpoint) could then check if it makes sense to pair the relayed candidate with the private range host candidates. By making this new attribute *comprehension optional* type, it would be incrementally deployable and would interoperate also with implementations that do not support this feature.

Adjusting the stopping criteria

In Section 3.4.3 we discussed how the criteria of when to stop the connectivity checks is crucial for ICE's performance. The obvious way to implement it is to make the criteria same for all ICE sessions. However, application layer hints and information about the previous sessions could be used to adjust the criteria and this way decrease the time it takes for ICE to conclude the connectivity checks.

For example, if the application knows that it is a high priority goal to get a connection up quickly, but there is going to be only a small amount of narrow bandwidth traffic, using what ever path seems to work first would make sense. This would be essentially the same as using ICE with aggressive nomination but using an extra ICE check for the nomination. Also, since the relayed path is the most likely to work, it could be tried first to speed up the process; or at least server reflexive candidates could be prioritized over host candidates. Similarly, an application hint telling that large amount of latency sensitive data will be transferred over the selected path could be used to adjust the algorithm to try more persistently if the more optimal paths might work.

Information gathered from the previous sessions could also be used for determining the ordering of the connectivity checks. For example, if all the previous ICE sessions with different peers ended up using a relay, the host is likely behind an address and port-dependent NAT which allows direct connectivity only in rare cases. In this case, the relayed path could be accepted earlier instead of trying the more optimal paths multiple times.

5.2.2 Further Evaluation of ICE

Even though we tested our implementation with 27 different NAT scenarios, we have barely scratched the surface of all the possibilities with different NAT settings. One topic for future work would thus be evaluating ICE in more complicated scenarios and with different types

of NATs than what we did in this work. ICE should be tested with NATs that have different mapping behaviors and also in networks with different NAT topologies, such as the one discussed in Section 4.1.2.

In addition, we should address the issues about the measurement result generality noted in Section 4.5.6. We could test higher RTT networks using nodes that are in different physical networks, or preferably even in different sites around the Internet. The effect of packet loss could be tested by artificially dropping some of the connectivity check packets and measuring how that affects the performance. The different stopping criteria are a bit harder thing to test, since there can be essentially infinite number of them. Yet, some different classes of criteria could be tested and compared, and likely a better criteria than what we used could be created. As noted multiple times before, this would probably lead to easiest performance increments for our current implementation.

Nevertheless, probably the best way to evaluate ICE is to get it deployed and in use in the Internet. Only experience from real use cases and real users will tell whether ICE really works efficiently enough — or if it even makes a difference whether ICE ends up sending, say, 4 or 6 messages during the connectivity check phase. Hopefully, also our ICE implementation can, and will be used as a tool for getting more of this experience.

5.3 Summary

HIP benefits P2PSIP with NAT traversal, and also with other features, but P2PSIP nodes can as well in turn help HIP by providing relaying functionality using the overlay nodes. If a relay is needed for the data traffic, HIP could additionally be used for replicating the data on multiple paths and hence increase the reliability of connections.

Because HIP uses encapsulation of both the signaling and data traffic, it has a higher overhead than a plain ICE solution. Thus, if only a few messages will be exchanged using a connection, the setup overhead can easily be higher than the payload that is transferred. Also, link MTU sets limits for maximum amount of ICE candidates that should be exchanged during the base exchange, but the limit is so high that it is not likely to cause any problems.

After analyzing the overhead, we presented three different ICE enhancement ideas left for future work: a self tuning RTO algorithm, a private address range aware TURN server, and using application layer hints for better ICE connectivity checks. In addition, one direction for future work would be to evaluate ICE more thoroughly, and also do it by deploying ICE to real applications in the Internet.

Chapter 6

Conclusions

Network Address Translators cause a lot of problems for different network protocols and especially for peer-to-peer communication. Numerous different kinds of NATs are deployed in the Internet, and depending on the NAT behavior, different NAT traversal solutions may, or may not, be able to create a peer-to-peer connection between two hosts.

Interactive Connectivity Establishment is a robust, but also fairly complex NAT traversal solution. To evaluate how well ICE works, we implemented an ICE library prototype and tested it with various different NAT scenarios. Our implementation was able to create a working path in all the different scenarios, but many of them resulted in using more messages and longer time than expected or necessary. These scenarios were thoroughly analyzed and we derived conditions on when too many messages are exchanged and presented solutions on how some of these redundant message exchanges could be avoided.

In addition to finding out whether a path can be created, we presented data on how much traffic was needed and how long time it took to create a working path in different scenarios. It appears that 4–5 connectivity check messages and less than a kilobyte of traffic is enough in many cases. However, NATs with endpoint-dependent mapping behavior can easily double the amount of messages needed for finding the optimal path. From the measurement data we also derived bitrate estimations for the connectivity check traffic and concluded that running the checks even on a narrow bandwidth access technology should not be a big problem.

The check duration results revealed that, with the timer values suggested by the ICE specification, the connectivity checks commonly take multiple seconds to finish. This was fixed by using much shorter timer values and the tests confirmed that a path was created this way much faster, without much increase in the amount of messages. We also presented reasoning why the shorter timer values should not cause much problems even in networks with higher latency.

Since ICE has a higher success probability than the simple UDP hole punching with different types of NATs, it makes sense to use ICE instead of just hole punching as a NAT traversal solution. Successful NAT traversal with various NAT types is especially important for P2P environments where both endpoints of a connection are likely to be behind NATs, or even multiple layers of them.

By integrating ICE to Host Identity Protocol, a generic NAT traversal solution, that also P2PSIP can easily benefit from, can be created. The P2PSIP overlay network can help HIP by providing a relaying service and HIP can, in addition to NAT traversal, help P2PSIP with security, mobility, multihoming and unreliable paths. We analyzed the cost of using HIP with ICE for creating connections and found out that with only a few small messages, the overhead can become substantial. However, already a couple of bigger messages are often enough to justify the overhead traffic.

Even though NAT traversal seemed to be a simple research topic at the first glance, we learned that the truth is quite different. As we noted from the experiences with the Linux NATs, legacy NATs can vary a lot on their behavior, and even classifying them may not always be straightforward. Running a distributed NAT traversal algorithm, such as ICE, certainly does not make things simpler and resulted in unexpected results on many scenarios. Thus, we conclude that *The Twelve Networking Truths* [7] hold also for NAT traversal: it is more complicated than you think.

Bibliography

[1] B. Aboba and W. Dixon. IPsec-Network Address Translation (NAT) Compatibility Requirements. RFC 3715 (Informational), March 2004.

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

[3] S. Baset and H. Schulzrinne. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–11, April 2006.

[4] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *In Proceedings of ACM SIGCOMM ASIA Workshop*. ACM, 2005.

[5] BitLord Frequently asked questions. `http://www.bitlord.com/faq.php`. Referenced on 22.7.2008.

[6] D. Bryan, P. Matthews, E. Shim, D. Willis, and S. Dawkins. Concepts and Terminology for Peer to Peer SIP. draft-ietf-p2psip-concepts-01.txt (work in progress), July 2008. Expires: Jan 8, 2009.

[7] R. Callon. The Twelve Networking Truths. RFC 1925 (Informational), April 1996.

[8] G. Camarillo, P. Nikander, and J. Hautakorpi. HIP BONE: Host Identity Protocol (HIP) Based Overlay Networking Environment. draft-camarillo-hip-bone-01 (work in progress), February 2008. Expires: August 19, 2008.

[9] B. Carpenter and K. Moore. Connection of IPv6 Domains via IPv4 Clouds. RFC 3056 (Proposed Standard), February 2001.

[10] L. Daigle and IAB. IAB Considerations for UNilateral Self-Address Fixing (UNSAF) Across Network Address Translation. RFC 3424 (Informational), November 2002.

[11] G. Van de Velde, T. Hain, R. Droms, B. Carpenter, and E. Klein. Local Network Protection for IPv6. RFC 4864 (Informational), May 2007.

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

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

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

[15] B. Ford, D. Kegel, and P. Srisuresh. Peer-to-Peer Communication Across Network Address Translators. In *Proceedings of the 2005 USENIX Technical Conference*, 2005.

[16] S. Guha, K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT Behavioral Requirements for TCP. draft-ietf-behave-tcp-07.txt (work in progress), April 2007. Expires: October 30, 2007.

[17] S. Guha and P. Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *IMC '05: Proceedings of the 5th ACM SIGCOMM conference on Internet measurement*, pages 1–13, New York, NY, USA, 2005. ACM.

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

[19] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), July 2006.

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

[21] C. Huitema. Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs). RFC 4380 (Proposed Standard), February 2006.

[22] A. Huttunen, B. Swander, V. Volpe, L. DiBurro, and M. Stenberg. UDP Encapsulation of IPsec ESP Packets. RFC 3948 (Proposed Standard), January 2005.

[23] IEEE. Part 3: Carrier sense multiple access with collision detect on (CSMA/CD) access method and physical layer specifications. *IEEE Std 802.3, 2000 Edition*, pages i–1515, 2000.

[24] International Telecommunication Union. Network Grade of Service Parameters and Target Values for Circuit-Switched Services in the Evolving ISDN. Recommendation E.721, Telecommunication Standardization Sector of ITU, Geneva, Switzerland, May 1999.

[25] The netfilter.org "iptables" project. `http://www.netfilter.org/projects/iptables/`. Referenced on 21.8.2008.

[26] P. Jokela, R. Moskowitz, and P. Nikander. Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP). RFC 5202 (Experimental), April 2008.

[27] P. Karn, P. Metzger, and W. Simpson. The ESP Triple DES Transform. RFC 1851 (Experimental), September 1995.

[28] M. Komu, T. Henderson, P. Matthews, H. Tschofenig, and A. Keranen. Basic HIP Extensions for Traversal of Network Address Translators. draft-ietf-hip-nat-traversal-04.txt (work in progress), July 2008. Expires: Jan 16, 2009.

[29] J. Laganier and L. Eggert. Host Identity Protocol (HIP) Rendezvous Extension. RFC 5204 (Experimental), April 2008.

[30] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS Protocol Version 5. RFC 1928 (Proposed Standard), March 1996.

[31] C. Madson and R. Glenn. The Use of HMAC-SHA-1-96 within ESP and AH. RFC 2404 (Proposed Standard), November 1998.

[32] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Informational), May 2006.

[33] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. draft-moskowitz-hip-08 (work in progress), October 2003. Expires: April 22, 2004.

[34] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. RFC 5201 (Experimental), April 2008.

[35] P. Nikander, T. Henderson, C. Vogt, and J. Arkko. End-Host Mobility and Multihoming with the Host Identity Protocol. RFC 5206 (Experimental), April 2008.

[36] P. Nikander and J. Laganier. Host Identity Protocol (HIP) Domain Name System (DNS) Extensions. RFC 5205 (Experimental), April 2008.

[37] P. Nikander and J. Melen. A Bound End-to-End Tunnel (BEET) mode for ESP. draft-nikander-esp-beet-mode-09 (work in progress), August 2008. Expires: Feb 6, 2009.

[38] P. Nikander, J. Ylitalo, and J. Wall. Integrating Security, Mobility and Multi-Homing in a HIP Way. In *Proceedings of NDSS Symposium 2003*. Internet Society, Feb 2003.

[39] H. Olofsson and A. Furuskar. Aspects of introducing EDGE in existing GSM networks. *Universal Personal Communications, 1998. ICUPC '98. IEEE 1998 International Conference on*, 1:421–426 vol.1, Oct 1998.

[40] S. Pierrel, P. Jokela, J. Melen, and K. Slavov. A Policy system for Simultaneous Multi-Access with the Host Identity Protocol. In *Proceedings of the 1st IEEE Workshop on Autonomic Communications and Network Management (ACNM 2007)*, May 2007.

[41] PJNATH - Open Source ICE, STUN, and TURN Library. `http://www.pjsip.org/pjnath/docs/html/`. Referenced on 21.8.2008.

[42] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[43] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (Standard), October 1985. Updated by RFCs 2228, 2640, 2773, 3659.

[44] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[45] J. Rosenberg. Interactive Connectivity Establishment. *IETF Journal*, 2(3), November 2006.

[46] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. draft-ietf-mmusic-ice-19 (work in progress), October 2007. Expires: May 1, 2008.

[47] J. Rosenberg. Guidelines for Usage of Interactive Connectivity Establishment (ICE) by non Session Initiation Protocol (SIP) Protocols. draft-rosenberg-mmusic-ice-nonsip-01 (work in progress), July 2008. Expires: January 15, 2009.

[48] J. Rosenberg, R. Mahy, and P. Matthews. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). draft-ietf-behave-turn-09 (work in progress), July 2008. Expires: January 13, 2009.

[49] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for (NAT) (STUN). draft-ietf-behave-rfc3489bis-13 (work in progress), November 2007. Expires: May 20, 2008.

[50] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002.

[51] 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.

[52] P. Saint-Andre. Jingle: Jabber Does Multimedia. *IEEE MultiMedia*, 14(1):90–94, 2007.

[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[54] L. Silvennoinen. Legacy Network Address Translator Traversal Using the Host Identity Protocol. Master's thesis, Helsinki University of Technology, Department of Electrical and Communications Engineering, Oct 2007.

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

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

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

[58] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework. RFC 3303 (Informational), August 2002.

[59] M. Stiemerling and J. Quittek. Problem Statement: HIP operation over Network Address Translators. draft-stiemerling-hip-nat-00 (work in progress), February 2004. Expires: August 2004.

[60] M. Stiemerling, J. Quittek, and L. Eggert. NAT and Firewall Traversal Issues of Host Identity Protocol (HIP) Communication. RFC 5207 (Informational), April 2008.

[61] TCPDUMP/LIBPCAP public repository. `http://www.tcpdump.org/`. Referenced on 27.8.2008.

[62] H. Tschofenig and D. Wing. Utilizing Interactive Connectivity Establishment (ICE) for the Host Identity Protocol (HIP). draft-tschofenig-hip-ice-00 (work in progress), June 2007. Expires: December 23, 2007.

[63] UPnP Standards, Internet Gateway Device (IGD) V 1.0. `http://www.upnp.org/standardizeddcps/igd.asp`. Referenced on 20.7.2008.

[64] J. Ylitalo, J. Melén, P. Nikander, and V. Torvinen. Re-Thinking security in IP based micro-mobility. In *In Proc. of 7th Information Security Conference (ISC04*, 2004.

Appendix A

Linux NAT configuration

The Linux boxes were configured to works as a NAT device using the following script:

```
#! /bin/bash
set -e
PATH="/sbin:/usr/sbin:/bin:/usr/bin:${PATH}"
IPTABLES="/sbin/iptables"
case "$1" in
    start)
        # accept by default
        $IPTABLES -P OUTPUT  ACCEPT
        $IPTABLES -P INPUT   ACCEPT
        $IPTABLES -P FORWARD ACCEPT

        # flush all chains
        cat /proc/net/ip_tables_names | while read table; do
        test "X$table" = "Xmangle" && continue
          $IPTABLES -t $table -L -n | while read c chain rest; do
              if test "X$c" = "XChain" ; then
                $IPTABLES -t $table -F $chain
              fi
          done
          $IPTABLES -t $table -X
        done

        # enable IP forwarding
        echo 1 > /proc/sys/net/ipv4/ip_forward

        # set up NATing
        $IPTABLES -t nat -A POSTROUTING -o eth0 -j MASQUERADE
        $IPTABLES -t nat -A POSTROUTING -o eth1 -j MASQUERADE
        $IPTABLES -t nat -A POSTROUTING -o eth2 -j MASQUERADE
        $IPTABLES -t nat -A POSTROUTING -o eth3 -j MASQUERADE
    ;;
  *)
    echo "Usage: /etc/init.d/proto-nat {start}"
    exit 1
    ;;
esac

exit 0
```

Appendix B

Linux NAT behavior

The following tcpdump capture demonstrates how Linux NAT changes an allocated mapping if there is an incoming packet from an unknown address.

```
# STUN Binding Request through the NAT. NAT allocates the port 51012.
13:50:02.398085 IP 10.1.0.185.51012 > 193.234.218.188.3478: UDP, length 20
13:50:02.398181 IP 193.234.218.189.51012 > 193.234.218.188.3478: UDP, length 20

# Response from the STUN server to the Binding Request is accepted.
13:50:02.398846 IP 193.234.218.188.3478 > 193.234.218.189.51012: UDP, length 44
13:50:02.399277 IP 193.234.218.188.3478 > 10.1.0.185.51012: UDP, length 44

# Connectivity checks start with host-host candidate pair.
13:50:02.438287 IP 10.1.0.185.51012 > 10.3.0.187.51002: UDP, length 144
13:50:02.438492 IP 193.234.218.189.51012 > 10.3.0.187.51002: UDP, length 144

# Peer's check to host's server reflexive address arrives.
13:50:02.926808 IP 193.234.218.183.51002 > 193.234.218.189.51012: UDP, length 144
# Linux NAT's filtering behavior blocks this and the NAT replies with ICMP "port unreachable".
13:50:02.926924 IP 193.234.218.189 > 193.234.218.183: ICMP 193.234.218.189 udp port 51012 unreachable

# Host start a check towards the peer's server reflexive address.
13:50:02.940746 IP 10.1.0.185.51012 > 193.234.218.183.51002: UDP, length 144

# Now, the NAT uses source port 59776 instead of the original 51012!
13:50:02.940879 IP 193.234.218.189.59776 > 193.234.218.183.51002: UDP, length 144

# Peer's Linux NAT replies with ICMP since there is no binding for the new source port.
13:50:02.941199 IP 193.234.218.183 > 193.234.218.189: ICMP 193.234.218.183 udp port 51002 unreachable
13:50:02.941300 IP 193.234.218.183 > 10.1.0.185: ICMP 193.234.218.183 udp port 51002 unreachable

[... relayed candidates get tested and succeed ... ]

# Host starts retransmits for the failed candidates .
13:50:05.466382 IP 10.1.0.185.51012 > 10.3.0.187.51002: UDP, length 144
# A check to the host candidate still uses the original port 51012.
13:50:05.466464 IP 193.234.218.189.51012 > 10.3.0.187.51002: UDP, length 144

13:50:05.964673 IP 10.1.0.185.51012 > 193.234.218.183.51002: UDP, length 144
# However, the check to the server reflexive address uses the new port, 59776.
13:50:05.964762 IP 193.234.218.189.59776 > 193.234.218.183.51002: UDP, length 144

# And the check for the server reflexive candidate fails again.
13:50:05.966178 IP 193.234.218.183 > 193.234.218.189: ICMP 193.234.218.183 udp port 51002 unreachable
13:50:05.966218 IP 193.234.218.183 > 10.1.0.185: ICMP 193.234.218.183 udp port 51002 unreachable
```

Appendix C

Results of the Quick Mode

The following graphs present the results of the connectivity checks using the quick mode described in Section 4.4.

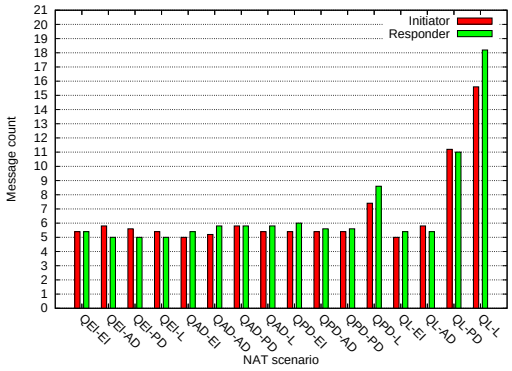

Figure C.1: Average sent messages in the quick mode

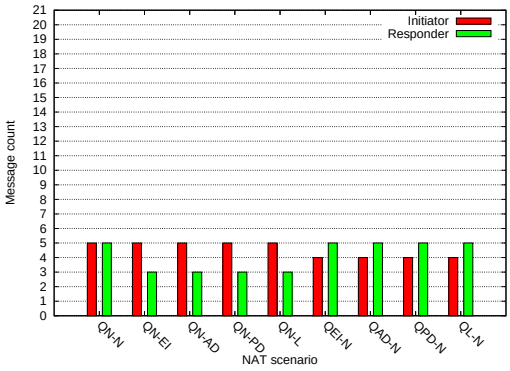

Figure C.2: Number of messages sent in the quick mode in non-NATed scenarios

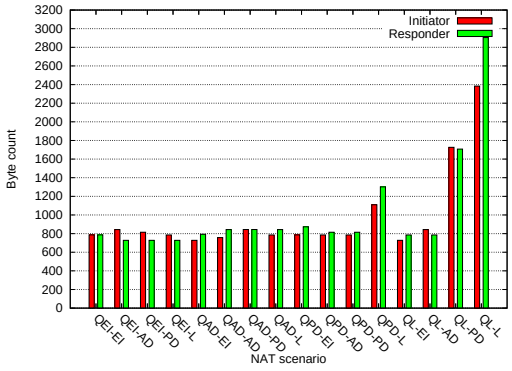

Figure C.3: Average sent bytes in the quick mode

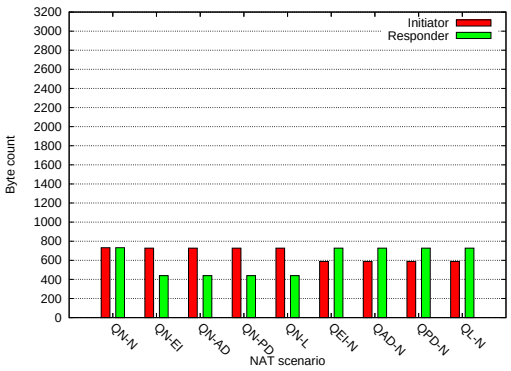

Figure C.4: Sent bytes in the quick mode in non-NATed scenarios

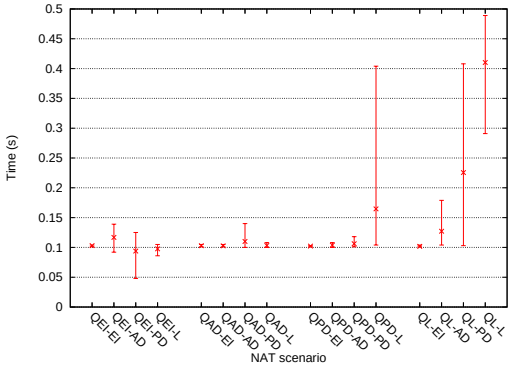

Figure C.5: First successful test's time in the quick mode

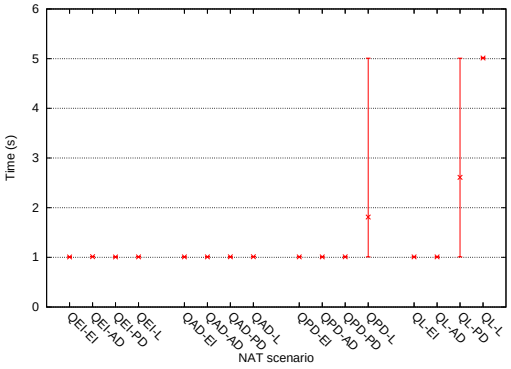

Figure C.6: Time when all checks are done in the quick mode

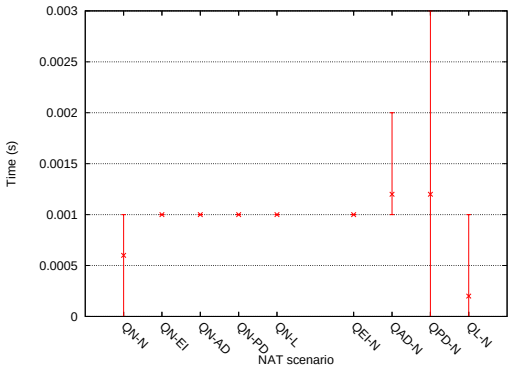

Figure C.7: First successful test's time in the quick mode in non-NATed scenarios

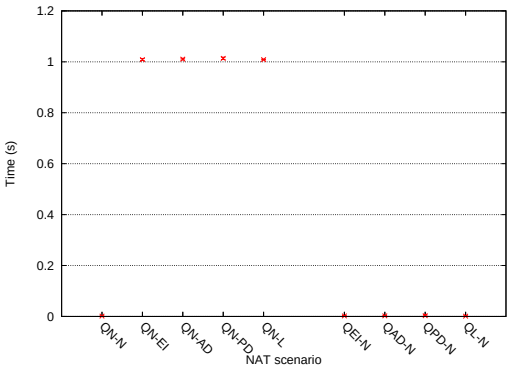

Figure C.8: Time when checks are done in the quick mode in non-NATed scenarios

Appendix D

Capture of a HIP Base Exchange

Following shows a tcpdump capture of a HIP base exchange (I1, R1, I2 and R2) between two hosts. The Initiator is at address 193.234.219.77 and the Responder at address 193.234.218.203. The D-H group ID was 3 (1536-bit MODP group) and the length of the public key was 1024 bits (128 bytes). The base exchange was not UDP encapsulated, but run directly on top of IPv4.

```
17:14:28.328503 IP 193.234.219.77 > 193.234.218.203:  ip-proto-139 40
17:14:28.421322 IP 193.234.218.203 > 193.234.219.77:  ip-proto-139 600
17:14:28.645849 IP 193.234.219.77 > 193.234.218.203:  ip-proto-139 680
17:14:28.886289 IP 193.234.218.203 > 193.234.219.77:  ip-proto-139 216
```