Legacy Network Address Translator Traversal Using the Host Identity Protocol

Lauri Silvennoinen

Helsinki University of Technology
Department of Electrical and Communications Engineering
Networking Laboratory

HELSINKI UNIVERSITY OF TECHNOLOGY		ABSTRACT OF THE MASTER'S THESIS
Author:	Lauri Silvennoinen	
Name of the thesis:	Legacy Network Address Translator Traversal Using the Host Identity Protocol	
Date:	October 30th 2007	**Number of pages:** 74
Department:	Department of Electrical and Communications Engineering	
Professorship:	S-38	
Supervisor:	Dr.Ing. Jörg Ott	
Instructor:	M.Sc. Miika Komu	

Network Address Translators (NATs) are used in the Internet to map Internet Protocol (IP) addresses from one realm to another. Currently deployed NAT devices can translate Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) packets, but other protocols are not supported. For example, Host Identity Protocol (HIP) based communication does not work with most of the legacy NATs which just drop HIP traffic. The goal of this thesis was to evaluate NAT traversal extensions for HIP by implementation.

We implemented extensions that enable HIP based communication through most types of legacy NATs while maintaining the benefits of the protocol. The design is based on encapsulating both the control and data traffic in User Datagram Protocol (UDP). Furthermore, to be reachable behind a NAT, we need a rendezvous point that lets HIP hosts behind a NAT to register an IP address and port number that can be used to contact them. Accordingly, we discuss the operation and design of a Rendezvous Server (RVS) and registration extension herein.

This work brought up some future work items. We need a more fine-grained division between different kinds of NATs for HIP. We need to improve both the registration extension and the functionality of the RVS to better meet the expectations set up by various scenarios involving NATs.

Keywords:	HIP, NAT, host, identity, middlebox

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Lauri Silvennoinen		
Työn nimi:	Perinteisten osoitteenmuuntajien läpäisy käyttäen koneen identiteetti protokollaa		
Päivämäärä:	30. lokakuuta 2007	**Sivuja:**	74
Osasto:	Sähkö- ja tietoliikennetekniikan osasto		
Professuuri:	S-38		
Työn valvoja:	TkT Jörg Ott		
Työn ohjaaja:	DI Miika Komu		

Osoitteenmuuntajia käytetään Internetissä eri alueiden välisten verkko-osoitteiden muunnoksessa. Koneen identiteettiprotokollaan (HIP) perustuva viestintä ei toimi useimpien perinteisten osoitteenmuuntajien kanssa, sillä nämä eivät laske HIP-liikennettä läpi. Tämän työn tavoitteena oli toteuttaa laajennus, joka mahdollistaa HIP-kommunikoinnin osoitteenmuuntajien läpi.

Työn tuloksena valmistunut laajennus mahdollistaa HIP-kommunikoinnin useimpien perinteisten osoitteenmuuntajien läpi säilyttäen samalla protokollan edut. Mekanismi perustuu sekä hyötykuorma- että kontrolliliikenteen paketoimiseen User Datagram Protokollaan (UDP). Jotta verkkopääte olisi tavoitettavissa osoitteenmuuntajan takaata, tarvitaan yleisesti tunnettu kohtaamispiste, joka mahdollistaa osoitteenmuuntajan takana olevan verkkopäätteen Internet Protocol-osoitteen ja porttinumeron rekisteröinnin. Tästä syystä tässä työssä käsitellään myös kohtaamispiste palvelimen (RVS) toimintaa.

Laajennuksen kehittäminen synnytti muutamia jatkokehitysideoita. Host Identity Protokollaa varten tarvitaan hienojakoisempi jaottelu eri tyyppisten osoitteenmuuntajien välille. Sekä rekisteröintilaajennuksen että kohtaamispiste palvelimen toimintaa täytyy parantaa, jotta voidaan paremmin vastata erityyppisten osoitteenmuuntajien asettamiin haasteisiin.

Avainsanat:	HIP, NAT, osoitteenmuunnos, osoitteenmuuntaja

Acknowledgements

Naturally, my first thanks go to professor Jörg Ott whose guidance has made the process of completing this thesis an interesting and rewarding journey. He has given many insightful comments and corrections both to structure and to the grammar of this thesis. He has also patiently allowed me to take my time in finishing this work.

M.Sc. Janne Lindqvist deserves thanks for allowing me to have the opportunity to work in the InfraHIP project in the first place. The financial support for the project has come from TEKES, Nokia, Ericsson, Elisa and The Finnish Defence Forces. They have provided the daily income which is humbly acknowledged.

M.Sc. Miika Komu is the single most influential person who has participated in and contributed to this work. His insights and ideas on networking and protocol design, knowledge of details and capability to apply a wider perspective form the cornerstone of this work. I have had the privilege to work under his guidance during the course of this work. Thank you.

I wish to thank Telecommunications Software and Multimedia Laboratory staff for providing me the facilities in the laboratory. I also wish to express my sincere appreciation to Helsinki University of Technology for providing the facilities during all these years of studying. Maarintalo especially has come into its own.

Obviously both my mother and father and all of my kin have earned my gratitude for their underlying support.

I shall not forget to thank all my fellow students here at Otaniemi. Their influence and presence have helped me to keep things in perspective and made it possible for me to mentally survive. My special thanks are due to Jari Peltonen for all the fruitful discussion sessions during the lunch and coffee breaks — it looks like one can survive the elements at Otaniemensilta after all.

Espoo, October 30th 2007.

Lauri Silvennoinen

Contents

Acknowledgements **i**

List of Figures **v**

Abbreviations **vii**

Introduction **1**

1 Background **3**
1.1 Network Address Translation 3
1.1.1 The End-to-End Principle 4
1.1.2 Types of NATs 4
1.1.3 Basic NAT 5
1.1.4 NAPT 7
1.1.5 Endpoint Mapping 8
1.1.6 Endpoint Independent Mapping 8
1.1.7 Endpoint Dependent Mapping 8
1.1.8 Endpoint Filtering 10
1.1.9 UDP Hole Punching 10
1.2 HIP 13
1.2.1 HIP Architecture 13
1.2.2 Base Exchange 14
1.2.3 Secured Data Exchange Over IPsec 16
1.2.4 HIP State Machine 18
1.2.5 HIP Packets 18
1.2.6 RVS 20
1.2.7 Complications with NATs 21
1.3 Chapter Summary 22

2 NAT Traversal Using the HIP **23**
2.1 NAT Detection 23
2.2 UDP Tunneling of HIP Traffic 25
2.3 NAT Extensions for the RVS 25
2.4 Scenarios Involving NATs 26
2.4.1 Initiator Behind a NAT 27
2.4.2 Responder Behind a NAT 28
2.4.3 Both Hosts Behind a NAT 29
2.4.4 Use of Rendezvous Service When Only I Is Behind a NAT 31
2.5 Chapter Summary 32

3 Implementation **33**
3.1 Implementation Overview 33
3.1.1 Software Organization 33
3.1.2 The RVS Implementation 34
3.1.3 The NAT Extension Implementation 37
3.2 UDP and Port Selection Logic 38
3.2.1 Responder Side Logic 38
3.2.2 Initiator Side Logic 38
3.2.3 Updating a HIP Association 40
3.3 Chapter Summary 40

4 Discussion **41**
4.1 Adding STUN Support to the Implementation 41
4.2 Further Analysis of the Both Hosts Behind NAT Case 42
4.2.1 Both NATs Employ Endpoint Dependent Mapping 43
4.2.2 Initiator Behind a NAT That Employs Endpoint Dependent Mapping 44
4.2.3 Responder Behind a NAT That Employs Endpoint Dependent Mapping 44
4.3 Improvements in the Next Development Cycle 45
4.3.1 Improvements in the NAT Keep-alives 45
4.3.2 Improvements in the Both Hosts Behind NAT Case . . . 46
4.4 STUN Relay Usage 46
4.5 Next Steps in NAT Traversal 48
4.6 Chapter Summary 49

Conclusions **51**

Bibliography **53**

A Behavioral Requirements for Future NATs **57**

List of Figures

1.1 Basic NAT operation 5
1.2 NAPT operation 7
1.3 NAT device employing Endpoint Independent Mapping 9
1.4 NAT device employing Endpoint Dependent Mapping 9
1.5 NAT device employing Endpoint Independent Filtering 10
1.6 Registration to server 11
1.7 UDP Hole Punching 12
1.8 The current IP stack and the HIP based stack 13
1.9 Base exchange 15
1.10 TCP/IPv6 packet and ESP transport mode packet 17
1.11 TCP/IPv6 packet and ESP tunnel mode packet 17
1.12 HIP state machine 19
1.13 HIP packet header 19
1.14 HIP base exchange via an RVS 21

2.1 Example STUN Configuration 24
2.2 UDP packet 25
2.3 UDP encapsulation of IPsec BEET-mode ESP packet 25
2.4 UDP encapsulated base exchange when I is behind NAT 27
2.5 UDP encapsulated base exchange when R is behind NAT 28
2.6 UDP encapsulated base exchange when I and R are behind NAT 30
2.7 UDP encaps. base ex. when I is behind NAT and R uses RVS . . 31

3.1 HIP implementation components 34
3.2 Logic diagram showing the role selection logic. 35
3.3 The rendezvous association data structure 36
3.4 The relay()-function 36
3.5 Logic diagram showing RVS functionality 37
3.6 Setting the NAT status on for a HIP association 37
3.7 Logic diagram showing UDP and port selection logic of R 39
3.8 Logic diagram showing UDP and port selection logic of I 39

4.1 Both HIP hosts behind NAT 43
4.2 STUN relay usage 48
4.3 ICE deployment scenario 49

A.1 Port assignment behavior 58
A.2 Hairpinning behavior 60

Abbreviations

AH Authentication Header

ALG Application Level Gateway

API Application Programming Interface

BEET Bound End-to-End Tunnel

DNS Domain Name Service

DoS Denial of Service

DSA Digital Signature Algorithm

ESP Encapsulating Security Payload

FQDN Fully Qualified Domain Name

FTP File Transfer Protocol

HI Host Identifier

HIP Host Identity Protocol

HIPL Host Identity Protocol for Linux

HIIT Helsinki Institute for Information Technology

HIT Host Identity Tag

HMAC Keyed-Hash Message Authentication Code

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

ICMP Internet Control Message Protocol

IETF Internet Engineering Task Force

I Initiator

ICE Interactive Connectivity Establishment

IP Internet Protocol

IPsec Internet Protocol security

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

ISP Internet Service Provider

LAN Local Area Network

LSI Local Scope Identifier

MAC Media Access Control

NAPT Network Address and Port Translator

NAT Network Address Translator

OpenDHT Open Distributed Hash Table

PID Process ID

PKI Public Key Infrastructure

QoS Quality of Service

R Responder

RFC Request for Comments

RSA Rivest, Shamir, & Adleman

RTP Real-time Transport Protocol

RTCP Real-time Transport Protocol Control Protocol

RVS Rendezvous Server

SA Security Association

SPI Security Parameter Index

STUN Simple Traversal of User Datagram Protocol through Network Address Translators

STUNT Simple Traversal of User Datagram Protocol through Network Address Translators and Transmission Control Protocol too

TCP Transmission Control Protocol

TKK Helsinki University of Technology

TLI Transport Layer Identifier

TLS Transport Layer Security

TLV Type-Length-Value

TURN Traversal Using Relay NAT

UDP User Datagram Protocol

UNSAF UNilateral Self-Address Fixing

Introduction

The current Internet has two important global namespaces: Internet Protocol (IP) addresses and Domain Name Service (DNS) names. These two namespaces have powered the Internet to what it is today. However, they also have a number of weaknesses. In particular, the IP addresses have a dual role in the current model. Firstly, IP addresses are used to locate a host and secondly, they provide routing information. These properties can be described as the identifier and the locator roles of the IP address respectively.

Unfortunately, the same IP address is also reused at the transport layer. As a drawback of this reuse, transport protocol connections, such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) connections, break when the underlying network layer IP addresses change. This limits the flexibility of the Internet architecture and hinders mobility and IP address renumbering.

The Host Identity Protocol (HIP) and the related architecture form a proposal to change the TCP/IP stack to securely support mobility and multihoming. The HIP architecture defines a new global cryptographic namespace, the Host Identity namespace. The new namespace decouples the identifier and the locator roles.

With HIP, the transport layer is based on Host Identities instead of using IP addresses as endpoint names. At the same time, the network layer uses IP addresses as pure locators. The benefit of this decoupling is that transport layer connections may persist between hosts while the network layer locators change.

Network Address Translators (NATs) are used in the Internet to map IP addresses from one realm to another. HIP based communication, however, cannot operate across most of the legacy NATs because these NATs can translate only TCP, UDP and Internet Control Message Protocol (ICMP) packets. The goal of this thesis is to evaluate NAT traversal extensions for HIP by implementation.

Our contribution has lead to a solution that enables HIP based communication through most types of legacy NATs while maintaining the benefits of the protocol. The solution is based on encapsulating both the control and data traffic in UDP. In this thesis, we discuss several different scenarios involving NAT devices and present the counter-measures to deal with the difficulties brought up by the NATs.

The scope of this thesis is limited to the implementation of the NAT traversal extension for HIP key exchange. Mobility is excluded. To lay ground for the discussion on the NAT traversal extension, we discuss the properties of the Host Identity Protocol and have a look on the most common types of NAT middleboxes. We do not extend our discussion to other types of middleboxes such as firewalls, proxy servers or anonymizers.

Chapter 1

Background

In this Chapter, we give an overview of the background topics fundamental to understanding of this thesis. The first section of this Chapter describes the most important features of Network Address Translator (NAT) devices, while the second section explains how HIP operates. We assume that the reader is familiar with the basic concepts of the TCP/IP suite and Internet routing.

1.1 Network Address Translation

The current Internet consists of a multitude of interconnected networks and subnetworks. Some of these networks are private while others are public. Public network Internet Protocol version 4 (IPv4) addresses are becoming more and more difficult to reserve. This address shortage limits the functionality of the Internet. During the past decade Network Address Translators have become popular as a way to deal with the IPv4 address shortage.

When two hosts establish a connection between each other, the traffic traverses many devices that may simply route and forward it, but may also modify it. Intermediary devices that do not modify the traffic are routers, while intermediary devices that modify the traffic in transit are designated as middleboxes[1]. Several types of middleboxes exist, the best known of which are firewalls and NATs. Firewalls mainly block or forward traffic according to a policy, while NAT behavior is more complicated. According to some measurements, up to 74% of all computers are located behind at least one NAT [7].

Network address translation is a function that dynamically assigns a globally unique address to a host that does not have one, without the knowledge of the host. Network address translation consists of modifying the source and destination addresses in the transport and IP headers, thereby allowing hosts in a private network to communicate with destinations on an external network and vice versa.

Architecturally Network Address Translators divide the Internet into independent administrative address domains. The network topology outside a private network can change in many ways. Whenever the external topology

[1] A middlebox is defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host. [3]

changes, address assignment for hosts within the private network must also change to reflect the external changes. Changes of this type can be hidden from users within the domain by centralizing changes to a single address translation router. Along with being a remedy to IPv4 address shortage, this capability of being able to hide global address changes is one of the main advantages of Network Address Translation. As a drawback, NATs break the end-to-end connectivity of the Internet as discussed next [5, 24].

1.1.1 The End-to-End Principle

The success of the Internet is based on few basic elements. Foremost is the end-to-end principle [5], which states that certain functions can only be performed in the endpoints. These endpoints are in control of the communication, and the network should be a simple datagram service that moves bits between the endpoints. The end-to-end principle argues that as much as possible, only the endpoints should hold critical state.

According to the principle, the endpoint applications are often the only place capable of correctly managing the data stream. Removing this concern from the internal network hosts streamlines the packet forwarding process and improves system wide efficiency. Another advantage of the end-to-end principle is that the endpoints need not to be aware of any network components other than the destination, first hop router and an optional name resolution service. Furthermore, the internal network hosts do not have to maintain per connection state information. This allows fast rerouting around broken links and failed network hosts [24].

Unfortunately NAT devices undermine most of the basic advantages of the end-to-end principle. NAT devices hold a critical state without which the communication becomes impossible. In other words, the connection fails if the NAT device becomes dysfunctional. Generally, NATs reduce the overall flexibility and often increase the operational complexity of the network [5].

In addition to breaking the end-to-end principle, NATs have several other disadvantages. NATs place constraints on the deployment of applications that carry IP addresses or transport identifiers[2] in the data stream. Also, they operate on the assumption that each transport layer session is independent of each others. However, applications such as File Transfer Protocol (FTP) use control sessions to characterize follow-on sessions. These kind of applications need Application Level Gateways (ALG) to aid in NAT traversal. ALGs add extra complexity to traffic management and they render end-to-end IP level security assured by Internet Protocol security (IPsec) impossible [24].

1.1.2 Types of Network Address Translators

Several types of NATs exist. We discuss two of the most common types of NAT middleboxes, the Basic Network Address Translator (Basic NAT) and Network Address and Port Translator (NAPT) in this Section. Of these, NAPT is by far the most commonly deployed NAT device. These two operations, Basic NAT and NAPT, are referred to as traditional NAT [26].

[2] A transport identifier is either a TCP/UDP port number or an Internet Control Message Protocol (ICMP) query ID.

Traditional NAT operation can be divided into three phases, as described in the following [26].

1. *Address binding* is the phase in which an IP address and optionally a transport identifier of a local host are associated with an external IP address and a transport identifier. The address is bound to an external address when the first outgoing session is initiated from a host behind a NAT.

2. Once a state is established for a session, all packets that belong to the session and use the same transport protocol (UDP, TCP), will be subject to *address lookup and translation*. Address lookup is the process of searching a NAT translation state using the IP address and the transport identifier of the incoming packet as a search key.

3. Finally, *address unbinding* is the phase in which a private address is no longer associated with a global address for purposes of translation. The related NAT state is torn down.

In a traditional NAT, session establishment is uni-directional from private address realm to public address realm. Sessions in the opposite direction may be allowed on an exceptional basis using static address maps for pre-selected hosts [24]. Static mapping means that a translation state that has been configured to the NAT device before any network traffic. The opposite of static mapping is dynamic mapping where translation state is established when the first packet belonging to a new session arrives.

1.1.3 Basic Network Address Translator

Basic NAT maps IP addresses from an address realm to another [24]. Typically, one of these realms has private addresses which a NAT device then translates to public addresses of an external realm. An external realm is a network with unique network addresses assigned by the Internet Assigned Numbers Authority (IANA), while a private realm has no such registration need and IP addresses can be assigned without coordination from IANA. Figure 1.1 illustrates Basic NAT operation with some example IP addresses.

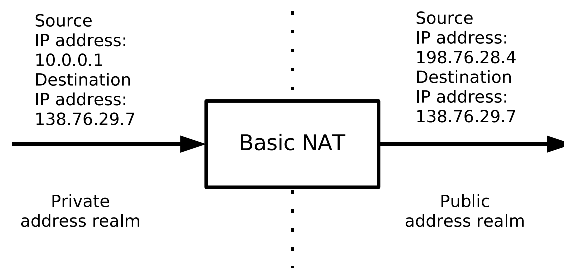

Figure 1.1: Basic Network Address Translator (NAT) operation

Here the private IP address (10.0.0.1) is the IP address of one of the hosts in the private realm, and public the IP address (198.76.28.4) is one of the addresses in the global set registered for the NAT device. Addresses in the private realm are local to that private address realm and are not valid outside the private address realm. Thus, addresses inside a private realm can be reused by any other private realm behind a different NAT. Private IP addresses are assigned by the private domain network administrator. Public IP address (198.76.28.4) is the address of the NAT interface to the external realm and it is assigned by the Internet Service Provider (ISP) (who in turn has registered the address from IANA).

When the number of local hosts is less than or equal to addresses in the global set, each local address is guaranteed a global address to map to. Otherwise, hosts allowed to have simultaneous access to the external network are limited by the number of addresses in the global set. In our example, the global set of addresses assigned to NAT might be for instance a class C address block (198.76.28.0/24).

When the private realm host (10.0.0.1) wishes to send a packet to some host in the global network whose IP address is, say, (138.76.29.7), it uses the globally unique address (138.76.29.7) as destination, and sends the packet to its primary NAT router. Before sending the packet to the next hop router en route to (138.76.29.7), the NAT router translates the source address (10.0.0.1) of the IP header to the globally unique (198.76.28.4) (or any other non-bound address from group 198.76.28.0/24).

As the host in the global network responds, it uses the IP address (198.76.28. 4) of the NAT device as destination. In fact, the global host believes that the address of the NAT device is actually the address of the final destination (host in the private realm). When the IP packets on the return path arrive to the NAT, the destination IP address (198.76.28.4) is translated to private IP address (10.0.0.1).

With Basic NATs, the private address is bound to an external address when the first outgoing session is initiated from private host. After that, all subsequent outgoing sessions originating from the same private address will use the same address binding for packet translation. When the last session based on an address binding is terminated, the binding itself will be terminated [24].

This raises the questions of how long does a session last and when can the address binding be torn down? Srisuresh and Holdrege have discussed this issue in RFC 2663 [26]. They state, that in the general case, it is not possible to distinguish between connections that have been idle for an extended period of time from those that no longer exist. Therefore, garbage collection is necessary on NAT devices to clean up unused state about TCP sessions that no longer exist. Srisuresh and Holdrege suggest garbage collection interval of 24 hours for TCP sessions and a couple of minutes for non-TCP sessions.

For TCP connections terminating normally, Srisuresh and Holdrege state that a session can be assumed to have been terminated after a period of four minutes subsequent to the detection of FIN[3] segment or a segment with the RST[4] bit set.

[3] TCP control bit: no more data from sender.
[4] TCP control bit: reset the connection.

1.1.4 Network Address and Port Translator

The basic idea behind a NAPT device is similar to Basic NAT; private realm addresses that are not valid globally are translated to globally routable external realm addresses. This allows communication between the private network and the public network. The fundamental difference between Basic NAT and NAPT is that translation in Basic NAT is limited to IP addresses alone, whereas translation in NAPT is extended to include the IP address and the transport identifier [24]. Transport identifier is either a TCP/UDP port number or an Internet Control Message Protocol (ICMP) query ID.

In the example of Figure 1.2, we have a private realm which uses internally a class A address block (10.0.0.0/8). A public IP address (198.76.28.4) has been assigned to the external interface of the NAPT. Note, that now there is only a single public source IP address instead of a set of IP addresses. In this example, we use a telnet connection with source port 3017 and destination port 23.

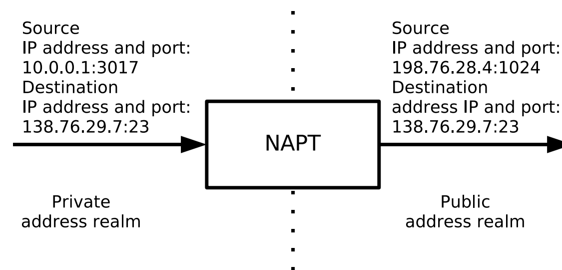

Figure 1.2: Network Address and Port Translator (NAPT) operation

When a host (10.0.0.1), located in the private realm, sends data to host (138.76.29.7), it uses the globally unique address (138.76.29.7) as destination, and sends the data through its NAPT. NAPT translates the source address (10.0.0.1) and the source TCP port (3017) in the IP and TCP headers into a globally unique IP address (198.76.28.4) and a uniquely assigned TCP port (1024). Packets on the return path go through similar address and port translation for destination IP address and destination TCP port. The TCP checksum is also rewritten.

With NAPT, the endpoint is bound to an external tuple of <*Assigned IP address, Assigned transport identifier*> when the first outgoing session is initiated from that endpoint. After that, all outgoing transport layer sessions originating from the same endpoint and using the same transport protocol will use the same translation binding while the translation state is valid. The destination port remains the same.

NAPT requires a transport identier to work. Therefore, sessions other than TCP, UDP and ICMP query type are simply not permitted from local hosts. Sessions initiating from the external realm (including TCP, UDP and ICMP query sessions) terminate to the NAT middlebox, unless the destination port is manually configured to a host in the private realm [24]. An example of this kind of manually configured port is a Hypertext Transfer Protocol (HTTP) server in

a corporation Local Area Network (LAN) listening for incoming connections on destination port 80.

1.1.5 Endpoint Mapping

Rosenberg et al. [22] have classified NAT implementations using the terms *Full Cone*, *Restricted Cone*, *Port Restricted Cone* and *Symmetric*. However, Srisuresh, Ford and Kegel [25] state that this terminology has been the source of much confusion. Therefore, this terminology has been removed e.g. from Interactive Connectivity Establishment (ICE) [18]. We will adapt NAT terminology from Audet and Jennings [2] for this reason. This terminology is based on the concept of *an endpoint*.[5] An endpoint is a session specific pair *<IP address, transport identifier>* on an end host.

When a host in the private realm wishes to send data to some host in the public realm through a NAT device, the NAT device assigns an external endpoint to translate the private endpoint to. Subsequent response packets from the external realm use this external endpoint to reach the host in the private realm. The translation of a private endpoint to a public endpoint and vice versa in a NAT device is called *Endpoint Mapping*.

1.1.6 Endpoint Independent Mapping

In Figure 1.3, we have a client (10.0.0.1) in the private realm, a public NAT interface (159.99.25.11) and two public servers, server S1 (18.181.0.31:1235) and server S2 (138.76.29.7:1235). The client is initiating two simultaneous outgoing transport layer sessions from the same endpoint (10.0.0.1:1234) to these two servers, S1 and S2. The NAT middlebox in Figure 1.3 performs Endpoint Independent Mapping which means that it reuses the Endpoint Mapping assigned to a private host endpoint for all sessions initiated by the private host from the same endpoint while the Endpoint Mapping is alive [25]. Now, the NAT assigns the same public endpoint of the NAT (159.99.25.11:62000) to both of the sessions although the destinations of the sessions are different. Guha and Francis [4] estimate that a majority (70.1%) of NATs deployed employ Endpoint Independent Mapping.

1.1.7 Endpoint Dependent Mapping

In Figure 1.4 we have exactly the same setting as in Figure 1.3 except that now the NAT device performs Endpoint Dependent Mapping. A NAT device that performs Endpoint Dependent Mapping does not reuse the same Endpoint Mapping for all sessions initiated by the private host from the same endpoint. The NAT device may assign a new Endpoint Mapping to each new session traversing the NAT device [25]. Now, we have two different public endpoints, one (155.99.25.11:62000) with the server S1 and another (155.99.25.11:62001) with the server S2.

[5] One can argue whether the term endpoint causes less confusion.

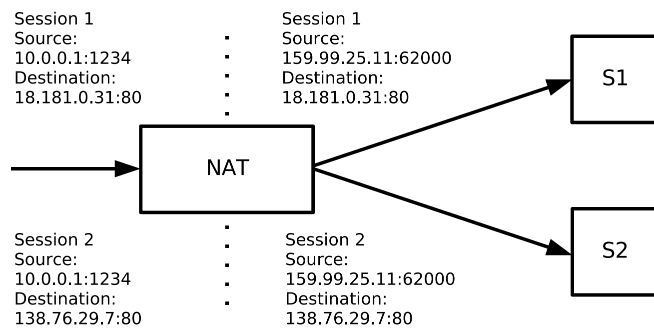

Figure 1.3: NAT device employing Endpoint Independent Mapping

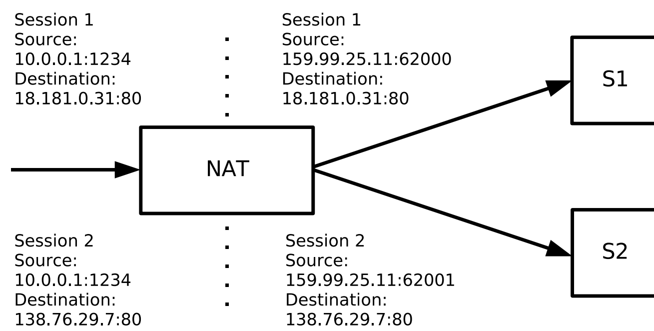

Figure 1.4: NAT device employing Endpoint Dependent Mapping

1.1.8 Endpoint Filtering

A NAT device performing Endpoint Mapping is further classified according to how liberally it accepts incoming traffic directed to an established public endpoint. That is, mapping refers to outgoing sessions and filtering refers to incoming traffic. The most extreme forms of filtering are *Endpoint Independent Filtering* and *Endpoint Dependent Filtering*. Endpoint Independent Filtering is the most liberal form of filtering and Endpoint Dependent Filtering is respectively the least liberal form of filtering [25].

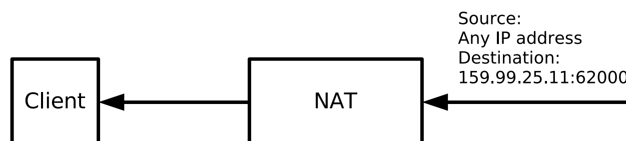

Figure 1.5: NAT device employing Endpoint Independent Filtering

A NAT device employing Endpoint Independent Filtering will accept incoming traffic to an established public endpoint from any host or endpoint in the public address realm. In our example of Figure 1.3, this means, that after an endpoint mapping has been established in the NAT, traffic from any host in the external realm is allowed to pass the NAT if the traffic is directed to the established NAT endpoint (159.99.25.11:62000). Figure 1.5 illustrates Endpoint Independent Filtering behavior.

A NAT device employing Endpoint Dependent Filtering accepts incoming traffic to an already-established public endpoint from only those external endpoints to which the host in the private realm has previously sent outgoing packets. Guha and Francis [4] estimate that a vast majority (94.2%)[6] of all NAT devices employ Endpoint Dependent Filtering.

1.1.9 User Datagram Protocol Hole Punching

In our previous examples in Figure 1.3 and Figure 1.4, one of the hosts has been in private realm while the other has been in the public realm. That is, only one of the hosts has been behind a NAT. In real world, this is not always the situation. We can have a scenario where both of the hosts reside behind NAT devices. This scenario raises a different kind of problem.

We noted in Section 1.1.2 that traditional NAT sessions are uni-directional i.e. only outbound sessions from the private network are allowed. Now, how the situation, where both hosts are behind a NAT, can be solved, when only outgoing connections can traverse the middlebox?

The solution to this problem is a technique called UDP hole punching [25]. It involves an additional proxy P, and it has the limitation that it works only when both of the NAT devices employ Endpoint Independent Mapping. UDP hole punching allows hosts to “punch holes” through the NAT device and establish direct connectivity with each other. When UDP hole punching is used, hosts initiate communication using both the IP address of P and the IP address of

[6] 81.9% are both address and port dependent and 12.3% are only address dependent.

the host they attempt to contact. However, before this, the peer behind NAT must register its public endpoint to P.

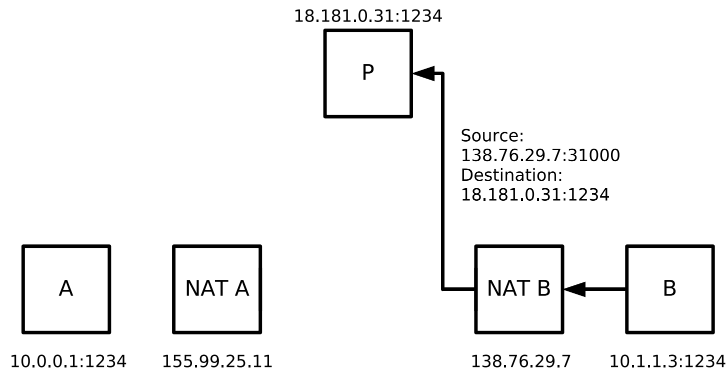

Figure 1.6: Registration to server

Suppose hosts A and B have private IP addresses and lie behind a different network address translators as in Figure 1.6. Both of the hosts use UDP source port 1234, and the proxy P is listening on port 1234. Host B has initiated an UDP communication session with P, causing NAT B to assign its own public UDP port 31000 for B's session with P. Accordingly, P stores public endpoint 138.76.29.7:31000 for host B.

Suppose now, that host A wants to establish a UDP communication session directly with host B. We are assuming that NAT A and NAT B are employing Endpoint Dependent Filtering. If A simply starts sending UDP packets to B's public endpoint (138.76.29.7:31000), then NAT B will discard these packets because it employs Endpoint Dependent Filtering. Bear in mind that Endpoint Dependent Filtering means that the source addresses and Transport identifier are relevant. Therefore all traffic not originating from source address of P (18.181.0.31:1234) is discarded.

To reach host B, host A needs to relay messages through proxy P. P relays the UDP packets and changes the source address from NAT A's public address (155.99.25.11:62000) to P's public address (18.181.0.31:1234). The packets reach host B now. However, when host B replies to A, it must also relay the traffic via P, as direct connections to host A are discarded because of the Endpoint Dependent Filtering at NAT A. Relaying all traffic this way has the obvious disadvantages that it consumes the processing power of the server and network bandwidth. UDP hole punching solves these issues.

The crux of the UDP hole punching technique is that host A starts to send UDP messages directly to B's public address, and *simultaneously* relays a request through P to B, asking B to start sending UDP messages to A's public address. UDP hole punching technique is visualized in Figure 1.7. The operation can be divided into four phases. Note that the registration of Figure 1.6 has taken place before these phases.

1. Host A sends a request to host B, asking B to start sending UDP messages

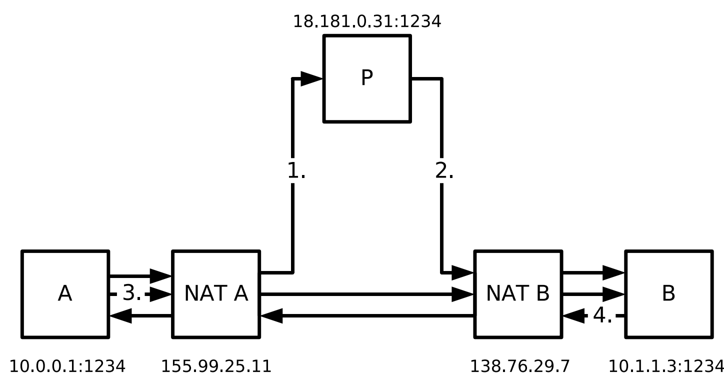

Figure 1.7: User Datagram Protocol Hole Punching

to A's public address. The message is directed to P.

2. Proxy P has a registry entry matching B's public endpoint, and the message from host A is relayed to B's public endpoint. However, P changes the source address of the message before relaying. The message has source address 18.181.0.31:1234 and destination address 138.76.29.7:31000 now. NAT B has a state matching this address pair and thus the message is allowed to pass NAT B.

3. *Simultaneously* with phase one, host A starts to send UDP messages to B's public endpoint. This opens a state in NAT A matching B's public endpoint and A's public endpoint. These messages, however, still get blocked at NAT B, since there is no state in NAT B matching these endpoints.

4. Once the messages of phase two reach host B, B starts to send messages to host A as requested. These messages open a state in NAT B matching A's public endpoint and B's public endpoint, allowing A and B to communicate directly.

If all of the NAT devices support Endpoint Independent Filtering, UDP hole punching is unnecessary. However, HIP NAT traversal design does not require NAT devices to support Endpoint Independent Filtering and therefore employs UDP hole punching. The reason for this is that Endpoint Independent Filtering is supported only in 5.8% of the deployed NAT devices according to Guha and Francis [4] and it is not mandated in RFC 4787 [2, REQ-8].

The Endpoint Independent Mapping property is required from all of the NAT devices on the path between the communicating hosts. If there is a single on-path NAT device not supporting this property, the NAT traversal for HIP will fail. The NAT traversal for HIP assumes Endpoint Independent Mapping for four reasons. First, majority of the deployed NAT devices (70.1%) support it [4]. Second, Endpoint Independent Mapping is also mandated by [2, REQ-1]. Third, not assuming Endpoint Independent Mappings would require inefficient triangular routing through a relay in a similar way as in Traversal Using Relay

NAT (TURN) [20]. Fourth, the Internet Draft *HIP Extensions for the Traversal of Network Address Translators* [12] assumes that the system initiating the HIP exchange starts to use a relay only when UDP hole punching fails.

The UDP hole punching method also works when either or both hosts are in a publicly addressable network. UDP hole punching technique even works automatically with multiple NATs, where one or both hosts are removed from the public Internet via two or more levels of address translation [25].

1.2 Host Identity Protocol

1.2.1 Host Identity Protocol Architecture

There are two global namespaces in the Internet today: IP addresses and DNS names. IP addresses are used both to identify and to locate a host. This dual role of IP addresses limits the flexibility of the Internet architecture and makes mobility and IP address renumbering difficult. In particular, traditional transport protocols (TCP, UDP) are bound to IP addresses and disconnect when an IP address changes. Three main problems with current Internet namespaces are:

1. Dynamic readdressing is not supported.
2. Lack of security and privacy.
3. Authentication for systems and datagrams is not provided.

To overcome the problems related to the current use of IP addresses, the HIP architecture adds a cryptographically based namespace, the Host Identity, and adds a new layer between the network and the transport layer in the TCP/IP stack. This modification is illustrated in Figure 1.8.

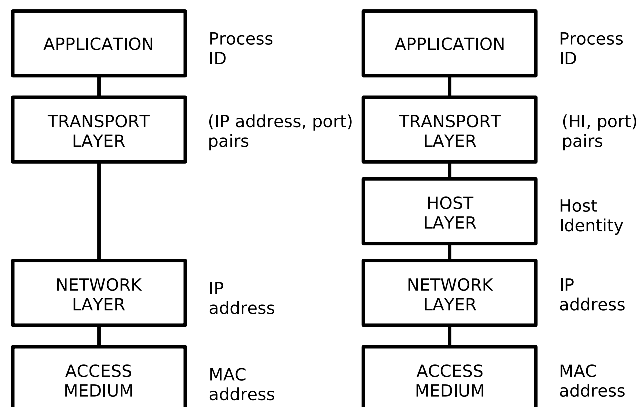

Figure 1.8: The current IP stack and the HIP based stack

In the current Operating Systems, each process is identified by a Process ID (PID). When the process establishes transport layer connections to other hosts,

the transport layer connection is identified using the source and destination IP addresses as well as the source and destination ports, and the protocol number. On the IP layer, the IP address is used as the endpoint identifier, and on the link layer, the Media Access Control (MAC) address of the network interface is used.

In the HIP architecture, each host has at least one Host Identifier (HI), which can be either public or anonymous. Each host is strongly recommended to have at least one public and one anonymous HI. The main purpose of the anonymous HIs is to provide privacy protection to the host, should the host not wish to use its public HIs. In addition, Rivest, Shamir, & Adleman (RSA) and Digital Signature Algorithm (DSA) algorithms are mandatory. In total, this results in four identifiers per host.

The transport layer connections are identified using the source and destination HIs as well as the source and destination ports. HIP then provides a binding between the HIs and the IP addresses. Accordingly, the 5-tuple socket becomes <*protocol, source HI, source port, destination HI, destination port*> from conventional <*protocol, source IP, source port, destination IP, destination port*>.

The public Host Identifiers are usually stored in the Domain Name System (DNS) or distributed using some other mechanism, such as a Public Key Infrastructure (PKI) or Open Distributed Hash Table (OpenDHT) [1]. Fundamentally, a HI is a public key of an asymmetric key pair. It serves as the endpoint identifier of the host. The endpoint owns the private key of the key pair, and therefore it is rather simple for the endpoint to prove that it owns the HI. In other words, it is very difficult for other endpoints to claim ownership of the HI.

Public keys tend to be rather long, and therefore the HIP architecture also includes fixed size presentations of the HI. A Host Identity Tag (HIT) is a 128-bit hash of the HI. The HIT can be used in IP address sized fields in Application Programming Interfaces (APIs) and protocols, because the HIT is the same length as an IPv6 address. Another presentation of the HI is the Local Scope Identifier (LSI), which has a size of 32 bits. LSIs provide backwards compatibility with IPv4 addresses, and are only used internally in the host.

The HIP architecture solves the three aforementioned problems present in the current TCP/IP architecture. The problem of dynamic readdressing becomes trivial, because the IP address no more functions as an endpoint identifier. A host may easily change its HI to IP address bindings as it moves. Anonymity is provided by anonymous HIs, and because the namespace is cryptographically based, it becomes possible to perform authentication based on the HIs. Furthermore, as all of the upper layer protocols are bound to the HI instead of the IP address, the IP address can now be used solely for routing purposes [13, 15].

1.2.2 Base Exchange

HIP communication is based on two different protocols. First, the Host Identity Protocol itself, for the establishment and management of communication. Second, the IPsec Encapsulating Security Payload (ESP) for application data encryption and authentication. We describe the communication establishment in this Subsection, and move on to the user and application data exchange in the next Subsection.

The first phase of HIP communication is the HIP base exchange. The base exchange is a two-party cryptographic protocol used to establish a communications context between two end-hosts designated by their Host Identies. This context is called *a HIP association* and it is maintained by procedures defined in the Host Identity Protocol [15]. By definition, the system initiating a HIP exchange is the Initiator (I), and the peer is the Responder (R). This distinction is discarded once the base exchange completes.

The purpose of the HIP base exchange is to create assurance that the peers indeed possess the private keys corresponding to their Host Identifiers. In consequence, the base exchange creates a pair of IPsec ESP Security Associations (SAs), one in each direction. Both the base exchange and the updating procedure of a HIP association are based on the exchange of HIP packets between the communicating hosts. Different types of HIP packets exist, however, they all have the same structure described in Subsection 1.2.5.

Before the base exchange can take place, the Initiator needs the locator and the HI of the Responder. Therefore, I needs to perform a DNS-query, using the Fully Qualified Domain Name (FQDN) of R as a search key. The Domain Name Server responds with the HI and the IP address of R. Figure 1.9 shows the process of base exchange.

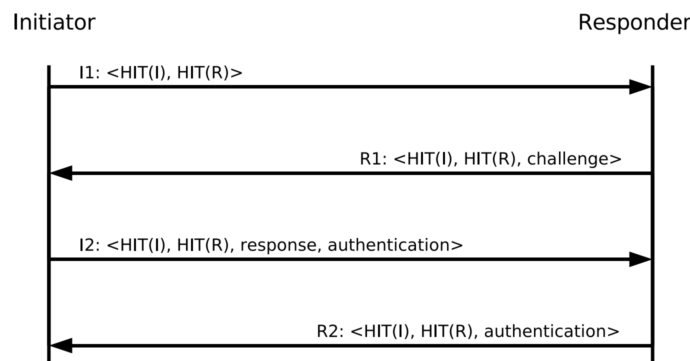

Figure 1.9: Base exchange [16]

The base exchange consists of four packets, namely the HIP Initiator Packet *I1*, the HIP Responder Packet *R1*, the Second HIP Initiator Packet *I2*, and the Second HIP Responder Packet *R2*. The first packet, I1, initiates the exchange, and the last three packets, R1, I2, and R2 constitute a standard authenticated Diffie-Hellman key exchange. In Figure 1.9 we have following phases:

1. The Initiator first sends a trigger packet, I1, to the Responder. The packet contains only the HIT of I and the HIT of R.

2. The Responder replies with an R1 message, which contains the HITs of I and R as well as a challenge (computational puzzle) based challenge for I to solve. The level of difficulty of the puzzle can be adjusted based on the level of trust with the Initiator, current load, or other factors. The

purpose of the challenge is to make the protocol resistant to Denial of Service (DoS) attacks. It allows the the Responder to delay state creation until receiving an I2.

3. Initiator solves the puzzle and sends in an I2 message the HITs of I and R, as well as the solution to the puzzle, and verifies the I2 public key signature. Without a correct puzzle solution, the I2 message is discarded.

4. An R2 packet finalizes the base exchange. It contains authentication [Keyed-Hash Message Authentication Code (HMAC) and HIP signature] and the HITs of I and itself. After the reception and validation of the R2 packet, a HIP association has been established between I and R. I and R have now performed mutual authentication and established SAs for ESPs, and can communicate in a secure manner [15, 16].

1.2.3 Secured Data Exchange Over IPsec

Once a HIP association has been established, the associated hosts can start sending data traffic. For this purpose HIP uses an end-to-end security protocol. Typically, but not necessarily, the data traffic in HIP is protected with Internet Protocol security (IPsec). As of today, the only completely defined method is to use IPsec Encapsulating Security Payload (ESP) to carry the data packets. In the future, other ways of transporting payload data may be developed, including ones that do not use cryptographic protection. When IPsec is used, the actual payload IP packets do not differ in any way from standard IPsec protected IP packets [15].

IPsec is a standard that provides security for IP based communication by encrypting and/or authenticating IP packets. IPsec is founded on two main cryptographic protocols, namely Encapsulating Security Payload (ESP) and Authentication Header (AH). Stephen Kent and Randall Atkinson have described the IPsec, ESP and AH architectures in RFC 2401 [9], RFC 2402 [10] and RFC 2406 [11] respectively.

While AH provides only integrity and data origin authentication, ESP provides integrity, data origin authentication and also encryption of the transmitted data. Since ESP is the main security protocol used in HIP, we will concentrate on ESP and do not discuss AH. Moreover, since this thesis focuses on the NAT traversal extensions of HIP, we will only give a general view of the ESP. The reader is advised to consult the Internet Draft *Using ESP transport format with HIP* [8] for a detailed description on this subject.

The concept of ESP consists of encrypting the IP packet payload and the transport header, adding an ESP header at the beginning of the packet and adding an ESP trailer with an authentication field at the end of the packet. Two different ESP modes exist, namely *transport* and *tunnel* mode.

In ESP transport mode an outgoing packet is obtained by encrypting the payload of an IP packet and then inserting an ESP header between the original IP header and encrypted data, and appending an ESP trailer. Finally, an ESP authentication field is computed and put at the tail of the packet. Figure 1.10 illustrates a TCP/IPv6 packet, and a corresponding secured version in ESP transport mode [10].

Although ESP tunnel mode is not supported by HIP, we will describe it here to lay ground for further discussion in this Subsection. In ESP tunnel mode not

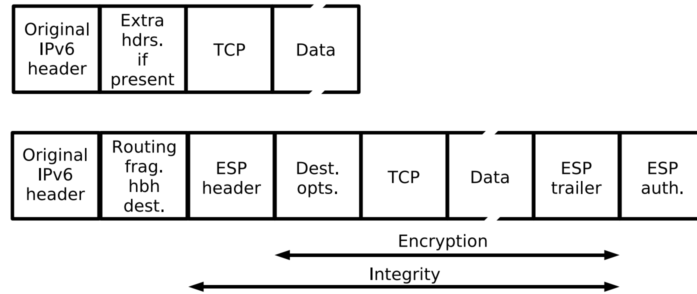

Figure 1.10: Original TCP/IPv6 packet (above) and ESP transport mode secured version of the packet (below)

only the IP payload but also the IP header is encrypted and included in the ESP payload as illustrated in Figure 1.11. The packet is forwarded using a new *external* IP header that is inserted in the head of the ESP packet.

The advantage of the tunnel mode is that while the external IP header can be modified by various middleboxes, the original *internal* IP header remains intact. The inner IP addresses can thus be used by the end-hosts as identifiers to reach each other. Consequently, the tunnel mode is used when the end-hosts are not in the public realm, but are located behind one or more middleboxes. The obvious disadvantage of the tunnel mode is that the external IP header introduces an overhead to the amount of transmitted data [10].

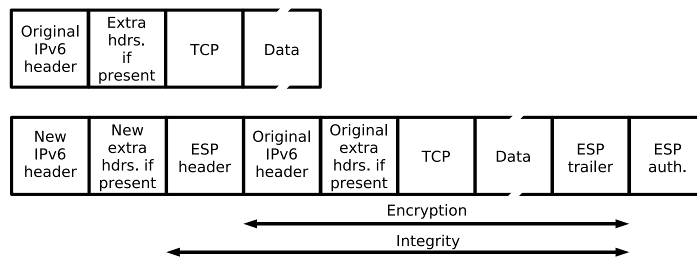

Figure 1.11: Original TCP/IPv6 packet (above) and ESP tunnel mode secured version of the packet (below)

To overcome this disadvantage, a new ESP mode has been developed that provides the same advantage as the tunnel mode but without its overhead. This new mode is named Bound End-to-End Tunnel (BEET) mode. Nikander and Melen have defined BEET mode in Internet Draft *A Bound End-to-End Tunnel (BEET) mode for ESP* [17]. The concept of the BEET mode is based on defining two pairs of IP addresses for each Security Association (SA). These pairs are designated as *inner addresses* and *outer addresses*.

The inner addresses are used for processing at all layers of the HIP based stack and in particular are the addresses seen and used by applications. The

outer addresses are only used to forward the packets on wire. This distinction between inner and outer addresses is perfect for the HIP model that specifically desires to separate endpoint identifiers from endpoint locators. Moreover, the BEET mode allows the use of HITs as inner addresses and since the HITs are already known and transmitted during the base exchange, there is no need to transmit them in each ESP packet. Effectively, this makes the format of ESP packets in BEET mode the same as in ESP transport mode. On the wire, the SPI acts as an short-hand for <*Source HIT, Destination HIT*> pair.

1.2.4 Host Identity Protocol State Machine

In the previous three Subsections, we have so far studied the motivation behind HIP, the HIP base exchange and the HIP data traffic exchange. We will go a little bit deeper into the details of the actual Host Identity Protocol next. First, we examine the state transitions in HIP, and then we study the structure of HIP packets.

The state machine of Figure 1.12 shows the major state transitions in the Host Identity Protocol. It focuses on the HIP I1, R1, I2 and R2 packets only, and it is presented in a single system view, representing either an Initiator or a Responder. Observe that receiving an I1 packet in UNASSOCIATED state does not generate a state transition. In fact, there is no such state as R1-SENT at all. [7] [8]

The idea is that the Responder has a number of pre-calculated R1 packets, and it selects one of these based on the information carried in I1. This way the R1 packets can be delivered as quickly as I1 packets arrive. When the Responder then later receives I2, it checks that the puzzle in the I2 matches with the puzzle sent in the R1. This makes it impractical for a potential attacker to first exchange one I1/R1, and then generate a large number of spoofed I2s that seemingly come from different IP addresses or use different HITs [15].

1.2.5 Host Identity Protocol Packets

The Host Identity Protocol consists of eight basic packets altogether. Of these, we have already introduced the four HIP packets used in the base exchange. The other half of the basic packets is composed of the HIP Update Packet *UPDATE*, the HIP Notify Packet *NOTIFY*, the HIP Association Closing Packet *CLOSE* and the HIP Closing Acknowledgement Packet *CLOSE_ACK*. UPDATE packet is for updating a HIP association, NOTIFY is for sending notifications and CLOSE and CLOSE_ACK are for closing a HIP association [15].

UPDATE, CLOSE and CLOSE_ACK packets are always both authenticated with an HMAC value and signed with an RSA or DSA signature. The HMAC is a light-weight signature based on symmetric cryptography. If either the HMAC or signature the validation fails in any of these packets, the packet is dropped. NOTIFY packet, however, is not authenticated with an HMAC parameter or signature. Therefore, NOTIFY packets are always treated as

[7] In Figure 1.12 UAL stands for Unused Association Lifetime, which is the time interval after which a host may begin to tear down an active association if no packet is sent or received.

[8] In Figure 1.12 EC stands for Exchange Complete, which is the time that the host spends at R2-SENT before it moves to ESTABLISHED state.

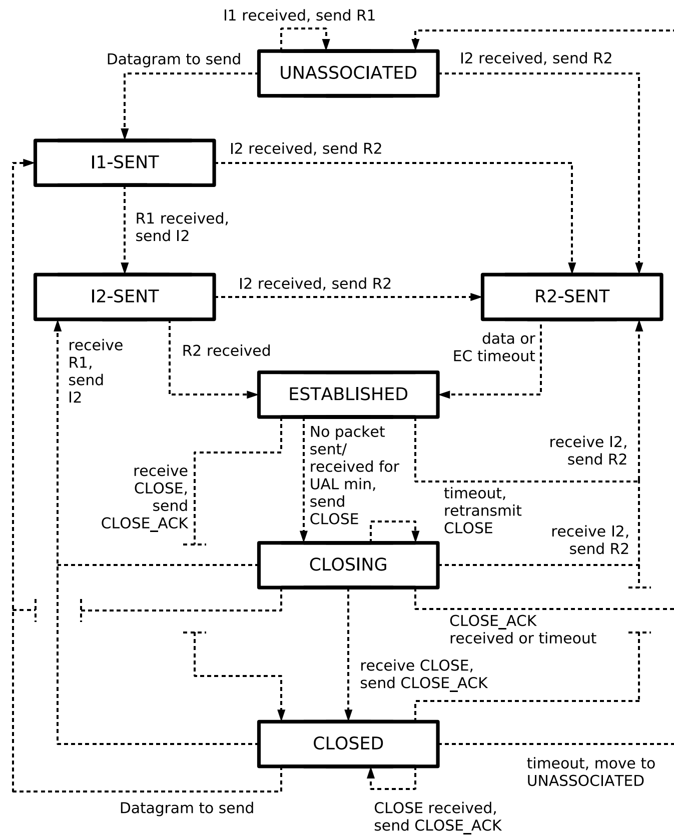

Figure 1.12: HIP state machine

Next Header	Header Length	0	Packet Type	Version	Res.	1
Length			Controls			
Sender's Host Identity Tag						
Receiver's Host Identity Tag						
HIP Parameters						

Figure 1.13: HIP packet header

informational, and no state information changes are made purely based on a received NOTIFY packet.

All HIP packets start with a fixed header. Figure 1.13 illustrates the HIP header format. It is logically an IPv6 extension header. The length of the *HIP Parameters* field varies from 0 to 2008 bytes, all other fields are mandatory and have a fixed length. Hence, the total length of a HIP packet varies between 40 and 2048 bytes. The HIP Parameters field is used to carry the public key associated with the sender's HIT, together with related security and other information. The field consist of Type-Length-Value (TLV) format encoded parameters ordered in ascending order [16].

The NAT traversal of HIP traffic is in large based on transmitting various NAT related information as HIP parameters. NAT traversal is studied in detail in the next Chapter. However, before moving on there, we take a look at the rendezvous mechanism and at the complications Network Address Translators introduce to Host Identity Protocol based communication.

1.2.6 Rendezvous Server

The HIP architecture introduces a rendezvous mechanism to help a HIP host contact a frequently moving HIP host. The mechanism involves a third party, namely the Rendezvous Server (RVS), that serves as an initial contact point for its clients. The clients of an RVS are HIP hosts that register their current location with the RVS. While the clients of RVS can be located in public or private realm, RVS itself is always located in the public realm. Note, that the RVS client equals to the base exchange Responder.

The registration process itself is a base exchange between the RVS client and the Rendezvous Server. The parties use additional parameters to announce their quality and grant or refuse registration. After the registration, other hosts can initiate a base exchange using the HIT of the Responder and the IP address of the RVS instead of the current IP address of the host they attempt to contact.

Figure 1.14 depicts a HIP base exchange involving a Rendezvous Server. Before the base exchange of Figure 1.14, host R has registered its location with the RVS. The Initiator of the base exchange first sends an I1 packet to RVS. This packet has the IP address of I as the source address and the IP address of RVS as destination. It also has the HITs of I and R. As the packet arrives at the RVS, the server changes the destination IP address of the I1 packet to the IP of R. Furthermore, the RVS also attaches two additional parameters to the relayed I1 packet. These parameters are the FROM and the RVS_HMAC parameters. The FROM parameter encapsulates the IP address of I that allows R to locate I. FROM parameter is integrity protected by the RVS_HMAC parameter.

When R receives the relayed I1, it first validates the HMAC. Next, R parses the IP address of I from the FROM parameter, and sends an R1 packet directly to I. The R1 packet has a special parameter, the VIA_RVS, which contains the IP address of RVS. The VIA_RVS parameter is included to allow network operators to diagnose possible issues encountered while establishing a HIP association via an RVS. After I receives R1, the base exchange continues directly between I and R without further help from the RVS.

The primary objective of the rendezvous extension is to improve reachability and operation when HIP hosts are mobile or multi-homed. However, the rendezvous extension is necessary when a middlebox separates the Responder

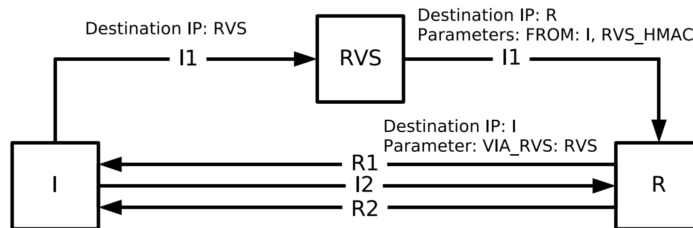

Figure 1.14: HIP base exchange via a Rendezvous Server

from the public realm. If a host has detected that it is behind a NAT, it must first register with the RVS when it is going to act as a Responder of a base exchange [12, 14].

1.2.7 Complications with Network Address Translators

Passing packets between different IP addressing realms requires changing IP header addresses, transport identifiers and checksums. This complicates the operation of HIP for two reasons. Firstly, as we noted in Section 1.1.4, only TCP, UDP and ICMP query type are permitted to traverse NAPT devices. This effectively blocks HIP signalling traffic. Blocking in the NAT device occurs because the *protocol* field of IPv4 header has a value that is not understood by the NAT. Secondly, ESP is not supported by all NATs.

NAT operates by modifying end host addresses, within the IP header. In the case of TCP/UDP packets, NAT needs to update the checksum in TCP/UDP headers, when an address in IP header is changed. However, as the TCP/UDP header is encrypted by the ESP, NAT is not able to make this checksum update. As a result, TCP/UDP packets encrypted in transport mode ESP will fail the TCP/UDP checksum validation on the receiving end and will simply not reach the target application. For this reason, HIP and ESP packets must be UDP encapsulated to traverse legacy NATs [6].

The scenario, where the Initiator or the Responder is behind a NAT device and a proxy P is in use, raises yet another complication. The problem setting is the same as we discussed earlier in Subsection 1.1.9. The incoming traffic cannot penetrate NAT devices because most of the deployed NATs employ Endpoint Dependent Filtering. Therefore, UDP hole punching has to be used. Implementing UDP hole punching complicates the functionality of both I and P.

A special case involving NATs is the case where both of the hosts reside behind the same NAT. This situation is referred as *hairpin translation*, because packets arriving at the NAT from the private realm are translated and then looped back to the private realm rather than being passed to the public realm. In hairpin translation, the private endpoint of the originating host is translated into its assigned public endpoint, and the public endpoint of the target host is translated into its private endpoint, before the packet is forwarded to the target host. Not all currently deployed NAT devices support hairpin translation, in which case HIP based communication in such a scenario is impossible [25].

In spite of all these complications, the new Host Identity namespace can assist with some NAT related problems. Envision a configuration of several HTTP servers behind single NAT. As a default rule, HTTP-requests are targeted at port 80. Without HIP, each server has to listen on different port and the NAT device then translates the port numbers. Whenever this setting changes, that is, a server is added or removed, the NAT device configuration has to be updated accordingly.

In HIP, each connection is identified using the HIs and ports instead of IP addresses and ports. Therefore, each of the server applications can listen on the default port 80 as all of the servers have their own unique HIs. Moreover, changes to the private realm HTTP server composition do not require reconfiguring the NAT device. The actual port numbers assigned by the NAT for different server hosts are hidden from the server applications by the IPsec tunnel. In short, HIP reintroduces the global name space.

1.3 Chapter Summary

Network Address Translators help with the IPv4 address shortage but at the same time they ruin the end-to-end connectivity of the Internet. Fundamentally the work of a NAT middlebox consist of modifying the transport and IP headers of a datagram. Several types of NAT middleboxes exist. NAT devices are classified according to their mapping and filtering characteristics. Since NATs allow only outbound sessions from the private network, an additional proxy and UDP hole punching technique is needed when both of the communicating hosts are behind NATs.

The Host Identity Protocol and the related architecture form a proposal to solve the dual role of IP addresses. HIP communication is based on two protocols, HIP itself for signalling, and ESP for secure data exchange. Currently deployed NAT devices can translate only TCP, UDP and ICMP packets, but other protocols are not supported. In the next Chapter, we study how we can accomplish successful NAT traversal for HIP packets, and how we can benefit from the UDP hole punching technique in the HIP architecture.

Chapter 2

Network Address Translator Traversal Using the Host Identity Protocol

This section binds together HIP and NAT. First, we discuss how NATs are detected in the HIP architecture, then we study UDP tunneling of HIP traffic and the NAT extensions for Rendezvous Server, and finally, we make a division between different scenarios involving NAT middleboxes. We describe how the Host Identity Protocol operates on a general level, but do not get involved with implementation level specifics yet.

2.1 Network Address Translator Detection

In order to know whether to use the NAT traversal mechanism or not, HIP hosts need to detect the presence of NAT middleboxes between them. The HIP architecture itself does not introduce any NAT detection mechanism but rather assumes the NAT is detected using some external mechanism. Therefore, no special HIP parameters are required in HIP control messages to detect NATs. The NAT detection must take place prior to base exchange, or after host movement, prior to sending UPDATE messages [12].

As an external NAT detection mechanism we describe Simple Traversal of User Datagram Protocol through Network Address Translators (STUN) herein.[1] STUN is a lightweight protocol that allows applications behind a NAT to first discover the presence and the type of a NAT, and then learn the address bindings allocated by the NAT. STUN works with many existing NATs, and does not require any special behavior from them. As a result, it allows a wide variety of applications to work through existing NAT infrastructure [22].

In STUN, the STUN client contacts a STUN server and the server then replies back letting the host know whether the STUN client is behind NAT or in public network. The STUN client is typically embedded in an application whereas STUN servers are attached to the public Internet and run on hosts

[1] Note that we describe STUN as defined in the RFC 3489 [22] and not as in the Internet Draft *Session Traversal Utilities for (NAT) (STUN)* [19].

dedicated to STUN. STUN can be used to detect NATs in all but one case where both of the hosts are behind the same NAT. This is commonly referred to as hairpin translation. We discussed hairpin translation earlier in Subsection 1.2.7 [22, 25].

In Figure 2.1 we have a typical STUN configuration. A STUN client is connected to private network 1 that in turn connects to private network 2 through NAT 1. Private network 2 is connected to the public Internet through NAT 2. The STUN server is located in the public Internet. In phase one of Figure 2.1, the client sends a Binding Requests to the server. In phase two, the server replies with a Binding Response.

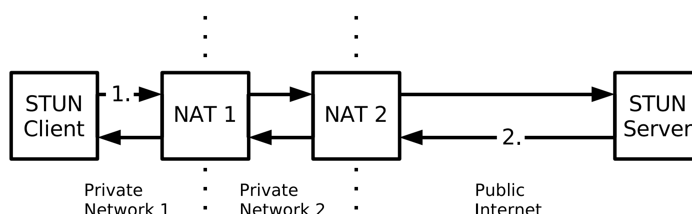

Figure 2.1: Example STUN Configuration

There are two types of requests in STUN, Binding Requests and Shared Secret Requests. The Binding Requests are sent over UDP and are used to discover the presence of a NAT, and to discover the public IP address and port mappings generated by the NAT. The Shared Secret Requests are used for authentication and message integrity, and they are sent over Transport Layer Security (TLS) over TCP. The details of these requests are out of the scope of this thesis.

The source address and port of the Binding Request received by the server are the mapped address and port created by the NAT middlebox closest to the server, i.e. NAT 2 in Figure 2.1. As the STUN client receives the Binding Response, it compares the IP address and port in the packet with the local IP address and port it bound to when the Binding Request was sent. If these do not match, the STUN client is behind one or more NATs. When a Binding Request arrives to a STUN server, the server examines the source IP address and port of the Binding Request, and copies them into a Binding Response that is sent back to the client.

To determine the type of the NAT middlebox the client is behind, the client sends an additional Binding Request, this time to a different IP address, but from the same source IP address and port. If the IP address and port are different from those in the first response, the client knows that the NAT device employs Endpoint Dependent Mapping. To determine the Endpoint Filtering characteristics of the NAT device, the client can send a Binding Request with flags that tell the STUN server to send a response from a different IP address and port than the request was received on [22].

Now that we have studied how NAT detection works, we can move on to the mechanisms that are used to penetrate NATs.

2.2 User Datagram Protocol Tunneling of Host Identity Protocol Traffic

Our NAT traversal mechanisms is based on encapsulating both the control and data traffic in UDP. In short, when UDP encapsulation is in use, the HIP packets and ESP packets are located in the payload of UDP packets. To refresh our memory, we have illustrated the structure of an UDP packet in Figure 2.2, it consists of only five fields. Figure 2.3 illustrates a BEET-mode UDP encapsulated TCP packet. NAT middleboxes see the encapsulated packet structure essentially as an UDP packet that uses IPv4 as network layer protocol.

Figure 2.2: UDP packet

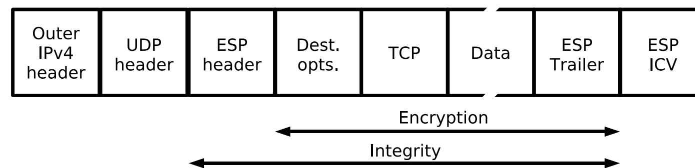

Figure 2.3: UDP encapsulation of an IPsec BEET-mode ESP packet containing a TCP segment

Some scenarios need also UDP hole punching mechanism to work, and for the UDP hole punching mechanism to work, the NAT device needs to employ Endpoint Independent Mapping as we discussed earlier in Section 1.1.9. Together these two mechanisms, UDP encapsulation and UDP hole punching, allow HIP traffic to traverse NAT middleboxes and enable hosts behind NATs to function as HIP base exchange Responders. We will limit our discussion on NAT traversal to HIP control traffic only, since the data traffic causes no state transitions in the HIP state machine and reuses the same UDP port numbers.

2.3 Network Address Translator Extensions for the Rendezvous Server

The registration process in NATted environments is identical to the case without NAT except that in NATted environments UDP encapsulation is used. The case

with NAT requires that the RVS stores also the source port number of R's NAT. This port number is copied from the I2 packet of the registration process. The RVS multiplexes I1 packets based on the destination HIT, not the UDP port numbers. If the registration process results in a successful HIP association, the translation state in the NAT must be kept valid with keep-alives. The keep-alive packets are UDP encapsulated NOTIFY packets that are sent at regular intervals to refresh the mapping in the NAT device. The default keep-alive interval the HIP architecture uses is 20 seconds. The Responder just discards the keep-alives [14].

2.4 Scenarios Involving Network Address Translators

Three basic cases exist for NAT traversal. In the first case, only the Initiator of a HIP base exchange resides behind a NAT. In the second case, only the Responder of a HIP base exchange resides behind a NAT. The third and most challenging case is where both of the parties reside behind a NAT device. Moreover, there is a special case where only I is behind a NAT and R uses the rendezvous service but is not located behind a NAT. We have such a special case, for example, when the Responder is a mobile host. To be reachable on UDP, both the peer and RVS have to listen on a single constant port, 50500.

Table 2.1: Notations used in this Section

Notation	Explanation
HIT-I	Initiator's HIT
HIT-R	Responder's HIT
IP-I	Initiator's IP address
IP-R	Responder's IP address
IP-RVS	IP address of the Responder's Rendezvous Server
IP-NAT-I	Public IP address of the NAT of the Initiator
IP-NAT-R	Public IP address of the NAT of the Responder
UDP(50500,11111)	UDP packet with source port 50500 and destination port 11111
UDP(11111,22222)	Example port numbers mangled by a NAT
UDP(44444,22222)	Port 44444 is used throughout the examples to denote the NAT translated source port of I2 as received by the Rendezvous Server during the registration.

Table 2.1 lists some short-hand notations used throughout this Section. We use example port numbers 11111, 22222 and 44444 to denote the NAT translated port numbers. In the examples, we assume that the translation state in the NAT expires after the I1/R1 exchange for illustration purposes. Therefore, we have different port numbers for I2/R2. The port number assigned by the NAT during I2/R2 exchange is used for all data traffic until the translation state in the NAT expires [12].

2.4.1 Initiator Behind a Network Address Translator

The simplest scenario involving a NAT middlebox occurs when only the Initiator of a base exchange is behind a NAT. The Responder is assumed to be located on the publicly addressable network. An example of this scenario is visualized in Figure 2.4. [2]

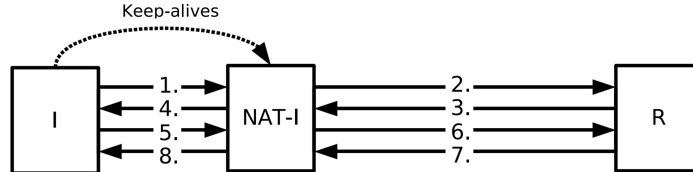

Figure 2.4: UDP encapsulated base exchange when the Initiator is behind a NAT

Table 2.2: Details of the Base Exchange of Figure 2.4

Phase	Network Layer	Transport Layer	Host Layer
1.	IP(IP-I, IP-R)	UDP(50500, 50500)	I1(HIT-I, HIT-R)
2.	IP(IP-NAT-I, IP-R)	UDP(11111, 50500)	I1(HIT-I, HIT-R)
3.	IP(IP-R, IP-NAT-I)	UDP(50500, 11111)	R1(HIT-R, HIT-I)
4.	IP(IP-R, IP-I)	UDP(50500, 50500)	R1(HIT-R, HIT-I)
5.	IP(IP-I, IP-R)	UDP(50500, 50500)	I2(HIT-I, HIT-R)
6.	IP(IP-NAT-I, IP-R)	UDP(22222, 50500)	I2(HIT-I, HIT-R)
7.	IP(IP-R, IP-NAT-I)	UDP(50500, 22222)	R2(HIT-R, HIT-I)
8.	IP(IP-R, IP-I)	UDP(50500, 50500)	R2(HIT-R, HIT-I)

Before beginning the base exchange, I detects that it is behind a NAT using an external mechanism such as STUN. I starts the base exchange by sending an UDP encapsulated I1 packet to R ①. The NAT middlebox forwards the I1 but replaces the source address IP-I with its own public address IP-NAT-I and the source UDP port 50500 with 11111 ②.

R replies with an R1 packet using the address and port information of I1 ③. Thus, the packet is destined to IP address IP-NAT-I and port 11111. When NAT receives the R1 packet, it replaces the destination of the packet with the IP address IP-I and port 50500 of I ④. I responds to R1 with an UDP encapsulated I2 packet having the same source and destination addresses and source and destination ports that it used for sending the corresponding I1 packet ⑤. The NAT again translates the source information ⑥. To illustrate a translation state timeout, the NAT chooses a different source port (22222) for I2 than for I1 (11111). Typically, there is no NAT translation state timeout between the I1/R1 and I2/R2 exchanges because these exchanges occur fast enough.

When R receives the UDP encapsulated I2 packet, it stores the source address information, i.e. address IP-NAT-I and port 22222, and uses that informa-

[2]In Figure 2.4 NAT-I indicates Initiator side NAT.

tion for further HIP communication with I. R finishes the base exchange with an R2 packet ⑦⑧. After the HIP association has been established, I starts to send periodic keep-alives. The phases of the base exchange are summarized in Table 2.2 [12].

2.4.2 Responder Behind a Network Address Translator

This Subsection describes the scenario where only the Responder of a base exchange is located behind a NAT. Before any communication between I and R can take place, the R must register to RVS. I is located in the public Internet. After the registration, I can initiate a base exchange with R via RVS, as illustrated in Figure 2.5. [3]

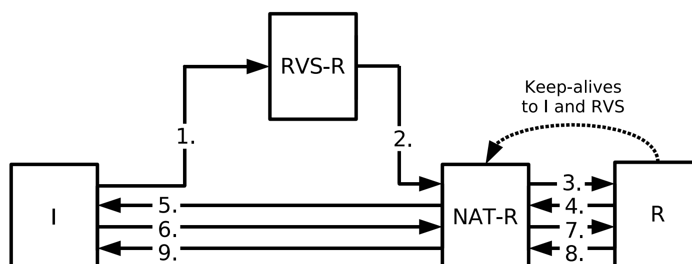

Figure 2.5: UDP encapsulated base exchange when the Responder is behind a NAT

To begin a base exchange with R, I sends an I1 packet to RVS ①. I sends this packet as a raw HIP packet without UDP encapsulation because there is no middlebox between I and RVS. The RVS, then, relays the inbound I1 packet to R ②. However, before doing so, RVS encapsulates the I1 packet in UDP because R has previously registered with the RVS using UDP encapsulation. Moreover, RVS also adds two additional parameters, namely RVS_HMAC and FROM, to the relayed I1, as described earlier in Section 1.2.6. RVS replaces the destination IP address of the packet with the IP address (IP-NAT-R) that it stored during the rendezvous client registration. The UDP header destination port is replaced with the stored port number (22222), that is, with the source port number of the registration I2 packet.

Next, the relayed I1 packet arrives to the NAT device. The NAT changes the destination address and the destination port in the packet according to the rules of the translation state present in the NAT ③. R receives the relayed I1 packet and parses the destination IP encapsulated in the FROM parameter. Using this IP address, R sends an UDP encapsulated R1 packet directly to I ④⑤. R also appends a VIA_RVS_NAT parameter to the message. The parameter contains the IP address of the Rendezvous Server and the port number the RVS used for relaying the I1.

I processes the R1 packet and replies using an I2 packet that is UDP encapsulated ⑥. The addresses and ports are derived from the received R1. The

[3] In Figure 2.5 RVS-R indicates Responder's Rendezvous Server and NAT-R indicates Responder side NAT.

NAT translates and forwards the I2 packet to R ⑦, and R responds using an UDP encapsulated R2 packet that uses address and port information derived from the received I2 packet ⑧⑨. This finishes the base exchange, and I and R can now send data traffic to each other directly without further assistance from the RVS. To maintain the translation state in the NAT, R starts to send periodic keep-alives to both of its established HIP associations, that is, to RVS and to I. The details of the base exchange are in Table 2.3 [12].

Table 2.3: Details of the Base Exchange of Figure 2.5

Phase	Network Layer	Transport Layer	Host Layer
1.	IP(IP-I, IP-RVS)		I1(HIT-I, HIT-R)
2.	IP(IP-RVS, IP-NAT-R)	UDP(50500, 22222)	I1(HIT-I, HIT-R, FROM:IP-I, RVS_HMAC)
3.	IP(IP-RVS, IP-R)	UDP(50500, 50500)	I1(HIT-I, HIT-R, FROM:IP-I, RVS_HMAC)
4.	IP(IP-R, IP-I)	UDP(50500, 50500)	R1(HIT-R, HIT-I, VIA_RVS_NAT(IP-RVS, 50500))
5.	IP(IP-NAT-R, IP-I)	UDP(44444, 50500)	R1(HIT-R, HIT-I, VIA_RVS_NAT(IP-RVS, 50500)
6.	IP(IP-I, IP-NAT-R)	UDP(50500, 44444)	I2(HIT-I, HIT-R)
7.	IP(IP-I, IP-R)	UDP(50500, 50500)	I2(HIT-I, HIT-R)
8.	IP(IP-R, IP-I)	UDP(50500, 50500)	R2(HIT-R, HIT-I)
9.	IP(IP-NAT-R, IP-I)	UDP(44444, 50500)	R2(HIT-R, HIT-I)

2.4.3 Both Hosts Behind a Network Address Translator

This subsection describes the details of enabling NAT traversal for HIP control and data traffic through UDP encapsulation, when I and the R are both behind two separate NATs, as is illustrated in Figure 2.6. Again, both hosts must first detect that they are behind a NAT before the base exchange. Also, R must register with the RVS before the base exchange can take place.

I starts the base exchange by sending an UDP encapsulated I1 packet to the RVS-R ①②. Because R has previously registered with the RVS, the Rendezvous Server knows the port number that it can use to communicate with R through the NAT (NAT-R in Figure 2.6). RVS adds a FROM_NAT parameter to the I1 packet. This parameter contains the source address and port of the I1 packet, which are effectively the address and port of the outermost NAT of I. The FROM_NAT is integrity protected by an RVS_HMAC parameter. Moreover, RVS replaces the destination IP address in the IP header of the packet with IP address (IP-NAT-R) that it stored during the rendezvous client registration. The UDP header destination port is replaced with the stored port number (22222), and the packet is relayed to R (3b)(4b).

In this case, where both Initiator and Responder are behind a NAT, the RVS sends two packets. In addition to relaying the I1 packet, it sends a NOTIFY packet back to I (3a)(4a). The NOTIFY packet contains the IP address and the port of the outermost NAT of R. This information is encapsulated in a

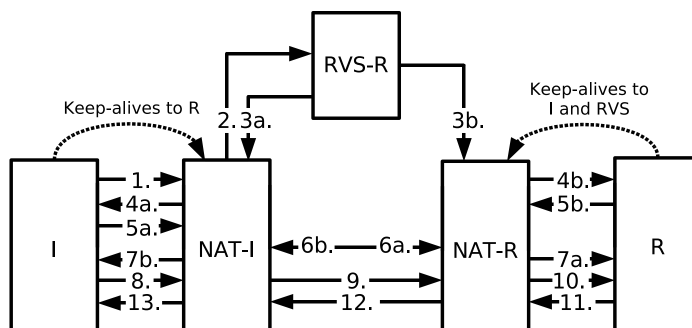

Figure 2.6: UDP encapsulated base exchange when both the Initiator and the Responder are behind a NAT

Table 2.4: Details of the Base Exchange of Figure 2.6

Phase	Network Layer	Transport Layer	Host Layer
1.	IP(IP-I, IP-RVS)	UDP(50500, 50500)	I1(HIT-I, HIT-R)
2.	IP(IP-NAT-I, IP-RVS)	UDP(11111, 50500)	I1(HIT-I, HIT-R)
3a.	IP(IP-RVS, IP-NAT-I)	UDP(50500, 11111)	NOTIFY(HIT-R, HIT-I, VIA_RVS_NAT(IP-NAT-R, 44444))
3b.	IP(IP-RVS, IP-NAT-R)	UDP(50500, 44444)	I1(HIT-I, HIT-R,) FROM_NAT(IP-NAT-I, 11111), RVS_HMAC)
4a.	IP(IP-RVS-R, IP-I)	UDP(50500, 50500)	NOTIFY(HIT-R, HIT-I, VIA_RVS_NAT(IP-NAT-R, 44444)
4b.	IP(IP-RVS, IP-R)	UDP(50500, 50500)	I1(HIT-I, HIT-R, FROM_NAT(NAT-I, 11111), RVS_HMAC)
5a.	IP(IP-I, IP-NAT-R)	UDP(50500, 44444)	NOTIFY(HIT-I, HIT-R)
5b.	IP(IP-R, IP-NAT-I)	UDP(50500, 11111)	R1(HIT-R, HIT-I, VIA_RVS_NAT(IP-RVS, 50500))
6a.	IP(IP-NAT-I, IP-NAT-R)	UDP(11111, 44444)	NOTIFY(HIT-I, HIT-R)
6b.	IP(IP-NAT-R, IP-NAT-I)	UDP(44444, 11111)	R1(HIT-R, HIT-I, VIA_RVS_NAT(IP-RVS, 50500))
7a.	IP(IP-NAT-I, IP-NAT-R)	UDP(11111, 50500)	NOTIFY(HIT-I, HIT-R)
7b.	IP(IP-NAT-R, IP-NAT-I)	UDP(44444, 50500)	R1(HIT-R, HIT-I, VIA_RVS_NAT(IP-RVS, 50500))
8-10.	I2(HIT-I, HIT-R), details similarly as in Tables 2.2 and 2.3.		
11-13.	R2(HIT-R, HIT-I), details similarly as in Tables 2.2 and 2.3.		

VIA_RVS_NAT parameter. Sending the NOTIFY to I is mandatory because UDP hole punching is required when both hosts are behind NATs. UDP hole punching, on the other hand, requires that I sends a packet directly to R.

Upon receiving the NOTIFY packet, I parses the IP address and the port of R from the VIA_RVS_NAT parameter and sends a NOTIFY message without any parameters to R (5a)(6a). This opens a state, in the NAT of I, matching the pair of the public endpoint of R and the public endpoint of I, as explained in phase 3 of Figure 1.6.

In the meantime, R receives the relayed I1 message, to which R responds by sending an R1 message (5b)(6b). The R1 message opens a state in the NAT of R matching the pair of the public endpoint of I and the public endpoint of R, as explained in phase 4 of Figure 1.6. R1 makes it to I because I already punched a hole into its own NAT with the empty NOTIFY message. However, should I only receive the NOTIFY with the VIA_RVS_NAT parameter, but never the R1 as would be the case for NAT with Endpoint Dependent Mapping, I may choose to use an external relay or to transition to E-FAILED state after a policy specific period. To keep the communication channel open after the base exchange, both I and R start to send keep-alives as indicated in Figure 2.6.

The Initiator and the Responder complete the base exchange with I2 ⑧⑨⑩ and R2 ⑪⑫⑬ messages. To avoid the NAT state timeout between I1/R1 and I2/R2 exchanges, I must refresh the state of the NATs by re-sending empty NOTIFY messages until it receives an R2 message. Table 2.4 summarizes the steps of the parallel base exchanges [12].

2.4.4 Use of the Rendezvous Service When Only the Initiator Is Behind a Network Address Translator

This Subsection examines the scenario where a host behind NAT uses rendezvous service to contact another host in the public realm. Figure 2.7 illustrates this scenario.

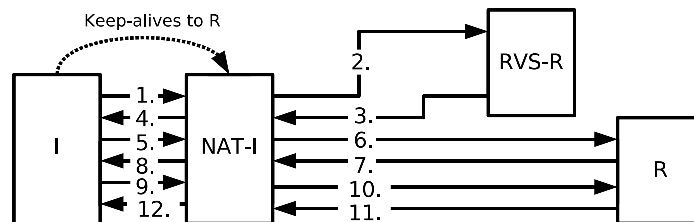

Figure 2.7: UDP encapsulated base exchange when the Initiator is behind a NAT and the Responder uses RVS

The mechanism described in this Subsection resembles the mechanism described in Subsection 2.4.3. There are two exceptions though, firstly the RVS does not relay the I1 message at all, but only replies to I with a NOTIFY. Secondly, upon receiving the NOTIFY from RVS, I does not send a NOTIFY to R but rather an I1 message. Moreover, I must continue retransmissions using the RVS. This is mandatory because NOTIFY messages are not protected with

signatures and can thus be forged by a rogue host.

I starts the base exchange by sending an UDP encapsulated I1 packet to RVS ①②. Because the RVS has a registry entry matching the HIT of R and the entry has no UDP port number, the server does not relay the I1 packet. There would be no use to relay this packet because the resulting R1 packet from R would not be able to traverse NAT-I as the NAT device employs Endpoint Dependent Filtering. Instead, RVS replies to I with a NOTIFY message that includes the IP address of R in a VIA_RVS parameter ③④. After I receives the NOTIFY, the base exchange continues in a similar manner to the base exchange in Figure 2.4 ⑤. . . ⑫. Table 2.5 has the details of the base exchange [12].

Table 2.5: Details of the Base Exchange of Figure 2.7

Phase	Network Layer	Transport Layer	Host Layer
1.	IP(IP-I, IP-RVS)	UDP(50500, 50500)	I1(HIT-I, HIT-R)
2.	IP(IP-NAT-I, IP-RVS)	UDP(11111, 50500)	I1(HIT-I, HIT-R)
3.	IP(IP-RVS, IP-NAT-I)	UDP(50500, 11111)	NOTIFY(HIT-I, HIT-R, VIA_RVS(IP-R))
4.	IP(IP-RVS, IP-I)	UDP(50500, 50500)	NOTIFY(HIT-I, HIT-R, VIA_RVS(IP-R))
5.	IP(IP-I, IP-R)	UDP(50500, 50500)	I1(HIT-I, HIT-R)
6.	IP(IP-NAT-I, IP-R)	UDP(22222, 50500)	I1(HIT-I, HIT-R)
7.	IP(IP-R, IP-NAT-I)	UDP(50500, 22222)	R1(HIT-R, HIT-I)
8.	IP(IP-R, IP-I)	UDP(50500, 50500)	R1(HIT-R, HIT-I)
9.	IP(IP-I, IP-R)	UDP(50500, 50500)	I2(HIT-I, HIT-R)
10.	IP(IP-NAT-I, IP-R)	UDP(33333, 50500)	I2(HIT-I, HIT-R)
11.	IP(IP-R, IP-NAT-I)	UDP(50500, 33333)	R2(HIT-R, HIT-I)
12.	IP(IP-R, IP-I)	UDP(50500, 50500)	R2(HIT-R, HIT-I)

2.5 Chapter Summary

STUN is a protocol that allows hosts behind a NAT to discover the presence and the type of NAT that separates the host from the public Internet. STUN also allows the hosts to learn the address bindings allocated by the NAT. To traverse legacy NATs, both HIP control and data traffic is sent on wire in the payload of UDP. Three basic and one special scenario that involve NATs exist. Using UDP encapsulation, special HIP parameters and the help of an RVS, we can overcome the difficulties NAT devices introduce to HIP traffic. However, we made one important assumption: none of the NAT devices en route must not employ Endpoint Dependent Mapping. If this assumption does not come true, the NAT traversal mechanisms introduced in this Chapter do not work.

Chapter 3

Implementation

This Chapter describes the implementation of the NAT traversal extension for HIP. To give the reader a more complete view of the overall software architecture and the interaction between the components, the main components and the internal structure of the software are introduced.

3.1 Implementation Overview

The goal of this thesis is to evaluate NAT traversal extensions for HIP by implementation. For this purpose, we extended the C-based Host Identity Protocol for Linux (HIPL) software that was developed in InfraHIP project at the Helsinki Institute for Information Technology (HIIT) and Helsinki University of Technology (TKK). At the time we started working on the project, the basic HIP protocol was nearly standardized and the project focused on developing the missing infrastructure pieces such as DNS, NAT, and firewall support.

Our contribution has lead to a solution that enables the NAT traversal of HIP traffic in all of the scenarios described in Chapter 2. Since the HIP protocol is still under development and since the nature of this project is experimental, our contribution consists not only of implementing the NAT extensions as defined in the *HIP Extensions for the Traversal of Network Address Translators* Internet Draft, but also of proposing corrections and enhancements to the draft itself. Moreover, the logic diagrams of Figures 3.2, 3.5, 3.7 and 3.8 are our contribution.

During the implementation process, we had to take into account protocol backwards compatibility issues with end-hosts and middleboxes. In practice, this meant that HIPL would still have to work with normal HIP traffic that is not encapsulated in UDP, and that the RVS would forward packets using UDP encapsulation only when needed.

3.1.1 Software Organization

The current HIP implementation can be divided into several components as shown in Figure 3.1. The components are:

- The HIP daemon *hipd* is the main component of the software. It listens to incoming connections and handles the base exchange, update and other HIP mechanisms.

- The *resolver* is the component that handles the mapping between HITs and IP addresses. The resolver is based on libinet6 library [27].
- The *hipconf*-utility is a program used for manual configuration of the HIP-daemon while the daemon is running.
- The *test software* consist of a server listening for incoming transport layer connections and a client that is used for connecting and sending data to the server.
- The *NAT extension* is the component that handles the UDP encapsulation and decapsulation. The decision whether to use the UDP encapsulation is made prior invoking the NAT extension. Most of the UDP and port selection logic is located in the base exchange code of the HIP daemon.
- The *RVS extension* handles the rendezvous service. We had to modify the existing RVS-module to handle and relay UDP encapsulated traffic. Furthermore, the RVS database that stores the HIT to IP address mappings had to be modified to include the UDP port number in addition to the IP address.

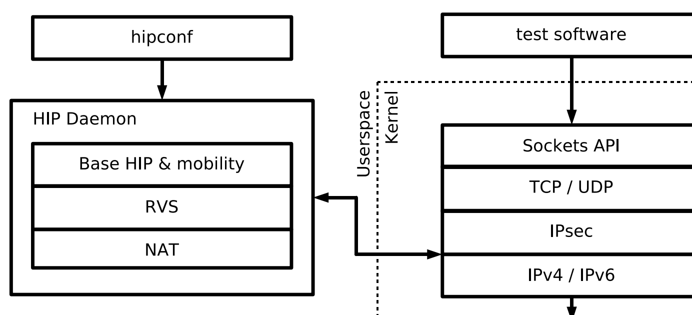

Figure 3.1: HIP implementation components

Of the components above, we describe our contributions, the NAT extension and the RVS extension, in more detail. We also discuss the UDP and port selection logic, thus we analyze parts of the hipd. We start with the RVS extension as its implementation is the easiest to decouple from the whole implementation. However, before going any further, we note, that our implementation is a single system implementation (mirrored state machine), which means that a HIP-host can take any of the roles of Initiator, Responder or Rendezvous Server. This role selection is presented in the logic diagram of Figure 3.2.

3.1.2 The Rendezvous Server Implementation

The rendezvous mechanisms was described earlier in Chapter 1.2.6. In the previous Chapter we explained that the RVS relays the I1 packet, replies with a NOTIFY packet or does both. The action the RVS chooses to take is based on whether one or both of the communication parties are behind a NAT. The

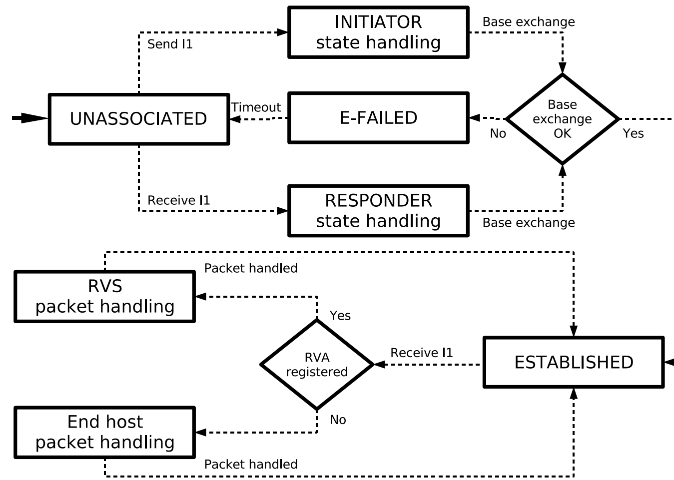

Figure 3.2: Logic diagram showing the role selection logic.

core of the Rendezvous Server are the rendezvous associations that are used for storing the information about HITs and IP addresses of the RVS clients. The implementation of the rendezvous association is shown in Figure 3.3.

The NAT-related data fields in Figure 3.3 are *hit*, *ip_ addrs*, *client_ udp_ port* and *send_ pkt*. Of these, hit and ip_addrs are used for storing the actual HIT-to-IP mapping. When the client is located behind a NAT, client_udp_port is used for storing the source port number of the registration I2 packet. The send_pkt field is a function pointer. It is set to point to a function that actually sends data on wire. Our implementation has two such functions, namely *hip_ send_ raw()* for sending packets without UDP encapsulation and *hip_ send_ udp()* for sending UDP encapsulated packets.

Figure 3.4 shows a simplified version of the *hip_ rvs_ relay_ i1()*-function that the RVS uses for relaying the incoming I1 packet. Before this function is invoked, a matching rendezvous association is already fetched from the rendezvous association data base. This rendezvous association is passed as the function argument, *rva*.

First, the code checks whether the incoming I1 was destined to UDP port 50500. If so, it is known that I is behind a NAT and we must insert a FROM _NAT parameter to the I1 packet to be relayed. Otherwise a plain FROM parameter is used. Next, the IP address of R is fetched from the rendezvous association. Finally, a HMAC value is added to the packet and the packet is sent on wire using the send_pkt-function of the matching rendezvous association.

The RVS occasionally needs to reply with a NOTIFY packet. To build a new NOTIFY, the IP address and port of R are first fetched from the rendezvous association database. Next, the RVS sends the NOTIFY to I over UDP using the address and port information from the received I1 packet.

To give the reader a more complete view of the RVS functionality, we present the corresponding logic diagram in Figure 3.5. In the diagram, we have illustrated both the registration process and the incoming I1 message handling. Note

```
typedef struct hip_rendezvous_association{
    struct list_head      list_hit;
    atomic_t              refcnt;
    spinlock_t            lock;
    hip_rvastate_t        rvastate;
    uint32_t              lifetime;
    struct in6_addr       hit;
    struct in6_addr       ip_addrs[HIP_RVA_MAX_IPS];
    in_port_t             client_udp_port;
    struct hip_crypto_key hmac_our;
    struct hip_crypto_key hmac_peer;
    hip_xmit_func_t       send_pkt;
}hip_rva_t;
```

Figure 3.3: The rendezvous association data structure

```
int hip_rvs_relay_i1(const struct hip_common *i1,
                     hip_rva_t *rva,
                     const hip_portpair_t *i1_info){

   struct hip_common *i1_to_be_relayed = NULL;
   struct in6_addr final_dst;
   int param_type = 0;

   if(i1_info->dst_port == HIP_NAT_UDP_PORT){
      param_type = HIP_PARAM_FROM_NAT;
   }
   else {
      param_type = HIP_PARAM_FROM;
   }

   hip_build_network_hdr(i1_to_be_relayed, HIP_I1,
                         &(i1->hits), &(i1->hitr));
   hip_rvs_get_ip(rva, &final_dst);
   hip_build_param(i1_to_be_relayed, param_type);
   hip_build_param_rvs_hmac_contents(i1_to_be_relayed, &rva->hmac_our);
   rva->send_pkt(&final_dst, rva->client_udp_port, i1_to_be_relayed);
}
```

Figure 3.4: The relay()-function

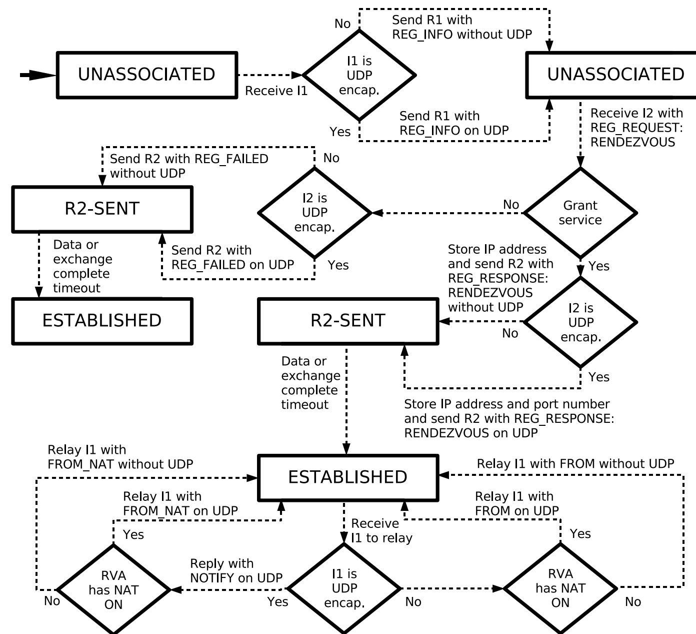

Figure 3.5: Logic diagram showing Rendezvous Server functionality

that if the RVS decides not to grant the service, a HIP association is still created between the RVS client and the server. In that case, however, R will not be reachable via the RVS.

3.1.3 The Network Address Translator Extension Implementation

The actual implementation of the NAT traversal extension is rather straightforward. Based on various state information, the NAT extension switches *"on"* or *"off"* the NAT status for a HIP association as shown in Figure 3.6.

```
int hip_nat_on_for_ha(hip_ha_t *entry){
   hip_hadb_set_xmit_function_set(entry, &nat_xmit_func_set);
   entry->nat_mode = 1;
}
```

Figure 3.6: Setting the NAT status on for a HIP association

First, the HIP association function pointer is set to point to *hip_send_udp()*, and then, the NAT status of the association is switched on. We discuss the logic behind the NAT status switching, i.e. when and on what basis is the function illustrated in Figure 3.6 called, next.

3.2 User Datagram Protocol and Port Selection Logic

The decision to use UDP encapsulation depends on whether the local host, the peer or both are behind a NAT. When there is a NAT between the local host and the public realm, the decision is not a hard one. We just use UDP encapsulation for each HIP association since all traffic traverses the NAT. For this kind of situation our implementation has a global NAT status variable *hip_nat_status* that can be set on or off using the hipconf tool.

When the peer is behind a NAT, and the local host is in the publicly addressable network, the situation becomes a little more complicated. The decision to use UDP encapsulation is now based solely on whether the base exchange or NOTIFY packets we receive are UDP encapsulated or not. Furthermore, the UDP target port has to be chosen based on the source port of the base exchange packets received. Also, there is a potential timeout between the I1/R1 and I2/R2 exchanges. Therefore, we cannot use the UDP port from the transport header of I1 or R1 packets for the ESP traffic, but have to wait for the I2 or R2 packet to arrive because the port number may change. This applies to all scenarios involving NATs.

3.2.1 Responder Side Logic

Let us first examine the scenario of Figure 2.4 where only the Initiator of a base exchange is behind a Network Address Translator. We examine the scenario from the Responder side, since the Initiator has the global NAT status set on, and there is no logic to study. To avoid DoS attacks, we need to keep R stateless until the I2 packet is processed successfully. Especially, the R1-SENT state does not exist. Thus, R stays in UNASSOCIATED state although the R1 packet is sent.

When R receives an I1 packet, R just sends one of its pre-calculated R1 packets to I. When the received I1 packet was UDP encapsulated, the R1 packet is also encapsulated and sent using the source port of I1 as destination port. R stores the source port of the I2 packet for further communication with the peer. When R receives an I2 packet in response to the R1 packet, it replies with an R2 packet that is encapsulated in UDP. Figure 3.7 illustrates the UDP and port selection logic of R.

3.2.2 Initiator Side Logic

In this Subsection, we discuss the logic of the Initiator in the scenario of Figure 2.5. The Responder is behind a NAT, it has the global NAT status set on, and there is no logic to study. The main difference between the logic of I and the logic of R in the scenario of the Subsection 3.2.1 is that now I sends the I1 packet from a state, namely from I1-SENT. Therefore we have an unestablished HIP association when R1 packet arrives. The logic diagram in Figure 3.8 summarizes the behavior of I.

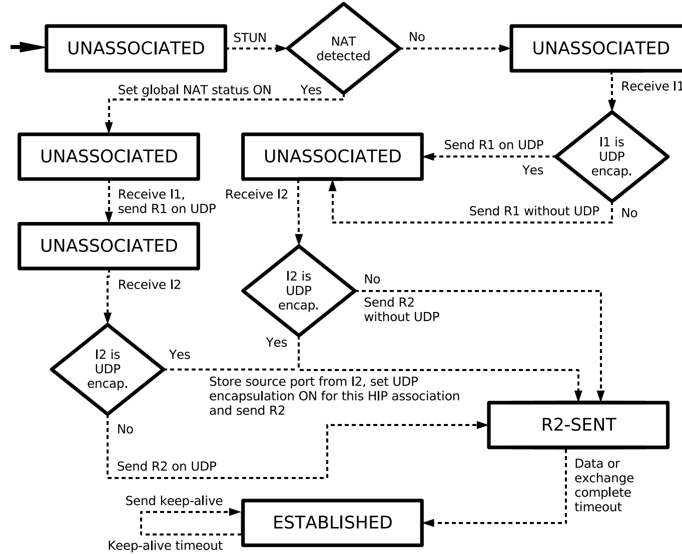

Figure 3.7: Logic diagram showing the UDP and port selection logic of the Responder

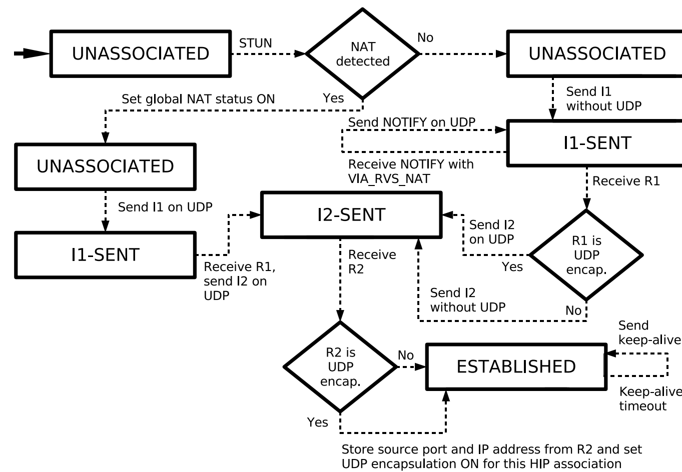

Figure 3.8: Logic diagram showing the UDP and port selection logic of the Initiator

3.2.3 Updating a Host Identity Protocol Association

After a successful base exchange, either host can change its network location. This Subsection describes how our implementation handles such mobility in the presence of NATs. However, the double jump scenario, where both hosts move simultaneously, is excluded because the current version of the related Internet Draft [12] does not specify the operating instructions for such a scenario. It is anticipated that the RVS just relays the first UPDATE packet. The RVS does not have to store the UPDATE packet temporarily. If the receiver of the UPDATE message has not yet acquired a new IP address when the RVS relays the message, and thus cannot be reached, the sender keeps retransmitting the message until it reaches its target or until the receiving host can be assumed to be out of reach.

The Rendezvous Server UPDATE relaying process is similar to the relaying process of I1. The RVS adds a FROM or FROM_NAT parameter depending on whether the received packet was UDP encapsulated, and protects the packet with a HMAC. Upon replying to the UPDATE, the corresponding host adds a VIA_RVS or VIA_RVS_NAT parameter to the reply. The evaluation of this feature is further work.

When the mobile host moves, it should detect the presence of NATs using STUN or some other external mechanism. The HIP association is updated using the UPDATE packet. When the peer is in the publicly routable network, the peer makes the decision to use UDP encapsulation solely based on the UDP encapsulation status of the received packets. It is therefore of great importance that the mobile host encapsulates the UPDATE packet within UDP only when it is behind a NAT. The corresponding host must reply using UDP when the packet was encapsulated within UDP, and without UDP when the UDP header was not present in the UPDATE packet.

The implementation switches the NAT status on or off based on the destination port of the transport header of the received UDP packet. When the UDP packet was destined to port 50500, the implementation switches on the UDP status. Any other port number, or the absence of the transport header causes the UDP status to be switched off.

3.3 Chapter Summary

Our contribution consist of the implementation of the NAT and RVS extensions, improvements on the HIP NAT traversal draft and creation of logic diagrams that visualize the functionality of the RVS and NAT extensions. The RVS extension is based on rendezvous associations. The NAT extension, on the other hand, operates on the basis of two different factors: the global NAT status and the destination port of the transport header of the received UDP encapsulated HIP packet.

Chapter 4

Discussion

The design and implementation effort spent on the NAT traversal extensions for HIP have brought up some future research and development ideas that we describe in this Chapter.

4.1 Adding STUN Support to the Implementation

Network Address Translator detection and STUN in particular were studied earlier in Section 2.1. We now discuss the possibility of adding STUN into Host Identity Protocol for Linux. Earlier, we used the word *lightweight* in context with STUN. This implies that it should be rather straightforward to implement a STUN extension to support our NAT traversal implementation. We choose to study one of the available open source STUN projects.

Our choice is the XSTUNT library [28] for three reasons. First, it is available for Linux, second, it is coded in C and C++ which means that it can be integrated to our HIP implementation, and third, because it appears to be rather extensively documented. The XSTUNT library is an implementation of the Simple Traversal of User Datagram Protocol through Network Address Translators and Transmission Control Protocol too (STUNT) protocol that extends the STUN protocol to include TCP functionality. We do not need the TCP functionality in our implementation, but we have to include it as a part of the XSTUNT library.

A closer look at the XSTUNT library reveals us that it consist of the following components:

- The *client* is a command line STUNT client that uses the functions in the *stun* and *udp* files to contact a STUNT server. A successful contact to the STUNT server results in an informative printout to the standard output. The printout has the type and optionally the preserved ports and hairpin translation properties of the NAT.
- The *server* is a STUNT server that listens for incoming Binding Requests. It is a command line application, but instead of the standard output, information is printed to a log. The functions in the *stun* and *udp* files are the basis of the STUN server.

- The *stun* file has the actual STUN protocol implementation as defined in RFC 3489 [22]. The TCP extension is also in this file. It has both the client and server side functions. Examples of these functions include, *stunBuildReqSimple()*, *stunParseMessage()*, *stunEncodeMessage()* and *stunServerProcessMsg()*.

- The *tlsServer* has the TLS functionality needed for the Shared Secret Requests.

- The *udp* file contains functions for sending and receiving messages on UDP.

Our NAT traversal implementation needs only the ability to detect NATs between different HIP hosts. Therefore, we do not need the server or tlsServer files at all. Nor do we need the command line client because the HIP daemon should be able to automatically detect NATs. The functionality that we need, is located in the stun and udp-files. To be exact, we would like to be able to invoke the stunBuildReqSimple() and stunParseMessage() functions before sending any HIP messages. The stunBuildReqSimple() function creates a Binding Request that is sent to a third party STUN server and the stunParseMessage() function then parses the information from the received Binding Response. Based on the results of the STUN protocol, the HIP daemon would then turn on or off the UDP encapsulation as needed. This would be especially handy when the HIP host is mobile.

To be able to call the aforementioned functions, we should bind a socket for STUN related network communication. Using this socket, the HIP daemon would then communicate with the STUN server, and automatically switch on the global NAT status in case it detects a NAT. We should also be able to force the UDP encapsulation on or off, because the STUN protocol may fail to detect NATs or because we choose not to use UDP encapsulation for some reason. Therefore, the NAT traversal extension should have three modes: *stun*, *force on* and *force off*.

From this quick study of the XSTUNT library, we can deduce that integrating the XSTUNT library to HIPL is possible, but it would require some modifications to the HIP daemon and to the hipconf tool.

4.2 Further Analysis of the Both Hosts Behind Network Address Translator Case

In this Section, we analyze how the base exchange behaves in the case where both hosts are behind different NATs of which one or both do not employ Endpoint Independent Mapping. Figure 4.2 will form the basis for our discussion. For simplicity's sake, we use only example port numbers (11111, 22222, 33333 and 44444) instead of full endpoints. Also, when we say that the NAT is sending traffic, we mean the traffic that is actually sent by the endpoint, but which has been translated by the NAT. We assume that both of the NATs employ Endpoint Dependent Filtering in all of the three cases studied.

4.2.1 Both Network Address Translators Employ Endpoint Dependent Mapping

Let us begin with the example scenario where both of the hosts are behind NATs that employ Endpoint Dependent Mapping. Before the base exchange can take place, R must register with the RVS. After the registration, the RVS knows the port number (11111) that is open in NAT-R, and the base exchange advances as described in the following.

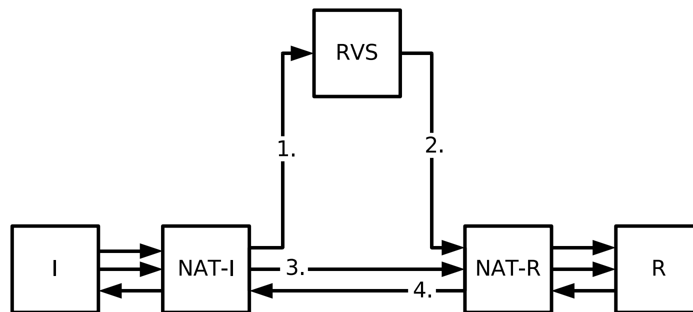

Figure 4.1: Both HIP hosts behind NAT

1. NAT-I sends an I1 message with source port 22222 and destination port 55000.
2. RVS forwards this I1 message to NAT-R with source port 55000 and destination port 11111. The message contains a FROM_NAT parameter with port number 22222. At the same time, the RVS lets I know port number 11111 by sending a NOTIFY message back to I (not shown in figure).
3. NAT-I sends a NOTIFY message with source port 33333 and destination port 11111. NAT-I assigns a different source port to the NOTIFY message than to the previously sent I1 message due to the Endpoint Dependent Mapping. The NOTIFY message punches a hole in NAT-I matching incoming traffic with source port 11111 and destination port 33333. Finally, NAT-R blocks the NOTIFY message because there is no mapping in NAT-R matching incoming traffic with source port 33333 and destination port 11111.
4. NAT-R sends an R1 message with source port 44444 and destination port 22222. Again, NAT-R assigns a different source port than with the registration to the RVS. This R1 message punches a hole in NAT-R matching incoming traffic with source port 22222 and destination port 44444. Finally, NAT-I blocks the message because there is no mapping in NAT-I matching incoming traffic with source port 44444 and destination port 22222.

As a result, the base exchange will fail because neither the NOTIFY nor the R1 message reaches its destination. Moreover, it does not make a difference which one the phases, 3. or 4., occurs earlier, since the created mappings do not match the incoming traffic in any case.

4.2.2 Initiator Behind a Network Address Translator That Employs Endpoint Dependent Mapping

In this scenario, I is behind a NAT that employs Endpoint Dependent Mapping, and R is behind a NAT that employs Endpoint Independent Mapping. Phases 1. and 2. detail similarly as in the previous Subsection. Phases 3. and 4. are described in the following.

3. NAT-I sends a NOTIFY message with source port 33333 and destination port 11111. NAT-I assigns a different source port to the NOTIFY message than to the previously sent I1 message due to the Endpoint Dependent Mapping. The NOTIFY message punches a hole in NAT-I matching incoming traffic with source port 11111 and destination port 33333. Although NAT-R employs Endpoint Independent Mapping, NAT-R still blocks the NOTIFY message because of the Endpoint Dependent Filtering.

4. NAT-R sends an R1 message with source 11111 and destination 22222. The source port is thus the same that was registered to RVS. The R1 message punches a hole in NAT-R matching incoming traffic with source port 22222 and destination port 11111. The R1 message gets blocked at NAT-I because there is no mapping matching incoming traffic with source port 11111 and destination port 22222.

Again, the base exchange will fail because neither the NOTIFY nor R1 message reaches its destination. The fact that the NOTIFY message does not reach R although NAT-R employs Endpoint Independent Mapping may come as a bit of a surprise. This behavior is not because of the mapping characteristics but, because of the filtering characteristics of NAT-R.

If the R1 of phase 4. is sent before the NOTIFY of phase 3., there is a mapping in NAT-R matching incoming traffic with source port 11111 and destination port 22222 when the R1 message arrives. Although the R1 is destined to the correct port (11111), it originates from a wrong port (33333), and therefore does not reach its target.

4.2.3 Responder Behind a Network Address Translator That Employs Endpoint Dependent Mapping

In this Subsection, we interchange the NAT properties of the scenario in Subsection 4.2.2. That is, we now have a scenario where I is behind a NAT that employs Endpoint Independent Mapping and R is behind a NAT that employs Endpoint Dependent Mapping. Phases 1. and 2. are the same as before. Phases 3. and 4. are as described in the following.

3. NAT-I sends a NOTIFY message with source port 22222 and destination port 11111. NAT-I assigns the same source port to the NOTIFY message than to the previously sent I1 message due to the Endpoint Independent Mapping. The NOTIFY message punches a hole in NAT-I matching incoming traffic with source port 11111 and destination port 22222, but NAT-R blocks the message because there is no mapping matching incoming traffic with source port 22222 and destination port 11111.

4. NAT-R sends an R1 message with source 33333 and destination 22222. The source port is the same that was delivered to R in the FROM_NAT parameter. The R1 message punches a hole in NAT-R matching incoming traffic with source port 22222 and destination port 33333, but NAT-I blocks the message because NAT-I has no mapping matching incoming traffic with source port 33333 and destination port 22222.

Accordingly, the base exchange fails also in this scenario. Due to the filtering characteristics of NAT-I, it is again insignificant which of the phases 3. or 4. occurs first. From these results, we conclude that our NAT traversal extension does not work when there is a single NAT employing Endpoint Dependent Mapping en route. We have summarized the NAT port assignment behavior of this Section in Table 4.1.

Table 4.1: Network Address Translator port assignment behavior

Both NATs Employ Endpoint Dependent Mapping		
	Expects	Receives
NAT-I	R1(11111, 33333)	R1(44444, 22222)
NAT-R	NOTIFY(22222, 44444)	NOTIFY(33333, 11111)
Initiator Behind a NAT That Employs Endpoint Dependent Mapping		
	Expects	Receives
NAT-I	R1(11111, 33333)	R1(11111, 22222)
NAT-R	NOTIFY(22222, 11111)	NOTIFY(33333, 11111)
Responder Behind a NAT That Employs Endpoint Dependent Mapping		
	Expects	Receives
NAT-I	R1(11111, 22222)	R1(33333, 22222)
NAT-R	NOTIFY(22222, 33333)	NOTIFY(22222, 11111)

4.3 Improvements in the Next Development Cycle

Our implementation is based on the first version of the Internet-Draft *HIP Extensions for the Traversal of Network Address Translators* [12]. However, the protocol characteristics presented in Chapter 2 are from the second version of the draft. The second version of the draft has two minor improvements over the first version. We go through these improvements in this Section.

4.3.1 Improvements in the Network Address Translator Keep-alives

The format of the NAT keep-alive messages has changed from the first version of the draft. In our implementation, empty UPDATE messages are used as NAT keep-alives, while the newer version of the draft uses empty NOTIFY messages for this purpose. The reasoning behind this change is that the NOTIFY messages suit better for this purpose because they are not acknowledged, and are handled as informational only. As described by Moskowitz et al. [15], the UPDATE and NOTIFY messages have the following properties:

UPDATE - HIP provides a general purpose UPDATE packet that can carry multiple HIP parameters, for updating the HIP state between two peers. UPDATE messages carry a monotonically increasing sequence number and are explicitly acknowledged by the peer. The UPDATE message is protected by both HMAC and HIP_SIGNATURE parameters, since processing UPDATE signatures alone is a potential DoS attack against intermediate systems.

NOTIFY - NOTIFY packets are unacknowledged. The receiver can handle the packet only as informational, and should not make any state information changes based purely on a received NOTIFY packet.

The downside of sending the NOTIFY message back to I is that it involves a privacy and availability issue. That is, when the rendezvous client registers with the RVS, it registers the IP address and a possible port number behind which the client is located. In Table 2.4 these values are IP-NAT-R and 22222. Now, when RVS replies back to I with a NOTIFY, it gives these values to I without first asking for a permission from R.

4.3.2 Improvements in the Both Hosts Behind Network Address Translator Case

The solution for the "both hosts behind NAT case" has changed since the first version of the draft. Our implementation has a slightly different solution than presented in the Subsection 2.4.3, where the Initiator responds to the NOTIFY packet received from the RVS by sending another NOTIFY message directly to the Responder. In our implementation, the Initiator responds to the NOTIFY packet received from the RVS by sending an I1 packet directly to the Responder. Thus in phase 5a of the Table 2.4 an I1 packet is sent instead of the NOTIFY packet.

By the time we implemented our NAT traversal extension, we believed that two parallel base exchanges were needed to guarantee successful NAT hole punching. Although this also works, it results in an unnecessary race between the parallel base exchanges. The race eventually terminates after the Initiator is in established state. A further examination on the both hosts behind NAT case showed us that it is adequate for the Initiator to just send a NOTIFY message.

4.4 Simple Traversal Underneath Network Address Translator Relay Usage

After the lessons learned from the discussion in Section 4.2, it is evident that the NAT extensions need to be further developed to cope with a situation involving NATs that employ Endpoint Dependent Mapping. The obvious solution for such a scenario is to relay all HIP signalling and data traffic via a third party server that resides in the public Internet. There are two ways to achieve full traffic relay; first a new HIP registration type could be created to allow the RVS to relay all traffic; second an external relay service could be used.

We began this Chapter with a discussion on the possibility of integrating STUN into HIPL. Conveniently, the STUN protocol has been further developed to include a full relay of packets. Therefore, when either I or R is behind a NAT employing Endpoint Dependent Mapping, a STUN server can be used to relay all traffic between these hosts. We have a closer look on the operation of STUN[1] relay usage [20] next. The STUN relay usage uses following definitions.

- *A Relayed Transport Address* is a transport address that terminates on a server, and is forwarded towards the client.
- *A STUN relay client* is a STUN client that obtains a relayed transport address that it provides to a small number of peers (usually one).
- *A STUN relay server* is a STUN server that relays data between a STUN relay client and its peer.

In a typical configuration, a STUN relay client is located in a private network and connected through one or more NATs to the public Internet. A STUN relay server is in the public Internet. The STUN relay usage defines several new messages and a new framing mechanism that add the ability for a STUN server to act as a packet relay.

First, the client sends an *Allocate request* to the server, which the server authenticates. The server generates an *Allocate response* with the allocated address, port, and target transport. A successful Allocate Request just reserves an address from the STUN server. Data does not flow through the allocated port until the STUN relay client asks the STUN relay server to permit the traffic. A permission can be opened either for UDP or TCP data streams, but since the NAT traversal mode HIP traffic is encapsulated in UDP, we study only the UDP relay properties of the STUN relay usage herein. The client can ask the STUN relay server to open an UDP permission by sending data to the far end with *a Send Indication* or by setting the default destination.

Once a permission is open, the client can then receive data flowing back from its peer. Initially this data is contained in *a STUN Data Indication* message. Since multiple permissions can be open simultaneously, the Data Indication contains *the Remote Address* attribute in order to the STUN relay client to know which peer sent the data. The client can send data to any of its peers with the Send Indication message. The data from the peer is only relayed to the client after the client sends packets towards the peer. Using the terms of the HIP architecture, this means that the Initiator (STUN relay client) must first contact the STUN relay server before any data can flow back from the Responder (peer).

Once the client wants to primarily receive from one peer, it can send *a SetActiveDestination request*. All subsequent data received from the active peer is forwarded directly to the client and vice versa. The client can send subsequent SetActiveDestination requests to change or remove the active destination. This

[1] There has been a lot of work on this area recently. For example, the ICE [18] specification has been updated frequently during the last year. Also, the possibility of using STUN server as a relay server has lead to the evolution of the acronym STUN itself. In addition to the original expansion of *Simple Traversal of User Datagram Protocol through Network Address Translators* [22], the expansions *Simple Traversal Underneath NAT* [20] and *Session Traversal Utilities for NAT* [19] exist. Moreover, the STUN relay usage was previously a stand-alone protocol called TURN.

request does not close other bindings in the STUN relay server. Data to and from other peers is still contained in Send and Data indications respectively.

Figure 4.2 illustrates an example STUN relay usage communication session. In the Figure, after the STUN client has reserved a relay address, it sends data to the peer using a Send Indication. Next, the peer sends data to the STUN client using a Data Indication. Finally, the STUN client decides to set the peer as an active destination, and all further data is sent without Send or Data Indication encapsulation until the STUN client decides to set another peer as an active destination.

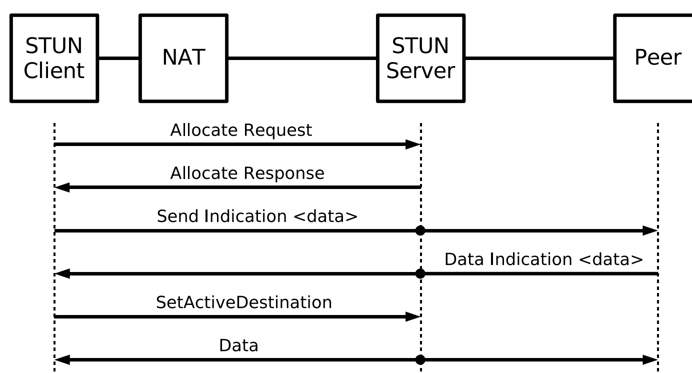

Figure 4.2: STUN relay usage

4.5 Next Steps in Network Address Translator Traversal

The problem of when to use a STUN relay server and when to use a normal Rendezvous Server is somewhat complicated since the base exchange hosts cannot communicate directly. If the Responder is behind a NAT that employs Endpoint Dependent Mapping, how can the Initiator know whether to launch the base exchange via an RVS or a STUN relay server? One way to solve this problem is to relay the whole base exchange via the RVS whenever there is NAT that employs Endpoint Dependent Mapping en route, and use the I2 and R2 packets to carry information about the Relayed Transport Address in use. The hosts can then exchange data via the STUN relay server using this Relayed Transport Address. The Interactive Connectivity Establishment (ICE) methodology [18] provides algorithms for selecting optimal route between the peers.

ICE is a methodology for NAT traversal for multimedia sessions established with the offer/answer model. In the offer/answer model, one host offers the other a description of the desired session from their perspective, and the other host answers with the desired session from their perspective. The offer/answer model is described in RFC 3264 [21]. ICE can be used by any protocol utilizing the offer/answer model.

ICE allows the hosts to discover enough information about their topologies

to potentially find one or more paths by which they can communicate. The basic idea behind ICE is as follows: each host has a variety of candidate transport addresses (combination of IP address and port) it could use to communicate with the other host. These might include:

- A transport address on a directly attached network interface or interfaces
- A translated transport address on the public side of a NAT
- A Relayed Transport Address of the STUN server the host is using.

The purpose of ICE is to discover which pairs of addresses will work. The way that ICE does this is to systematically try all possible pairs until it finds one or more that works [18].

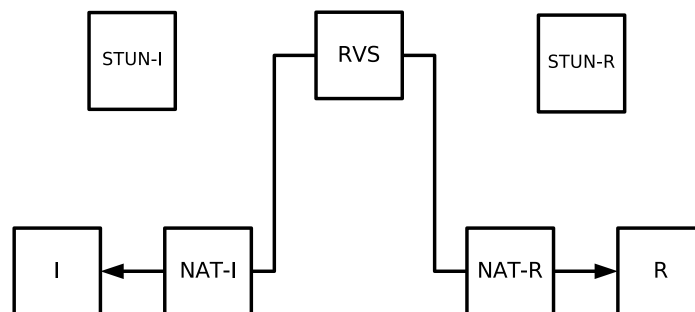

Figure 4.3: Interactive Connectivity Establishment deployment scenario

ICE could be used as it is for HIP NAT traversal, but there are some security related advantages in just merging the ideas from ICE to HIP. Figure 4.3 shows how we anticipate the next version of HIP NAT traversal based on ICE to work. Before ICE is used, both I and R have obtained a Relayed Transport Address from their STUN servers, and R has registered to RVS. I and R can either use their own STUN server, or the server can be the same. Next, I and R perform a base exchange in which all of the packets are relayed through RVS. The I2 and R2 packets carry the candidate transport addresses.

After the base exchange, ICE is run using HIP UPDATE packets for carrying the ICE messages. The hosts are now able to select which applicable route they use to communicate with each other.

4.6 Chapter Summary

The XSTUNT library can be integrated to HIPL to allow the HIP daemon to use STUN protocol for NAT detection. This integration would require some changes to XSTUNT, but carrying out these changes would be a smaller task than implementing the STUN protocol from scratch. We found out that NAT devices that perform Endpoint Dependent Mapping are very cumbersome to deal with. A single NAT of this kind on the path is enough to wreck our current NAT traversal design. The solution for this kind of scenario is to relay all

HIP signalling and data traffic via a third party proxy residing in the public Internet. We can use the STUN relay usage servers in combination with the ICE methodology for this purpose in the future.

Conclusions

Network Address Translators introduce a host of challenges to protocol designers. The root of these challenges is that the currently deployed NAT devices translate only TCP, UDP and ICMP packets by default. To make the situation more challenging, NAT traversal is further complicated by the lack of NAT standardization. As a result, the current Internet is now full of NATs whose characteristics vary from each other. During the course of this work, it has become evident that this diversity complicates NAT traversal mechanisms. When one solution is optimal in some network topologies, it is a poor choice in others.

In this thesis, we have presented a solution that enables HIP based communication through most types of legacy NATs. We applied the solution in Linux environment. Our implementation enables HIP Initiators behind a NAT to contact HIP Responders behind another NAT. The implementation is based on encapsulating all traffic in UDP, and on using a third party proxy, the Rendezvous Server, as an initial contact point when a receiving host is behind a NAT.

We claim that our implementation works on majority of network topologies. Moreover, the implementation is guaranteed to work when only I or R is behind a NAT — regardless of the type of the NAT. However, when both of the communicating parties are behind a NAT, the operation of the implemented NAT traversal extension is limited by the type of the NATs between I and R.

The limitations the type of the NAT middlebox set up to our implementation motivated us to study the theory of network address translation and to propose enhancements to the HIP NAT traversal specifications. We found out that the mapping and filtering characteristics are the two most dominating properties of NAT devices. We state that when both hosts are behind different NATs, a single NAT performing Endpoint Dependent Mapping is enough to cripple the implementation presented herein.

Our work has highlighted the importance of an effective NAT detection mechanism. Before a HIP host initiates a contact with another HIP host, it must be able to detect whether or not there is a NAT device on path. The details of how this detection is carried out remain future work. One possibility is to take advantage of the STUN protocol.

The results of this work suggest that we need to improve the NAT traversal extension to deal with NATs that perform Endpoint Dependent Mapping. For the time being, the only working solution for these kind of scenarios seems to be to relay both control and data traffic via a relay server. This area requires further studies. However, it is becoming more and more evident that we can use the ideas from ICE to create a more flexible NAT traversal mechanism than the current design as ICE supports dynamic negotiation of data relays.

Bibliography

[1] Jeff Ahrenholz. *Internet-Draft: HIP DHT Interface.* IETF, The Boeing Company, February 2007. `http://www.ietf.org/internet-drafts/draft-ahrenholz-hiprg-dht-01.txt`, work in progress.

[2] Francois Audet and Cullen Jennings. *RFC 4787: Network Address Translation (NAT) Behavioral Requirements for Unicast UDP.* IETF, Nortel Networks, Cisco Systems, January 2007. `http://www.ietf.org/rfc/rfc4787.txt`.

[3] Brian E. Carpenter and Scott W. Brim. *RFC 3234: Middleboxes: Taxonomy and Issues.* IETF, IBM Zurich Research Laboratory, February 2002. `http://www.ietf.org/rfc/rfc3234.txt`.

[4] Saikat Guha and Paul Francis. *Internet-Draft: NAT Behavioral Requirements for Unicast UDP Firewalls.* IETF, Cornell University. `http://www.imconf.net/imc-2005/papers/imc05efiles/guha/guha.pdf`.

[5] Tony Hain. *RFC 2993: Architectural Implications of NAT.* IETF, Microsoft, November 2000. `http://www.ietf.org/rfc/rfc2993.txt`.

[6] Matt Holdrege and Pyda Srisuresh. *RFC 3027: Protocol Complications with the IP Network Address Translator.* IETF, ipVerse, Jasmine Networks Inc., January 2001. `http://www.ietf.org/rfc/rfc3027.txt`.

[7] Opportunistic network and web measurement. `http://illuminati.coralcdn.org/stats`, retrieved 8th May 2007.

[8] Petri Jokela, Robert Moskowitz, and Pekka Nikander. *Internet-Draft: Using ESP transport format with HIP.* IETF, Ericsson Research NomadicLab, ICSAlabs - a Division of TruSecure Corporation, February 2007. `http://www.ietf.org/internet-drafts/draft-ietf-hip-esp-05.txt`, work in progress.

[9] Stephen Kent and Randall Atkinson. *RFC 2401: Security Architecture for the Internet Protocol.* IETF, BBN Corporation, @Home Network, November 1998. `http://www.ietf.org/rfc/rfc2401.txt`.

[10] Stephen Kent and Randall Atkinson. *RFC 2402: IP Authentication Header.* IETF, BBN Corporation, @Home Network, November 1998. `http://www.ietf.org/rfc/rfc2402.txt`.

[11] Stephen Kent and Randall Atkinson. *RFC 2406: IP Encapsulating Security Payload (ESP).* IETF, BBN Corporation, @Home Network, November 1998. `http://www.ietf.org/rfc/rfc2406.txt`.

[12] Miika Komu, Simon Schuetz, Martin Stiemerling, Lars Eggert, and Abhinav Pathak. *Internet-Draft: HIP Extensions for the Traversal of Network Address Translators.* IETF, Helsinki Institute for Information Technology, NEC Network Laboratories, Nokia Research Center, March 2007. `http://www.ietf.org/internet-drafts/draft-ietf-hip-nat-traversal-01.txt`, work in progress.

[13] Mika Kousa, Miika Komu, Kristian Slavov, Catharina Candolin, and Janne Lundberg. *Host Identity Protocol for Linux.* Helsinki Institute for Information Technology, Helsinki University of Technology: Telecommunications Software and Multimedia Laboratory.

[14] Julien Laganier and Lars Eggert. *Internet-Draft: Host Identity Protocol (HIP) Rendezvous Extension.* IETF, DoCoMo Communications Laboratories Europe GmbH, NEC Network Laboratories, June 2006. `http://tools.ietf.org/wg/hip/draft-ietf-hip-rvs/draft-ietf-hip-rvs-05.txt`, work in progress.

[15] Robert Moskowitz and Pekka Nikander. *RFC 4423: Host Identity Protocol (HIP) Architecture.* IETF, ICSAlabs - a Division of TruSecure Corporation, Ericsson Research NomadicLab, August 2006. `http://www.ietf.org/rfc/rfc4423.txt`.

[16] Robert Moskowitz, Pekka Nikander, and Petri Jokela. *Internet-Draft: Host Identity Protocol.* IETF, ICSAlabs - a Division of TruSecure Corporation, Ericsson Research NomadicLab, The Boeing Company, June 2006. `http://hip4inter.net/documentation/drafts/draft-ietf-hip-base-06.txt`, work in progress.

[17] Pekka Nikander and Jan Melen. *Internet-Draft: A Bound End-to-End Tunnel (BEET) mode for ESP.* IETF, Ericsson Research NomadicLab, February 2007. `http://www.ietf.org/internet-drafts/draft-nikander-esp-beet-mode-07.txt`, work in progress.

[18] Jonathan Rosenberg. *Internet-Draft: Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols.* IETF, Cisco Systems, March 2007. `http://www.ietf.org/internet-drafts/draft-ietf-mmusic-ice-15.txt`, work in progress.

[19] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Dan Wing. *Internet-Draft: Session Traversal Utilities for (NAT) (STUN).* IETF, Cisco, Microsoft, Plantronics, Cisco Systems, March 2007. `http://www.ietf.org/internet-drafts/draft-ietf-behave-rfc3489bis-06.txt`, work in progress.

[20] Jonathan Rosenberg, Rohan Mahy, and Christian Huitema. *Internet-Draft: Obtaining Relay Addresses from Simple Traversal Underneath NAT (STUN).* IETF, Cisco Systems, Plantronics, Microsoft, March 2007. `http:`

//www.ietf.org/internet-drafts/draft-ietf-behave-turn-03.txt, work in progress.

[21] Jonathan Rosenberg and Henning Schulzrinne. *RFC 3264: An Offer/Answer Model with the Session Description Protocol (SDP).* IETF, Dynamicsoft, Columbia University, June 2002. http://www.ietf.org/rfc/rfc3264.txt.

[22] Jonathan Rosenberg, Joel Weinberger, Christian Huitema, and Rohan Mahy. *RFC 3489: STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs).* IETF, dynamicsoft, Microsoft Corporation, Cisco Systems, March 2003. ftp://ftp.rfc-editor.org/in-notes/rfc3489.txt.

[23] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. *RFC 3550: RTP: A Transport Protocol for Real-Time Applications.* IETF, Department of Computer Science Columbia University, Packet Design, Blue Coat Systems Inc., July 2003. http://www.ietf.org/rfc/rfc3550.txt.

[24] Pyda Srisuresh and Kjeld Borch Egevang. *RFC 3022: Traditional IP Network Address Translator (Traditional NAT).* IETF, Jasmine Networks Inc., Intel Denmark ApS, January 2001. http://www.ietf.org/rfc/rfc3022.txt.

[25] Pyda Srisuresh, Bryan Ford, and Dan Kegel. *Internet-Draft: State of Peer-to-Peer(P2P) Communication Across Network Address Translators(NATs).* IETF, Consultant, Massachusetts Institute of Technology, Kegel.com, October 2006. http://www.ietf.org/internet-drafts/draft-ietf-behave-p2p-state-00.txt, work in progress.

[26] Pyda Srisuresh and Matt Holdrege. *RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations.* IETF, Lucent Technologies, August 1999. http://www.ietf.org/rfc/rfc2663.txt.

[27] Usagi project - linux IPv6 development project. http://www.linux-ipv6.org/.

[28] Xstunt library(stunt library in c/c++). http://www.cis.nctu.edu.tw/~gis87577/xDreaming/XSTUNT/index.html.

Appendix A

Behavioral Requirements for Future Network Address Translators

Different NAT device implementations vary widely in terms of how they handle different network layer protocols and Endpoint Mapping and Filtering capabilities. Therefore, it is very difficult to predict how a new protocol, such as HIP, will behave on a network containing a NAT. This introduces extra complexity to protocol design.

According to Audet and Jennings [2], discussions with many of the major NAT vendors have made it clear that the vendors would prefer to deploy NATs that were deterministic and caused the least harm to applications while still meeting the requirements that caused their customers to deploy NATs in the first place. The problem NAT vendors have faced is that they have had no NAT standards to comply with.

To solve the problems this high diversity in the behavior of NATs introduces, Audet and Jennings have defined a set of common terminology for describing the behavior of NATs and produced a set of requirements on behaviors for NATs in RFC 4787. There is a total of 14 requirements and they apply to traditional NATs. In this Section, we go through all of these 14 requirements but have focus on the most HIP related requirements. The reader is encouraged to consult RFC 4787 for a detailed description of these requirements and for the justification behind these requirements. We present some of the requirements in a simplified form herein.

A NAT device performing Endpoint Independent Mapping reuses the port mapping for subsequent packets sent from the same internal IP address and port to any external IP address and port. This property enables for example peer-to-peer applications and UDP hole punching technique to work correctly. The first requirement is:

REQ-1 A NAT must have an *Endpoint-Independent Mapping* behavior.

Some NATs are capable of assigning IP addresses from a pool of IP addresses on the external side of the NAT, as opposed to just a single IP address. This is

especially common with larger NATs. Some NATs use the external IP address mapping in an arbitrary fashion: one internal IP address could have multiple external IP address mappings active at the same time for different sessions. This IP address pooling behavior is called *Arbitrary*.

Other NATs use the same external IP address mapping for all sessions associated with the same internal IP address. This IP address pooling behavior is called *Paired*. NATs that use an IP address pooling behavior of Arbitrary can cause issues for applications that use multiple ports from the same endpoint.

REQ-2 It is recommended that a NAT has an IP address pooling behavior of *Paired*. Note that this requirement is not applicable to NATs that do not support IP address pooling.

Some NATs employ *Port Preservation*, which means that they attempt to preserve the port number used internally when assigning a mapping to an external IP address and port. Using Figure A.1 as a reference, Port Preservation means that port *x1* equals port *x1'* and that port *x2* equals port *x2'*. In case of port collision, these NATs attempt a variety of techniques for coping. Some NATs, however, use *Port Overloading*, i.e., they always use port preservation even in the case of collision (i.e., *X1'* = *X2'* and *x1* = *x2* = *x1'* = *x2'*).

REQ-3 A NAT must not have a *Port Assignment* behavior of *Port Overloading*.

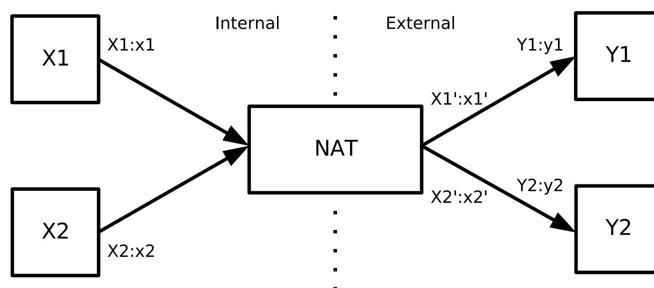

Figure A.1: Port assignment behavior

Some NATs preserve the parity of the UDP port. That is, an even port will be mapped to an even port, and an odd port will be mapped to an odd port. This behavior respects the port parity used in Real-time Transport Protocol (RTP) and Real-time Transport Protocol Control Protocol (RTCP) [23].

REQ-4 It is recommended that a NAT has a *Port parity preservation* behavior of *Yes*.

The mapping timer is defined as the time a mapping will stay active without packets traversing the NAT. There is great variation in the values used by different NATs.

REQ-5 A NAT UDP mapping timer must not expire in less than two minutes, except for specific destination ports in the well-known port range (ports 0-1023).

While most NATs refresh the mapping timer when packets traverse the NAT from the internal side of the NAT to the external side of the NAT, few NATs also refresh the mapping timer when packets arrive from the external side of the NAT. These mapping behaviors are referred to as *NAT Outbound Refresh Behavior* and *NAT Inbound Refresh Behavior* respectively.

REQ-6 The NAT mapping Refresh Direction must have a *NAT Outbound refresh behavior* of *True*. The NAT mapping Refresh Direction may have a *NAT Inbound refresh behavior* of *True*.

Dynamically configured NATs should operate as follows.

REQ-7 A NAT device whose external IP interface can be configured dynamically must either (1) automatically ensure that its internal network uses IP addresses that do not conflict with its external network, or (2) be able to translate and forward traffic between all internal nodes and all external nodes whose IP addresses numerically conflict with the internal network.

Endpoint Independent Filtering is the most liberal form of filtering and Endpoint Dependent Filtering is the least liberal form of filtering.

REQ-8 If application transparency is most important, it is recommended that a NAT has an *Endpoint Independent Filtering* behavior. If a more stringent filtering behavior is most important, it is recommended that a NAT has an *Address Dependent Filtering* behavior.

Hairpinning allows two endpoints on the internal side of the same NAT to communicate even when they only use each other's external IP addresses and ports. Hairpinning is illustrated in Figure A.2. The NAT may translate the hairpinned packet to either an internal (*X1:x1*) or an external (*X1':x1'*) source IP address and port. Therefore, the NAT hairpinning behavior can be either *External source IP address and port* or *Internal source IP address and port*. Internal source IP address and port may cause problems by confusing implementations that expect an external IP address and port.

REQ-9 A NAT must support *Hairpinning*. A NAT Hairpinning behavior must be *External source IP address and port*.

The next requirement is about ALGs.

REQ-10 To eliminate interference with UNilateral Self-Address Fixing (UNSAF) NAT traversal mechanisms and allow integrity protection of UDP communications, NAT ALGs for UDP-based protocols should be turned off.

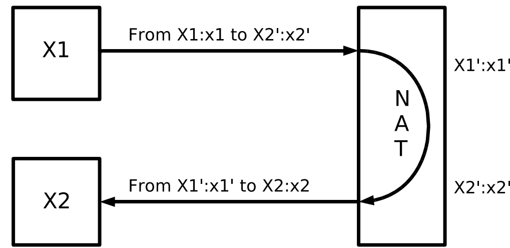

Figure A.2: Hairpinning behavior

While most NATs behave deterministically, some NATs will exhibit different behaviors under some conditions. This fact further complicates the classification of NATs. For example, the NAT translation and the filtering behavior may be indeterministic when the NAT employs port preservation or has specific algorithms for selecting a port other than the next free one.

REQ-11 A NAT must have deterministic behavior, i.e., it must not change the NAT translation or the filtering behavior at any point in time, or under any particular conditions.

When a NAT sends an ICMP query to a host on the other side of the NAT, the peer may send an ICMP in response. The ICMP response may be sent by the destination host or by any router along the network path. Although NATs allow outbound ICMP query type messages, some NATs block inbound ICMP messages. This kind of behavior is undesired, because it can interfere with application failover and traceroute among other things.

REQ-12 Receipt of any sort of ICMP message must not terminate the NAT mapping.

The last two requirements deal with packet fragmentation.

REQ-13 If the packet received on an internal IP address has DF=1 (Don't Fragment bit set to 1), the NAT must send back an ICMP message *Fragmentation needed and DF set* to the host.

REQ-14 A NAT must support receiving in-order and out-of-order fragments, so it must have *Received Fragment Out of Order* behavior.