NAT TRAVERSAL THROUGH TUNNELING

by

Arun Madhavan

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 1 0

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: _____

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

_____ _____

Beichuan Zhang — Date

Assistant Professor, Department of Computer Science

TABLE OF CONTENTS

LIST OF FIGURES **6**
ABSTRACT **8**
CHAPTER 1 Network Address Translation **9**
1.1 Types of NAT 10
CHAPTER 2 Introduction **13**
CHAPTER 3 NAT traversal **15**
3.1 The NAT traversal problem 15
3.2 Port forwarding 16
3.3 UPnP IGD and NAT-PMP 17
3.4 Hole punching 19
3.5 STUN 22
3.6 TURN 23
CHAPTER 4 NAT Traversal by Tunneling **24**
4.1 Basic tunneling scheme 24
4.2 Learning NAT, server addresses 25
4.3 Traversing client-side NATs 26
4.4 Traversing nested server-side NATs 27
4.5 Traversing legacy NATs 29
4.6 Stateless operation of NATTT 30
4.7 Deployment considerations 31
CHAPTER 5 Implementation **33**
5.1 NATTT client daemon 33
5.1.1 Architecture of the client daemon 33
5.1.2 Receiving traffic on the NATTT daemon 34
5.2 NATTT server daemon 35
5.2.1 Running NATTT server daemon on a NAT 36
5.2.2 Running NATTT server daemon on a host 37
5.2.3 Architecture of the NATTT server daemon 38
5.3 Overview of the entire system 38
5.4 Implementation of NATTT on a NAT 38
5.5 Detecting a single NAT's external address 40
CHAPTER 6 Future work **41**
6.1 Automatic NAT address detection 41
6.2 Nested NAT traversal 41
6.3 NAT Traversal with an unmodified client host 42

TABLE OF CONTENTS — *Continued*

REFERENCES **43**
APPENDIX A Code structure **45**
A.1 nat3d 45
A.1.1 Methods 45
A.1.2 Member Variables 48
A.2 LruCache 48
A.2.1 Methods 49
A.2.2 Member Variables 49
A.3 TunnelMgr 50
A.3.1 Methods 51
A.3.2 Member Variables 52
A.4 win_tun_mgr 53
A.4.1 Methods 55
A.4.2 Member Variables 56
A.5 linux_tun_mgr 56
A.5.1 Methods 57
A.6 TunnelEntry 57
A.6.1 Methods 59
A.6.2 Member Variables 59
A.7 tun_hdr_t 59
A.7.1 Member Variables 61
A.8 PcapArpHandler 61
A.8.1 Methods 62
A.8.2 Member Variables 62
A.9 p_thread 63
A.9.1 Methods 63
A.10 log 65
A.10.1 Methods 65
A.10.2 Member Variables 65
A.11 functions 66
A.11.1 Methods 66
A.12 UPnP 66
A.12.1 Methods 67
A.12.2 Member Variables 67
A.13 DnsResolver 67
A.13.1 Methods 69
A.13.2 Member Variables 70

TABLE OF CONTENTS — *Continued*

A.14 DnsQuery 70
A.14.1 Methods 72
A.14.2 Member Variables 72
A.15 DnsPacket 73
A.15.1 Methods 74
A.15.2 Member Variables 74
A.16 DnsHeader 75
A.16.1 Methods 76
A.16.2 Member Variables 76
A.17 DnsName 77
A.17.1 Methods 77
A.17.2 Member Variables 78
A.18 DnsRR 78
A.18.1 Methods 78
A.18.2 Member Variables 80
A.19 DnsA 81
A.19.1 Methods 81
A.20 DnsCompression 82
A.20.1 Methods 82
A.20.2 Member Variables 82

LIST OF FIGURES

1.1 Network Address Translation 11
3.1 Port forwarding 17
3.2 Hole punching 21
4.1 Traversing a single NAT 26
4.2 Learning NAT, private host addresses 27
4.3 Tunneling through a client side NAT 28
4.4 Nested NAT traversal 28
4.5 Deployment of NATTT with a legacy NAT 29
4.6 Stateless implementation of NATTT 31
5.1 NATTT client daemon 34
5.2 Illustration of a TUN device 35
5.3 Basic tunneling - Actual implementation 36
5.4 NATTT server daemon on a host 39
5.5 Overview of the entire system 40
A.1 UML diagram of the NATTT daemon 46
A.2 Component diagram: NATTT daemon 47
A.3 Component diagram: LruCache 48
A.4 Component diagram: TunnelMgr 50
A.5 Component diagram: win_tun_mgr 54
A.6 Component diagram: linux_tun_mgr 56
A.7 Component diagram: TunnelEntry 58
A.8 Component diagram: tun_hdr_t 60
A.9 Component diagram: PcapArpHandler 61
A.10 Component diagram: p_thread 63
A.11 Component diagram: log 64
A.12 Component diagram: functions 65
A.13 Component diagram: UPnP 66
A.14 Component diagram: DnsResolver 68
A.15 Component diagram: dns_query 71
A.16 Component diagram: DnsPacket 73
A.17 Component diagram: DnsHeader 75
A.18 Component diagram: DnsName 77
A.19 Component diagram: DnsRR 79

LIST OF FIGURES — *Continued*

A.20 Component diagram: DnsA 81
A.21 Component diagram: DnsCompression 82

ABSTRACT

Network Address Translation (NAT) is widely prevalent solution adopted to alleviate the IPv4 address exhaustion problem. Due to the use of private IP addresses on hosts behind the NAT, it is not possible for external hosts to initiate communication with these hosts. This poses a hurdle to many emerging applications, such as VoIP and P2P. Although a plethora of NAT traversal solutions have been proposed in recent years, they suffer from being application-specific, complex, or requiring some behavioral compliance from the NAT.

The work presents an simple technique that is generic, works with nested NATs, is incrementally deployable and only expects minimalistic common behavior across all NAT implementations. The design includes the use of UDP tunnels and a sequence of NAT addresses and private IP addresses to uniquely identify a host. Simple and incrementally deployable changes are proposed to DNS to learn the addresses.

CHAPTER 1

Network Address Translation

Network Address Translation is a solution developed to alleviate the exhaustion of the IPv4 address space by allowing the use of private IP addresses on home and corporate networks behind routers, with a single public IP address facing the public Internet. Network address translation breaks end-to-end connectivity since the NAT has no automatic method of determining the internal host for which incoming packets are destined. Furthermore, since it is possible to have multiple levels of NAT between a host and the Internet, there is no one unique address that identifies a host.

Figure 1.1 provides an illustration of a typical NAT setup. Alice, Bob and Carol are three hosts with private IPs behind a NAT which has a single public IP 'NAT'. Since the only public IP in this setup is 'NAT', there are two implications: (1) When these private hosts send a packet to the Internet, the NAT replaces the private source address of the packet with its public IP 'NAT' since it is not possible to send a packet having a private address on the Internet. (2) Incoming packets from the Internet are always destined to 'NAT'. Therefore, the NAT uses the destination port number in the incoming packet to identify which private host this incoming packet should be sent to. This scheme is described below.

In step (1) of the figure, Bob sends a packet from Bob:100 to Dave:80. As this packet passes through the NAT, the NAT replaces the private source IP with

its public IP 'NAT'. It also replaces the source port number 100 with port number 6000 (a free port on the NAT), and records a mapping from port 6000 to the origin of the packet i.e. Bob:100. Any incoming packets on port 6000 are forwarded to Bob:100.

In step (2), Dave:80 responds to the packet from NAT:6000, which is originally a packet from Bob:100. When the packet arrives on NAT:6000, the NAT uses the mapping created in step (1) to forward this packet to the correct private host i.e. Bob:100.

This mapping scheme allows the host behind the NAT to initiate the connection to a host in the Internet. However, if a host in the public Internet wants to initiate a connection to a host behind a NAT, it is not possible since the NAT has no automatic way of determining which private host the packet is destined for. This is the biggest problem associated with NATs and is termed the 'NAT traversal problem'. The problem and some solutions to the problem are described in detail in Section 3: NAT traversal.

1.1 Types of NAT

The major classification criteria with NATs is the nature of the mapping function from a port to private host. The NAT classifications presented below only serve as a guideline, since attempts to classify have failed and the problem has proven to be intractable.

Full Cone: A full cone NAT is one where all requests from the same internal IP address and port are mapped to the same external IP address and port. Furthermore, any external host can send a packet to the internal host, by sending a packet to the mapped external address. Techniques to traverse full cone NATs have been

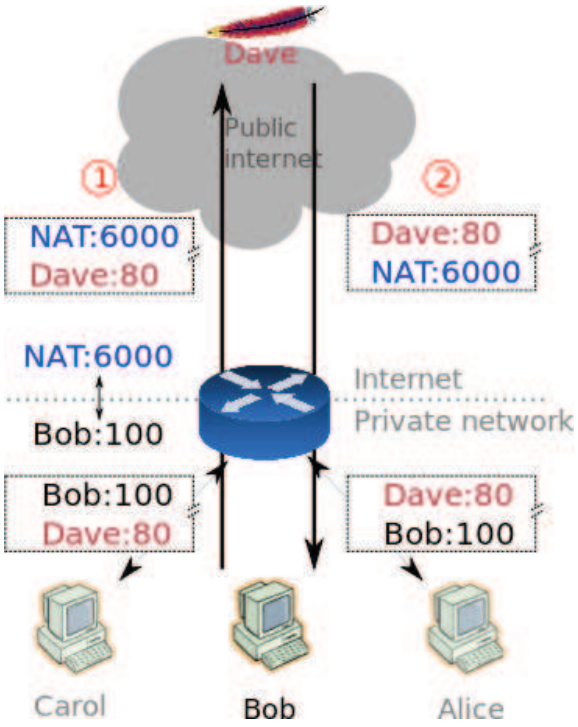

Figure 1.1: Network Address Translation

quite successful.

Restricted Cone: A restricted cone NAT is one where all requests from the same internal IP address and port are mapped to the same external IP address and port. Unlike a full cone NAT, an external host (with IP address X) can send a packet to the internal host only if the internal host had previously sent a packet to IP address X. Techniques to traverse restricted cone NATs have been successful, albeit less successful than techniques for full cone NATs

Port Restricted Cone: A port restricted cone NAT is like a restricted cone NAT, but the restriction includes port numbers. Specifically, an external host can send a packet, with source IP address X and source port P, to the internal host only if the internal host had previously sent a packet to IP address X and port P. Techniques

to traverse restricted cone NATs have been successful, albeit less successful than techniques for full cone NATs

Symmetric: A symmetric NAT is one where requests from the same internal IP address and port are mapped to a different external IP address and port, for each destination IP address and port. Furthermore, only the external host that receives a packet can send a packet back to the internal host. Traversing symmetric NATs has proven to be the more challenging than any other type of NAT.

CHAPTER 2

Introduction

NAT Traversal by Tunneling (NATTT) provides a NAT traversal solution that is generic, (supporting all applications and transport protocols), supports nested NATs and is incrementally deployable.

Suppose an external host A wants to initiate communication with an internal host B behind a NAT Y. If A knows both Y's public address (Y_{pub}) and B's private address, A can tunnel packets to B as follows. The outer header of a packet is destined to Y_{pub}, so that the packet can be routed over the public Internet to reach Y; The inner header is addressed to B, so that when Y receives the packet, it can remove the outer header and find out where to forward the packet within the private network.

To implement this idea, there are 3 main design questions that must be answered:

1. How does the initiator of the communication learn the tunnel end-point(s) at the destination network?

2. The hosts behind a NAT suffer from the non-uniqueness problem of their NAT addresses. That is, even if a host A can reach other hosts behind NATs, these other hosts may be behind different NATs and happen to have the same private IP addresses, how can A distinguish different remote hosts when they

use the same (private) IP addresses?

3. How can this design handle the current NAT traffic, and work with the existing NATs?

This solution uses a new type of DNS record to learn the tunnel end-point addresses. The solution is implemented as a daemon running on the client host, as well on the NAT or on one host behind the NAT. The daemon uses UDP tunnels to encapsulate packets, which can then traverse deployed NATs that are capable of decapsulating the packet and forwarding them to the destination host in the NAT's private network.

CHAPTER 3

NAT traversal

3.1 The NAT traversal problem

Section 1: Network Address Translation discussed how NATs function, and how they masquerade an entire private network behind a single public IP address. This section briefly explains the the primary problem associated with such a setup and discusses some solutions to this problem.

The problem with Network Address Translation is that it is possible to deliver packets to a private host only if there exists a mapping from a port number on the NAT. And such a mapping is created when an outbound packet from a private host passes through the NAT. This effectively means that it is only possible for hosts behind a NAT to initiate a connection to a host in the Internet. Hosts on the Internet can not initiate a connection to hosts behind the NAT unless there is a way to tell the NAT which private host the packet is destined for (recall that packets to private hosts behind a NAT are addressed to the NAT, since the NAT is the only entity with a public IP. The NAT uses port numbers to determine the destination private host).

NAT traversal refers to techniques that establish and maintain TCP/IP or UDP connections traversing NAT devices. NAT traversal is an important problem since services that accept inbound connections (such as HTTP, SMTP, VoIP, P2P

applications etc) may be located on hosts behind a NAT and must be accessible from the public Internet.

Many techniques exist, but no single technique works in every situation since NAT behavior is not standardized, and very little common behavior can be expected across NATs. There are also different types of NATs, as described in section 1.1. Therefore, for a design to be successful, it is important to treat the NATs as black boxes, and expect very minimalistic common behavior from them. The problem is compounded by the fact that many NATs are designed to be invisible to both sides of a connection.

Some of the most common techniques used in NAT traversal are described in the forthcoming sections.

3.2 Port forwarding

Port forwarding is the simplest NAT traversal technique. Port forwarding involves setting up an explicit, static mapping rule from a port on the NAT to a port on a private host behind the NAT.

Port forwarding is illustrated in figure 3.1. In the figure, a port forwarding rule is set up to map packets arriving on NAT:8080 to Alice:80. Therefore, a packet arriving from Dave:port on NAT:8080 is forwarded to Alice:80.

Discussion of port forwarding:

The port forwarding technique has several advantages: (1) It is typically available on all types of NATs (2) Legacy clients can connect to a private host behind a NAT. i.e. the clients need not run any special NAT traversal technique/require any modification of any sort.

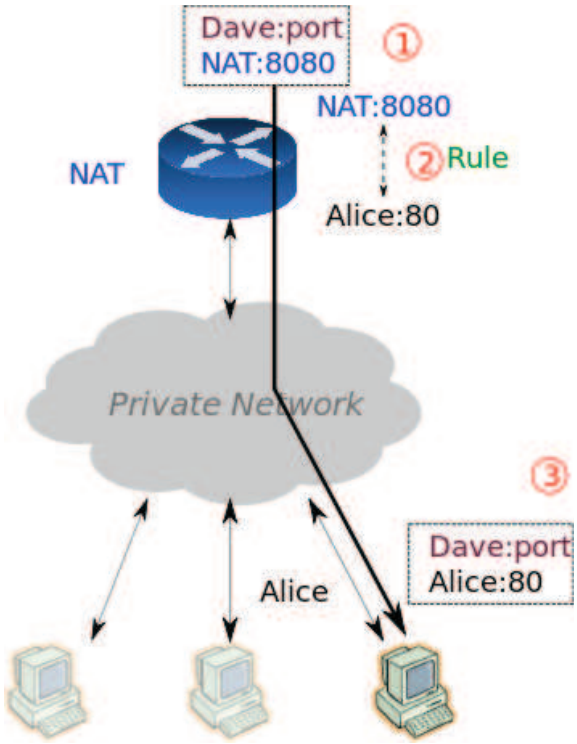

Figure 3.1: Port forwarding

The disadvantages of port forwarding are: (1) It is a static rule, and if the IP address of the host changes, the mapping rule needs to be changed to reflect that. (2) It is not possible to forward the same port to a second private host. If a user ran two ssh servers behind a NAT, he would need to forward different ports to the ssh servers (Although the ssh servers may themselves run on the same port).

3.3 UPnP IGD and NAT-PMP

The big problem with port forwarding is that the forwarding/translation rule is static. UPnP and PMP let a host 'talk' to a NAT and set up forwarding rules as required.

UPnP (Universal Plug and Play) is an architecture for pervasive peer-to-peer network connectivity of PCs and other devices or appliances. It enable these devices to automatically connect with one another and work together.

One of the sub-protocols in the UPnP suite is Internet Gateway Device (IGD) Standardized Device Control Protocol, which allows hosts to interact with Internet gateway devices, such as NATs in our case. This allows a host to forward ports automatically by interacting with NATs. Also included in the protocol is the ability for a host to learn the public (external) IP address of the gateway device.

NAT-PMP (NAT Port Mapping Protocol) was proposed by Apple computer as an elegant alternative to the cluttered and complicated IGD protocol. NAT-PMP is implemented on all Apple home routers (NAT devices) and as well on Microsoft Windows and Apple OS X. It is also supported by many applications that require to configure port forwarding.

NAT-PMP is the most popular implementation of a set of techniques called zeroconf (Zero configuration networking). Other implementations by Microsoft and some open source implementations for Linux exist.

Discussion of UPnP and NAT-PMP:

The major advantage of these methods is that port forwarding is automated and applications can forward ports dynamically as required.

Some of the disadvantages with these methods are: (1) As with port forwarding, it is not possible for two hosts to forward the same ports. (2) Applications need to be modified to be UPnP aware. (3) Not all NATs implement UPnP or the zeroconf protocols. UPnP enabled NAT devices were not prevalent until a few years ago. (4) Nested NAT traversal is undefined. It requires all the nested NATs to be able to accept arbitrary port forwarding requests, something that is not likely to be

work, especially if the outermost NAT is an ISP's NAT.

3.4 Hole punching

Hole punching is a very widely used NAT traversal technique where two hosts, both behind NATs, establish connections to each other with the help of a public server. Both private hosts send a registration packet to the public server. When the packet passes through the NAT, a mapping is created from a free port X on the NAT to the private host's address (as described in section 1). Therefore, any packets that arrive on port X on the NAT are forwarded to the private host. Then, the two hosts learn about the mapped ports on each other's NATs from the registration server, and establish connections to each other on the mapped ports.

In figure 3.2, clients Alice, Bob behind publicly addressable NATs register and maintain maintain a *traversal session* with a public registration server S. To register for a session, clients provide an "internal endpoint", which is combination of their private IP address and port used for the traversal session. As this packet passes through the NAT, the NAT maps the packet's source address and port to a free port on the NAT. When this packet arrives on S, it records the internal endpoint provided by the client, and additionally records an "external endpoint", which consists of the registration datagram's source address and port (i.e. the NAT's public IP and mapped port).

In the figure, NAT1 and NAT2 are assumed to be restricted cone NATs. In this type of NAT, a packet sent by a host H to any mapped port P on a NAT is forwarded to the private host only if the private host had first sent a packet to H, and the NAT had mapped the packet's source address to P. In other words, the NAT's mapping $NAT{:}P \leftrightarrow$ *Private address:port* is only valid for the session $NAT{:}P \leftrightarrow H$.

In figure 3.2(a), In step (1) Alice retrieves Bob's external and internal endpoints from S. In step (2) S simultaneously sends Bob the external and internal endpoints of Alice, and instructs it to try connecting to both endpoints of Alice.

In figure 3.2(b), the hosts now have each other's endpoints. The hosts send packets to each other on both internal and external endpoints, utilizing the same socket used to maintain the session with the server.

In the figure step (1), (2) signify that packets sent by the hosts to internal endpoints do reach other since the NATs are themselves not behind a NAT.

In the figure, step (3) signifies the packet from Alice reaching Bob's external end-point first. Since the packet passes through NAT1, a session NAT1:4444 $\leftrightarrow$ NAT2 is established for the mapping on NAT1:4444. The packet is refused at NAT2 as the session NAT1 $\leftrightarrow$ NAT2:6666 is not established for the mapping on NAT2:6666. Step (4) signifies that the packet from Bob via NAT2 arriving from NAT2:6666 (i.e. Bob:3333) on NAT1:4444 is forwarded to the private host (i.e. Alice:2222) since NAT1 is aware of the session. Also, since the packet from Bob has passed through NAT2, the session is set up on NAT2. Future packets between Alice and Bob are able to traverse both NATs and reach their respective destinations.

The transport protocol used for hole punching is typically UDP. TCP is also used, but less successfully.

Discussion of hole punching:

Some of the disadvantages with hole punching are: (1) Hole punching requires a lot of behavioral compliance from the NAT, especially with respect to how the NAT maps ports. Such behavior is non-standard amongst NATs. (2) When both NATs are behind a NAT themselves, hole punching is not well-defined and requires

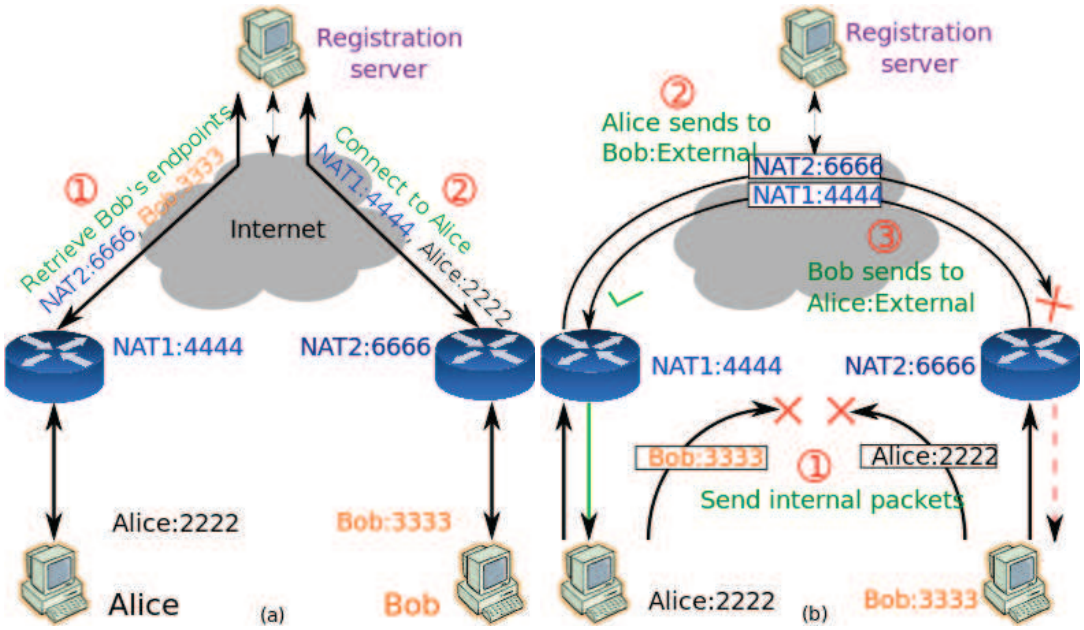

Figure 3.2: Hole punching

the outermost NAT to support specific routing features. (3) A publicly addressable external server is required. (4) It is required to be built into the application.

Some of the advantages of hole punching are: (1) The port mapping is dynamic and suits applications like VoIP. (2) The NAT's explicit cooperation is not required. There is no need to configure the NAT with a mapping rule etc.

3.5 STUN

STUN was an acronym for *Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators (NATs)*. It was originally a technique to classify the type of NAT automatically and use a standardized hole punching technique to traverse the NATs. Eventually, the methods used in the original STUN protocol became intractable, given the plethora of NAT implementations. The original set of methods was updated, and a new RFC issued. STUN now stands for *Session Traversal Utilities for NAT*

The new STUN is a client-server protocol, requiring an external server with two public IP addresses. The main purpose of the STUN protocol is to enable an application running on a host behind a NAT to discover the port translation done by the NAT; i.e. which port other devices can use to connect to it from outside the network. Armed with this information, applications can use a form of hole punching.

Discussion of STUN:

Since STUN is not a traversal mechanism in itself, and uses the hole punching technique, STUN has the same advantages and drawbacks as hole punching.

3.6 TURN

TURN (Traversal Using Relay NAT), is a technique to traverse NATs by using a third-party relay server. The destination application running on a host behind the NAT establishes and maintains a session with the TURN server. Hosts that need to send data to the application send it via the TURN server. TURN is typically used as a last-resort technique for NAT traversal. TURN is currently in an Internet draft status.

Due to the high load on the public TURN server, TURN is typically used as a last resort technique in case other techniques fail.

Discussion of TURN:

The major advantage of TURN is that it is perhaps the only NAT traversal technique that is always successful!

Some of the disadvantages of TURN are: (1) The requirement for a third party (publicly accessible) relay server causing concerns regarding scalability, latency, bandwidth and processing power wastage. (2) TURN is only a means of connectivity between two known hosts; not between an arbitrary client and a server behind a NAT.

CHAPTER 4

NAT Traversal by Tunneling

4.1 Basic tunneling scheme

This section describes the basic tunneling scheme, the players involved, and how the system traverses a single NAT.

In Figure 4.1, client A on the public Internet wants to access web server B behind a NAT Y. By querying B's DNS name, A learns B's private IP address and Y's public IP address (See next section for details). A then encapsulates its packets in UDP. The outer IP/UDP header is destined to Y on a well-known UDP port number assigned for the use of NATTT (100 is used in this writing). The inner header is destined to B on its service port, which is 80 for its web service in this example. Routers in the Internet will forward the packet according to its outer header until it reaches Y, which recognizes it as a NAT tunnel packet because of the well-known destination UDP port number 100. Y removes the outer header and forwards the packet to B according to the inner header. The server B receives, processes, and sends packets as usual without any change or being aware of the NAT. When Y receives the outbound packet from B to A, it encapsulates the packet in UDP and sends it back to A. Upon receiving the packet, A can uniquely identify the packet source by using addresses of both Y and B.

This approach has two major advantages. First, the server B does not

require any modifications to be made to it at all! Second, by using the appropriate destination address in the inner header of the tunneled packet, it is possible to reach any host behind the NAT!

4.2 Learning NAT, server addresses

Figure 4.1 illustrated the working of the tunneling mechanism. This section describes how the address of the NAT and the server B are learnt.

DNS is the mechanism used to learn the addresses of the NAT and the server. The destination server is identified by a DNS Fully Qualified Domain Name (FQDN). When a DNS query is made for the FQDN, the IP address of the NAT as well as the destination server are returned. Since it is possible to store only one IP address in a standard "A" DNS record, a new type of DNS record called a "NAT" record is used to store both IP addresses. Since legacy applications are only capable of making "A" DNS requests, the NATTT daemon makes the "NAT" requests on behalf of the application. It does this by intercepting the application's DNS requests, issuing the "NAT" DNS request, and returning a single IP address to the application in place of the NAT and server addresses retrieved via the "NAT" DNS request.

In figure 4.2, The application requiring traversal (web browser, ssh) makes a DNS request for the domain name of the destination server (i.e. the private host behind the NAT). The NATTT client daemon intercepts the DNS request and forwards it to the DNS server. The DNS server responds with an "NXDOMAIN" signifying that there is no "A" record by that name. The daemon next makes a "NAT" DNS request for the same host name. The server responds with the public NAT address and (private) address of server B. Next, the daemon responds to the

applications “A” DNS request by sending it a single special IP in place of the two IP addresses received from the DNS server. The NATTT daemon then intercepts the application’s traffic on the special IP and tunnels it to B via Y. When traffic arrives from B (tunneled via Y), the daemon forwards this traffic onto the application using the special IP as source address.

A point to note is that the “NAT” DNS request is made only if there exists no “A” record for the FQDN. Therefore, responses to successful “A” requests are forwarded on to the application with no delay.

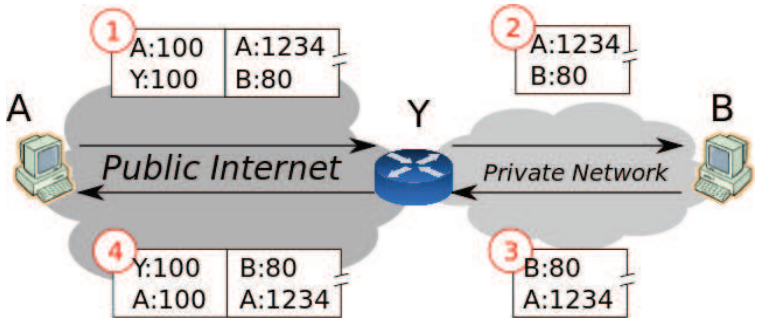

Figure 4.1: Traversing a single NAT

4.3 Traversing client-side NATs

A NAT on both ends of the communication is a complex situation for other NAT traversal techniques They require the assistance of an external server to punch holes through the NAT, set up a session with an external server (TURN, hole punching) or require the technique to be application-aware i.e. to know beforehand the transport protocol and port number the application uses (UPnP, STUN).

In contrast, NATTT only requires a small modification to be made when both ends of the communication are behind NATs.

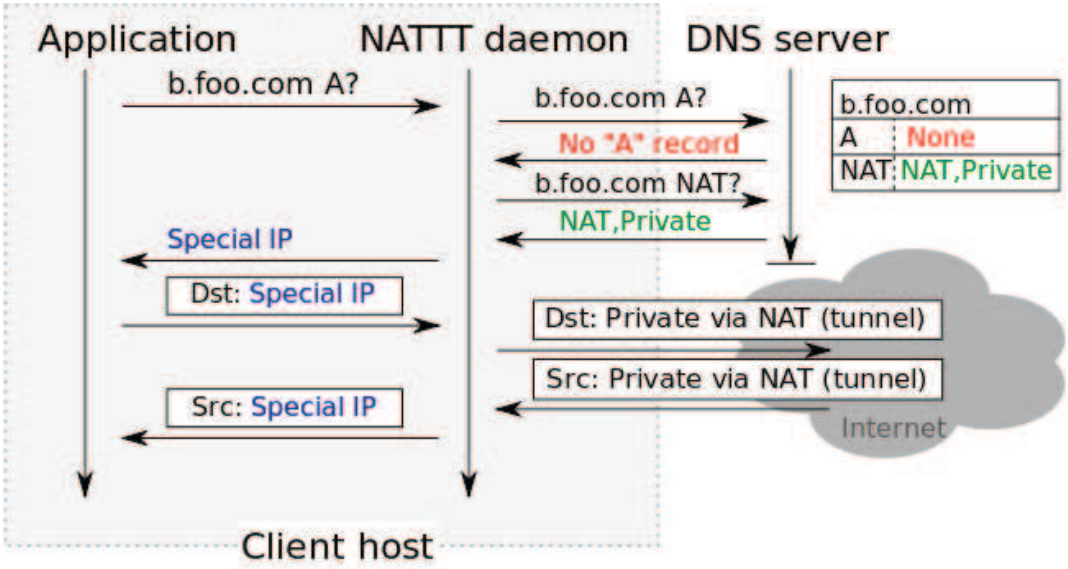

Figure 4.2: Learning NAT, private host addresses

In Figure 4.3, A is behind a NAT X which does not understand the tunneling scheme. When the tunneled packet passes through the NAT X, the NAT does traditional port mapping. Therefore, in the outer header, the source address A is replaced with the NAT's public address X, and A's source port of 100 is mapped to external port 5000 on the NAT. However, after the packet reaches B, returning packets must be addressed to A's NAT X, on port 5000. Since Y records the external source (X:5000) during packet decapsulation (2) to (3), the external source is restored when it encapsulates outbound packet (4) to (5). This enables server host B to work with the new tunneling scheme without any changes.

4.4 Traversing nested server-side NATs

Traversing nested NATs (multiple layers of NATs between a host and the Internet) is a very complicated situation for traversal techniques. Some techniques do not

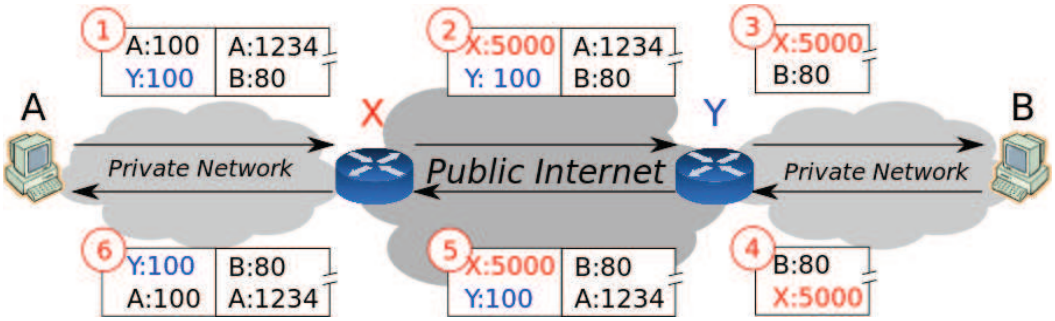

Figure 4.3: Tunneling through a client side NAT

work well with nested NATs or expect the NATs to have a knowledge of the other NATs present in the system (UPnP, STUN). In the case of NATTT, the basic encapsulation/decapsulation scheme can be extended to work with nested NATs. To traverse nested NATs, (1) the NATTT daemon needs to be installed on all nested NATs. (2) the encapsulated packet needs to have as many headers are as there are NATs.

In figure 4.4, B is behind two levels of NAT, X and Y. A learns from DNS (1) X's public IP address X_{pub}, (2) Y's NAT address in the outer NAT, and (3) B's NAT address in the second level NAT. A encapsulates its packets in two levels: the outermost header is destined to X:100, the middle header Y:100, and the innermost header B:80. The same tunneling mechanism delivers the packets.

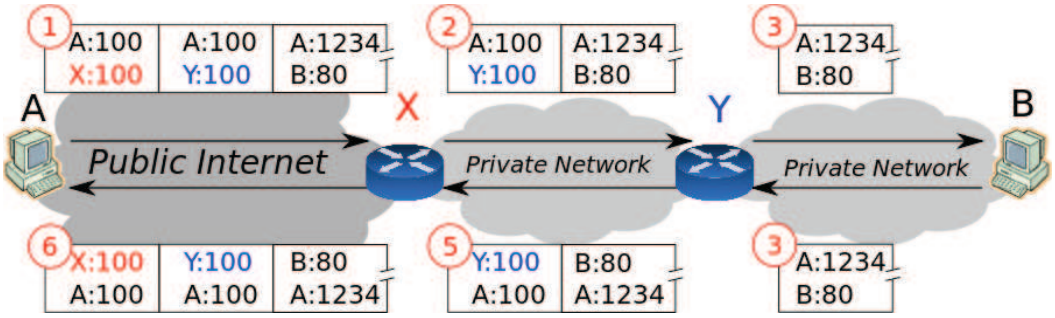

Figure 4.4: Nested NAT traversal

4.5 Traversing legacy NATs

It is also possible to deploy NATTT without having to modify the NAT i.e. using a legacy NAT. This is achieved by delegating the NATs job of encapsulating/decapsulating the packets to a host behind the NAT.

In figure 4.5, a legacy NAT X is configured to forward traffic of UDP port 100 to a host H behind X. H is installed with the NATTT daemon. H decapsulates and forwards packets to their internal destinations. One small modification needs to be made to NATTT to support operation on a host. To ensure that return traffic flows through H, the source address of the decapsulated packet is changed to 'H'. Also, source port translation is performed to ensure that there is no conflict with port numbers used by other applications. In the figure, source address A:1234 is translated to H:2000

All a user needs to do to run NATTT in this configuration are (1) install the daemon on a regular host H, and (2) configure one UDP port forwarding entry on the legacy NAT X. This scheme can also be used by enterprise networks as an incremental deployment approach.

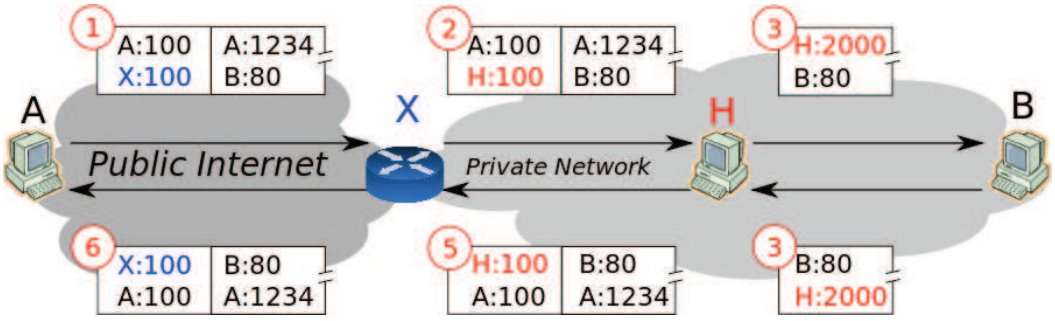

Figure 4.5: Deployment of NATTT with a legacy NAT

4.6 Stateless operation of NATTT

The Internet is designed in such a way that only the end hosts maintain state about the communication taking place between them. Stateless operation is a significant principle since it ensures that failure/reboot of a network device acting as an intermediary in the communication does not break the communication. All the intermediary Internet devices like routers function statelessly. NATs, however maintain state about the mapping of ports from their public address to private addresses. If NATTT is fully deployed (i.e. on the 2 end hosts and all intermediary NATs), it can make NAT traversal stateless. This allows load balancing and connection backup over multiple NATs.

The key fact that makes stateless NAT traversal possible is that the encapsulated packet's inner header contains the addresses of the client and the server.

Stateless operation is illustrated in figure 4.6. The figure describes how NATTT could function statelessly when tunneling through a client-side NAT.

Client host A sends an encapsulated packet to the NATTT daemon on Y:100 through NAT X. X discovers this is an NATTT packet by looking at the destination port number of 100. X does not do the traditional port mapping which would lead to X maintaining state about this connection. Instead, X replaces the private source address A (on the outer header) with its public, routable address X. Therefore, the client side of the communication is stateless.

On the destination side, Y receives the encapsulated packet. Y recognizes this as a NATTT packet by looking at the destination port number of 100. Now, Y does not decapsulate this packet (as done previously in this section). Instead, it forwards the full, encapsulated packet to the destination address present in the inner header ('B').

The packet is received and decapsulated by the NATTT daemon on B. When traffic needs to be sent from B to A, the same procedure is followed i.e. Y does not do source port translation, and X does not do packet decapsulation.

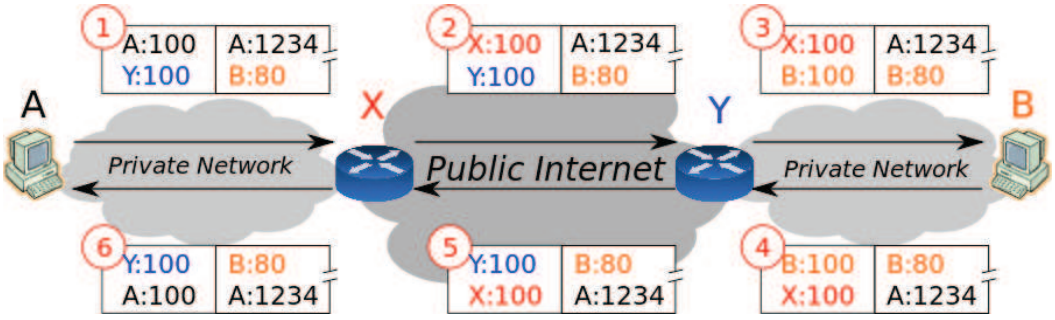

Figure 4.6: Stateless implementation of NATTT

4.7 Deployment considerations

This section talks about the ease and simplicity in the deployment of NATTT.

On the client side, NATTT only requires users to install a daemon program on the local host. There is no change to applications, operating systems, or network devices (such as NATs). Since most client users can benefit from applications enabled by NATTT, they have the incentive to deploy, but may not have the expertise to make significant changes.

On the server side, there are two options: (1) The NAT needs to be upgraded to support NATTT or (2) The NAT needs to forward port 100 to a host that runs NATTT. Further, the DNS zone file needs to include the new NAT resource record for servers behind the NAT. Hosts behind the NAT do not need any changes. This works well for enterprise networks as the system administrators can make the necessary changes at a couple of places, without the need to upgrade a large number

of server hosts.

For modifying DNS records, there are a number of free DNS services available for end users to register their server hosts. In the case that users do not have access to add a new type of resource record, they can add an additional A RR to emulate the NAT RR. For example, querying b.foo.com returns B's private IP, querying b-nat.foo.com returns B's NAT device, X's public IP, and the daemon can be configured to accommodate this setup.

CHAPTER 5

Implementation

This section describes the implementation details of the NATTT daemon. The NATTT daemon can be configured to operate either in the client mode or in the server mode. It consists of 3 major components:

1. **Tunnel manager:** Handles encapsulation, decapsulation and forwarding of packets.
2. **DNS handler *(client daemon only)*:** Intercepts and handles applications' DNS requests. Does the 'NAT' DNS query on behalf of the application and returns a single IP address in place of the NAT and private host addresses.
3. **ARP handler *(server daemon only)*:** Required to retrieve response traffic from other servers behind the NAT.

5.1 NATTT client daemon

5.1.1 Architecture of the client daemon

The client daemon is illustrated in figure 5.1. The client daemon intercepts the applications DNS request for FQDN, issues the 'NAT' DNS query on the absence of an 'A' record for the FQDN and returns a single special IP address to the application. The daemon then intercepts traffic on the special IP and tunnels the data

to the destination server behind the NAT. Upon reception of a encapsulated packet destined for an application on the local host, the daemon decapsulates and forwards the packet to the application.

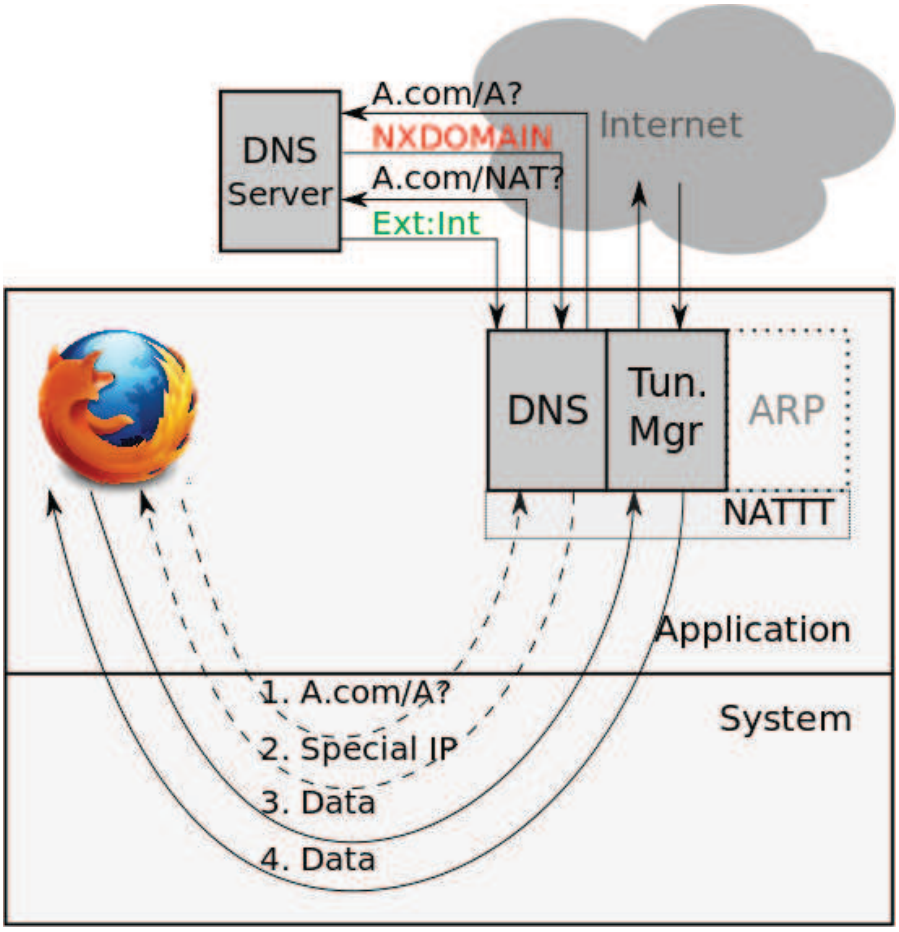

Figure 5.1: NATTT client daemon

5.1.2 Receiving traffic on the NATTT daemon

This section deals with how traffic is intercepted on the 'special IP' (mentioned in section 5.1).

NATTT uses a virtual network adapter called a “TUN” network adapter to receive traffic on an entire network subnet. The ‘special IP’ referred to in section 5.1 is any IP from that network subnet.

The TUN adapter works as follows: It is first assigned a network subnet. It then manipulates the system routing tables so that the traffic destined to that subnet is sent to the TUN adapter. When it receives such traffic, it forwards the traffic to the NATTT daemon. The traffic may be sent by applications on the local machine or arrive on other network adapters.

The TUN network adapter is illustrated in Figure 5.2. It is assigned a network subnet of 10.1.1.0/24. Therefore, Any packets arriving on eth0 or eth1 destined for an IP in 10.1.1.0/24 are routed via the TUN adapter tun0 to the daemon.

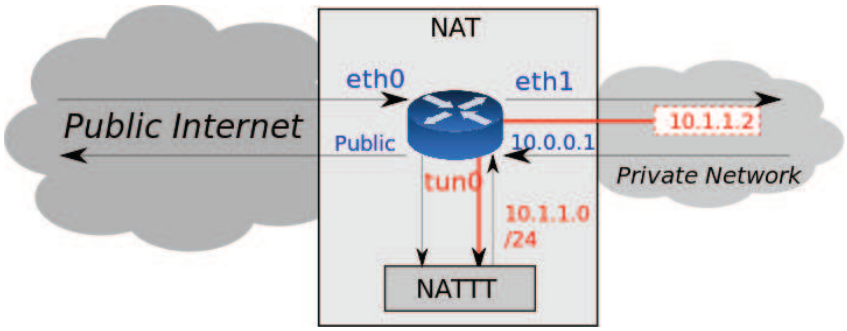

Figure 5.2: Illustration of a TUN device

5.2 NATTT server daemon

The implementation of the NATTT daemon is slightly different from the scenarios described in section 4: NAT Traversal by Tunneling. This section illustrates the differences and the consequences it has.

The difference is that after the NATTT server daemon decapsulates the incoming packet, it replaces the source IP address of the packet (i.e. the inner header) with an address from the TUN subnet. This is done so that the return packets from the server are addressed to the TUN subnet IP and hence, received by the NATTT daemon.

This difference is illustrated in figure 5.3. The NATTT daemon translates the source address of the decapsulated packet from A:1234 to T:1234 (T is an IP from the TUN subnet, as described in section 5.1.2) and forwards the packet. The server B responds to address T:1234 and since this is an IP address from the TUN subnet, the NATTT daemon receives the response packet.

This has a major consequence: Return packets from the other hosts are destined to the IP address in the TUN subnet. It is not immediately clear why these return packets should arrive on the NAT/host running the NATTT daemon. The forthcoming sections explore this point in detail.

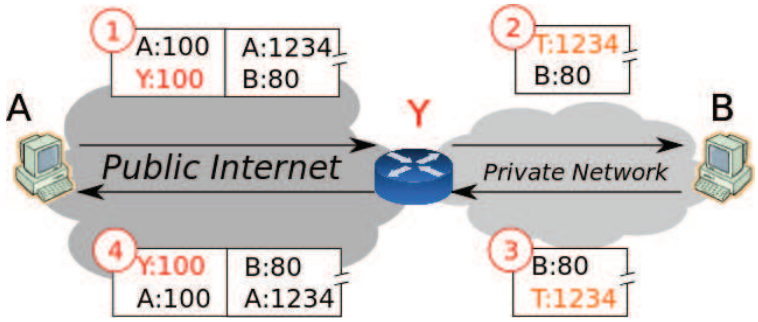

Figure 5.3: Basic tunneling - Actual implementation

5.2.1 Running NATTT server daemon on a NAT

As described in the last section, it is not immediately clear why packets destined to the TUN subnet arrive on the host/NAT running NATTT. This section and the

next one explore this point.

The NAT is the default route for the private hosts behind it. Therefore, packets sent by the private hosts to any destination outside of the local network are sent via the NAT.

Thus, it is possible to assign any subnet (excluding the local subnet) to the TUN adapter on the NAT, and packets destined to that subnet would reach the NAT. Once these packets arrive at the NAT, the packets are routed to the TUN adapter, and hence reach the NATTT daemon (as described in section 5.1.2).

Therefore, when the NATTT daemon is run on a NAT, it receives all packets destined to IP addresses in the TUN subnet.

5.2.2 Running NATTT server daemon on a host

As mentioned in the last section, private hosts send packets to destinations outside the local network via the NAT. Therefore, when NATTT is run on a host, there are 2 options for the address range used on the TUN adapter:

1. Use a subnet outside the local subnet and modify routing tables of other hosts so that they send packets destined for that subnet to the host running NATTT.

2. Use a range of addresses from the local subnet and advertise the presence of those addresses on the host running NATTT.

Option (1) defeats the basic purpose of NATTT. It is no longer possible to access any host behind the NAT; only hosts with a modified routing table. Further, devices like DVRs and surveillance cameras may not provide an interface to

modify their routing tables. Therefore, option (2) is chosen. The 'advertisement of addresses' referred to is achieved by having the NATTT daemon respond to ARP requests on that range of IP addresses. Therefore, others hosts send packet destined to the TUN subnet to the host running NATTT.

5.2.3 Architecture of the NATTT server daemon

Figure 5.4 illustrates the architecture of a NATTT server daemon running on a host. The NATTT daemon consists of two components: The tunnel manager and the ARP handler. The local host's IP is 10.0.0.5/24. The subnet assigned to the TUN device is 10.0.0.128/25. Therefore, decapsulated packets have a source IP in the 10.0.0.128/25 subnet (10.0.0.128, 10.0.0.129 in the figure) when they are forwarded to other hosts The ARP handler responds to ARP requests in this subnet. Therefore, others hosts send packets destined to this subnet to this host.

5.3 Overview of the entire system

The overall illustration of the system is presented in figure 5.5. The client and the server applications (Apache and Firefox) view the session as talking with the NATTT daemon. However, the user perceives the session as end-to-end. NATTT functions transparently and helps to achieve the end-to-end connectivity goal.

5.4 Implementation of NATTT on a NAT

There exist several third-party Linux-based firmware solutions for NATs (OpenWRT, DD-WRT, Tomato etc). These are in wide use for the functionality and

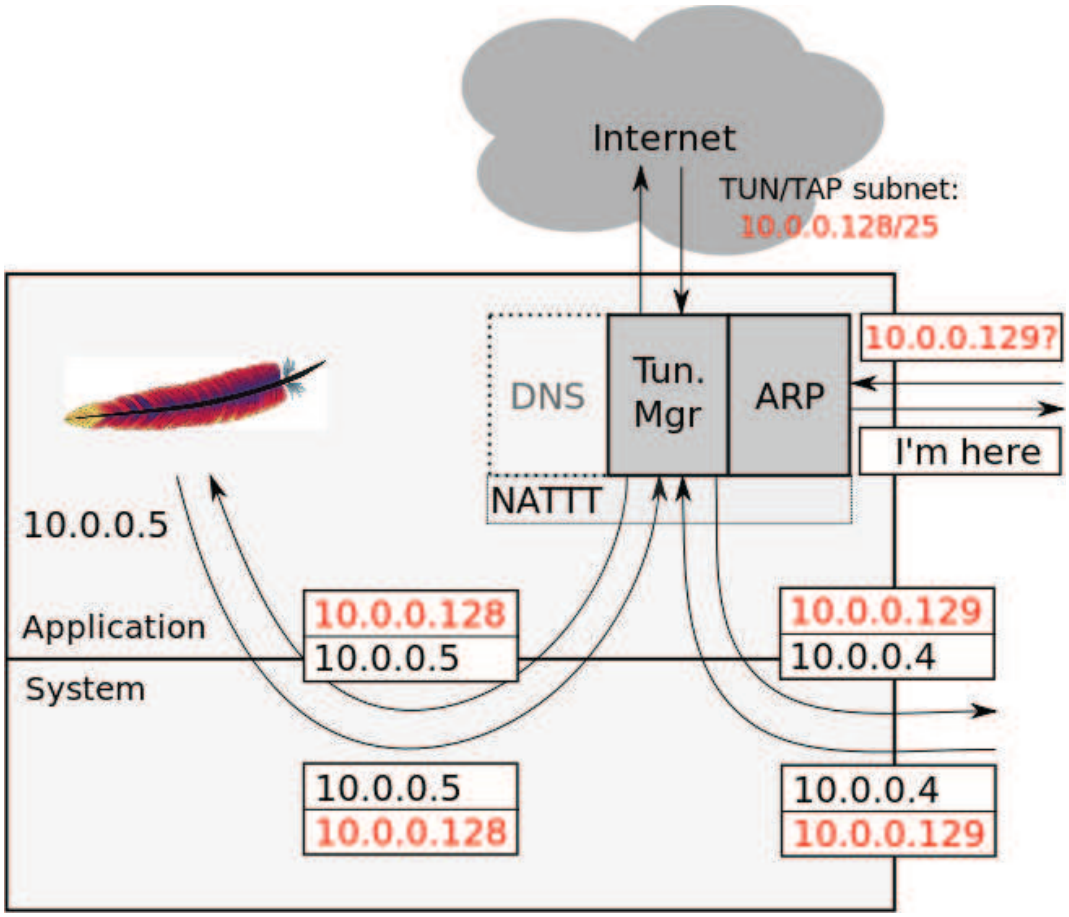

Figure 5.4: NATTT server daemon on a host

extensibility they provide. The NATTT daemon has been ported onto OpenWRT. The ease with which the NATTT daemon is deployed onto a NAT running OpenWRT/DD-WRT provides hope that the system will be rapidly deployed by users.

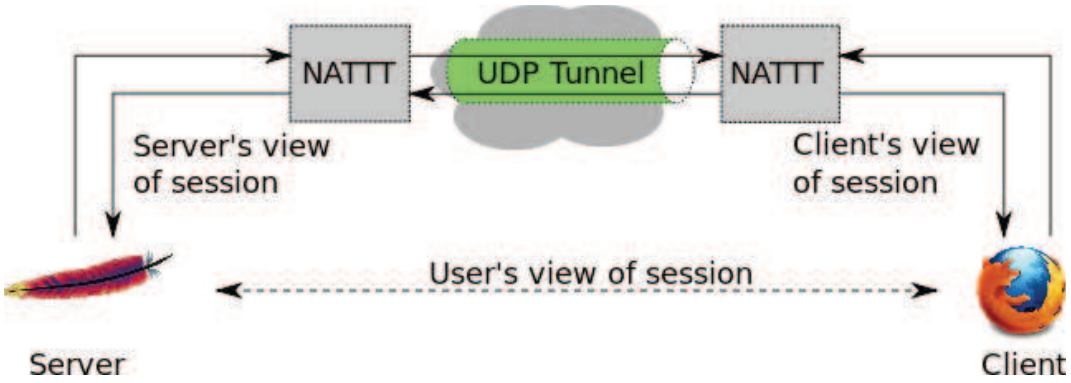

Figure 5.5: Overview of the entire system

5.5 Detecting a single NAT's external address

The NATTT system incorporates a mechanism to detect the external address on one layer of NAT. It is is implemented as a small application that runs on the server. It polls the local interfaces and external address of the NAT device for a change in IP address. The external address is retrieved from a HTTP server that returns the source address of the packet it receives. It is also possible to use standard protocols such as STUN to detect the external address.

When the client detects a change in the internal or external IP address, it makes a request via HTTP to a NATTT dynamic DNS server that updates the external and the internal IP addresses in the server's "NAT" DNS record.

It is also possible to use a standard dynamic DNS service to use two domain names to store two "A" records (external and internal IP) address and modify the NATTT daemon to work accordingly.

CHAPTER 6

Future work

6.1 Automatic NAT address detection

One of the key issues with NATTT and with any NAT traversal mechanism is the ability to detect the presence of a NAT device between a host and the Internet. The NATTT system already has a means to detect one layer of NAT (section 5.5). It needs to be extended to detect multiple layers of NAT. If the destination host could automatically detect external addresses of NAT devices and update the DNS records with their addresses, it would very much ease deployment and scalability of the NATTT system.

6.2 Nested NAT traversal

The NATTT client daemon does not support the ability to encapsulate a data packet with multiple headers. Therefore, the NATTT daemon implementation is unable to traverse nested NATs at this stage. It should however be noted that multiple headers are not required to traverse NAT devices that can forward port 100 to a host in the destination network, as described in section 4.5.

6.3 NAT Traversal with an unmodified client host

One drawback with the current NATTT system is that client hosts need to be deployed with the NATTT daemon. This would work very well as a NAT traversal mechanism when the client host is aware of the presence of a NAT between it and the destination, and NATTT is deployed on the client. In other words, NATTT would work when the client host is known. It would not work very well for a public service used by arbitrary clients unless they were NATTT aware.

NATTT as it is, goes one step closer in restoring end-to-end connectivity to hosts behind a NAT. However, it is not a totally effective solution unless an unmodified host can seamlessly connect to hosts behind a NAT device, as though the hosts had a publicly routable address.

One scenario where this might be most applicable is a user needing access to devices behind his home NAT from an external host which is not NATTT aware. An example of this would be a office worker, who on a foreign trip, needs to access his home PC and surveillance camera. It would very possible to forward the ports on the NAT device to the respective devices, but it would be difficult to access a device/service that the user did not expect that he would access remotely.

If this problem is solved, an arbitrary client would be able to access all services on all hosts behind the NAT. All it would take to traverse NATs would be deployment of the daemon on one host/the NAT (on the server side).

REFERENCES

[1] Andreas Mller, Georg Carle and Andreas Klenk, *Behavior and Classification of NAT Devices and Implications for NAT Traversal*, (IEEE Network Volume: 22, Issue: 5, p. 14-19)

[2] Yaw-Chung Chen and Wen-Kang Jiab, *Challenge and solutions of NAT traversal for ubiquitous and pervasive applications on the Internet* , (Journal of Systems and Software, Volume 82, Issue 10, October 2009, p. 1620-1626)

[3] Hechmi Khlifi, Jean-Charles Grégoire, and James Phillips, *VoIP and NAT/Firewalls: Issues, Traversal Techniques, and a Real-World Solution*, (IEEE Communications Magazine, July 2006, p. 93-99)

[4] Bryan Ford, Pyda Srisuresh and Dan Kegel, *Peer-to-Peer Communication Across Network Address Translators*, (Presented at the USENIX Annual Technical Conference, April 2005.)

[5] Salman A. Baset and Henning Schulzrinne, *An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol*, (IEEE Infocom, Barcelona, Spain, April 2006, p. 1-11)

[6] Saikat Guha, Paul Francis, *Characterization and Measurement of TCP Traversal through NATs and Firewalls*, (Internet Measurement Conference, October 2005)

[7] Andreas Mller, Georg Carle and Andreas Klenk, *On the Applicability of Knowledge Based NAT-Traversal for Home Networks*, (Lecture Notes in Computer Science, Volume 4982/2009)

[8] J. Rosenberg, J. Weinberger, C. Huitema, R. Mahy, *RFC 3489: STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)* (http://www.ietf.org/rfc/rfc3489.txt)

[9] J. Rosenberg, R. Mahy, P. Matthews, D. Wing, *RFC 3489: Session Traversal Utilities for NAT (STUN)* (http://www.ietf.org/rfc/rfc5389.txt)

[10] J. Rosenberg, R. Mahy, P. Matthews, *Internet Draft: Traversal Using Relay NAT (TURN)* (http://tools.ietf.org/html/draft-ietf-behave-turn-16)

[11] David Koblas, *SOCKS* (Presented at the USENIX UNIX Security Symposium

III, September 1992)

[12] M. Leech, M. Ganis, Y. Lee, R. Kuris, R. Kuris, L. Jones, *RFC 1928:SOCKS Protocol Version 5* (http://tools.ietf.org/html/rfc1928)

[13] Stuart Cheshire, Marc Krochmal, Kiren Sekar, *Internet Draft: NAT Port Mapping Protocol (NAT-PMP)*, (http://tools.ietf.org/html/draft-cheshire-nat-pmp-02)

[14] J Rosenberg, *Internet Draft: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*, (http://www.rfc-editor.org/authors/rfc5245.txt)

[15] H. Levkowetz, S. Vaarala, *Internet Draft: Mobile IP NAT/NAPT Traversal using UDP Tunnelling*, (http://tools.ietf.org/id/draft-ietf-mobileip-nat-traversal-05.txt)

[16] C. Boulton, J. Rosenberg, G. Camarillo, *Internet Draft: Best Current Practices for NAT Traversal for SIP*, (http://tools.ietf.org/id/draft-ietf-sipping-nat-scenarios-05.txt)

[17] Y.Takeda, *Internet Draft: Symmetric NAT Traversal using STUN*, (http://tools.ietf.org/id/draft-takeda-symmetric-nat-traversal-00.txt)

APPENDIX A

Code structure

The architecture of NATTT has 3 major components: the DNS Resolver, Tunnel Manager and the ARP handler. This appendix breaks these 3 high level components down into their C++ classes and specifies their interfaces and interactions.

The overall design and interactions of these classes is shown by Figure A.1. The individual classes are described below.

A.1 nat3d

This section describes the nat3d daemon component, which is described by Figure A.2. This daemon is the actual NATTT executable. It runs 2 threads: one for the DNS resolver logic, and a second one that manages the IP tunnels. This component is very lightweight and is not responsible for much logic.

A.1.1 Methods

Public Methods

- main(): This is the main method of the daemon. It spawns a worker thread for the resolver and the directly calls the tunnel manager's logic.

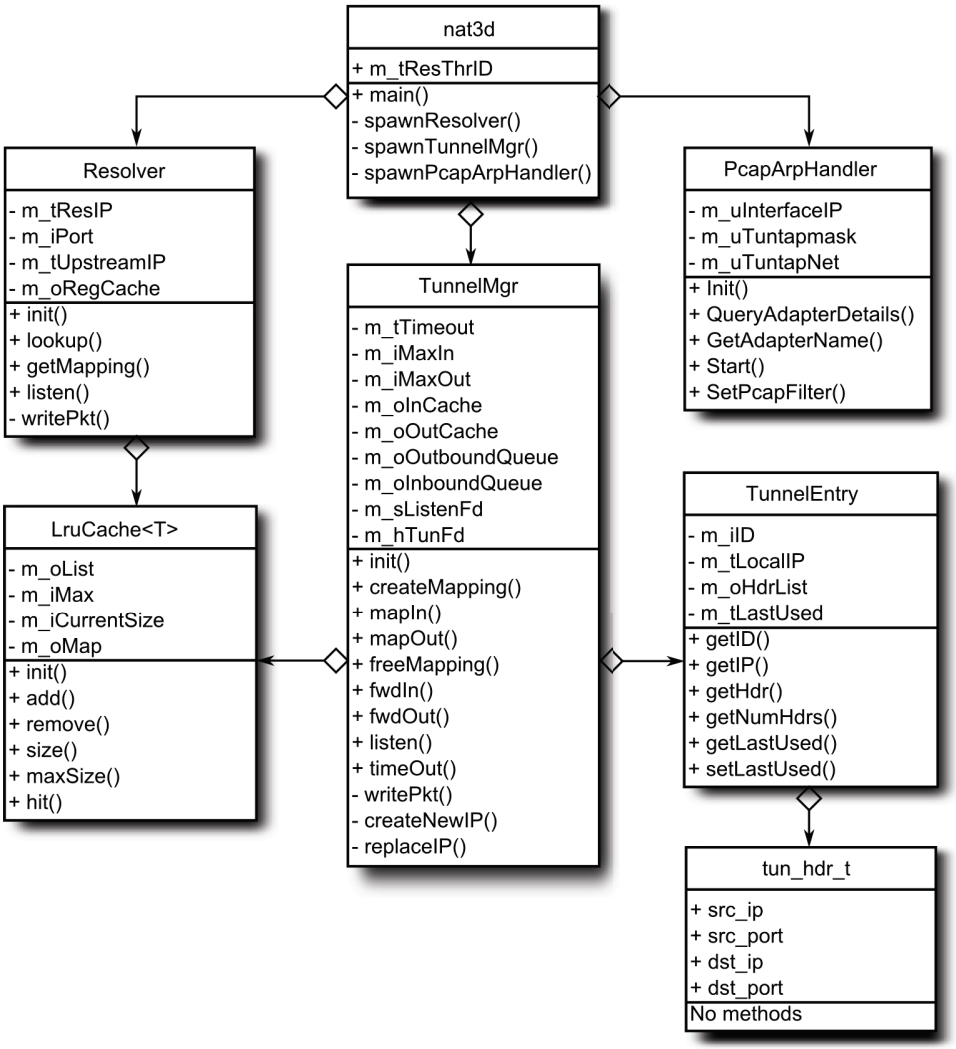

Figure A.1: UML diagram of the NATTT daemon

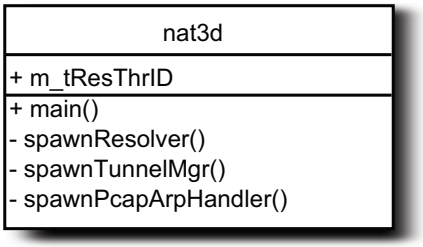

Figure A.2: Component diagram: NATTT daemon

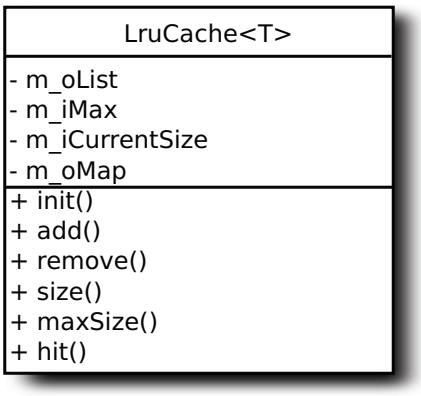

Figure A.3: Component diagram: LruCache

- spawnResolver(): This method is uses as a callback to pthread_create() and initializes and drives the DNS resolver logic.
- spawnTunnelMgr(): This method initializes and drives the IP tunnel manager logic.
- spawnPcapArpHandler(): This method initializes and and drives the ARP handling logic. The PCAP library is used to filter out incoming ARP requests and respond to ARP requests only on the TUN subnet.

A.1.2 Member Variables

- m_tResThrID: thread_t This is the ID of the thread that runs the DNS logic.

A.2 LruCache

This section describes the LruCache component, which is described by Figure A.3. The LRU cache is a simple template class that implements a Least Recently Used

eviction policy for its entries. It is configurable to have a maximum size and has simple accessors and an init function. It uses an STL map¡¿ class to to store its data and an custom list class that supports random access.

A.2.1 Methods

Public Methods

- init(): This method initializes all of the internal state. This method takes an optional parameter that specifies the maximum size that the cache may grow.
- add(): This method takes a template type ¡T¿ and adds it to the internal structures.
- remove(): This method removes an object from the cache.
- size(): This method returns the current size of the cache.
- maxSize(): This method returns the upper limit on how large the cache may grow.
- hit(): This method is used to lookup an entry in the cache and move it to the most recently used if it exists.

A.2.2 Member Variables

- m_oList
- m_iMax
- m_iCurrentSize
- m_oMap

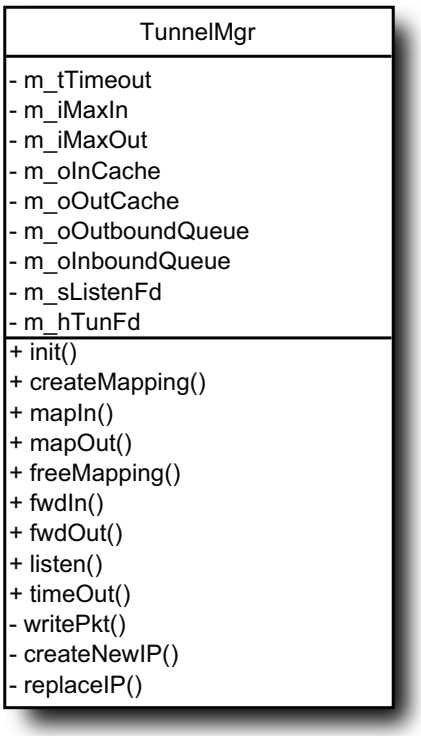

Figure A.4: Component diagram: TunnelMgr

A.3 TunnelMgr

This section describes the TunnelMgr component, which is described by Figure A.4. This component maintains a cache of inbound and outbound connections. These caches are instantiations of the LRU Cache class (described in Section A.2). The connections are placed in caches so that if connections are not properly cleaned up, the manager will not continue to maintain stale connection mappings.

As outbound connections are made new mappings are created via the createMapping() method. The results are then placed in the outbound connection cache, and referenced by the outbound index (which is an STL map). Similarly,

inbound connections are managed by the inbound analogs. These structures are managed separately so that audit trails can be maintained about active inbound vs. outbound connections, and so that logic can (potentially) be different when handling them. Caches are filled with TunnelEntry objects that fully describe the headers needed to encapsulate outbound packets, and the local IP address needed to forward inbound packets.

The main thread of the manager blocks on the listen() method which awaits incoming connections and outgoing packets (via a select() call). Outbound packets are passed the the fwdOut() method which creates the proper headers and encapsulates the packet. Inbound connections are handed to the fwdIn() method which tries to interpret the inbound headers and look them up in the inbound cache.

A.3.1 Methods

Public Methods

- init(): Initialize the internal state of the manager and clean up all data structures.
- createMapping(): Create a new tunnel. Map a new IP address to a set of TunnelEntry objects that specify the headers needed to create the tunnel. Place this mapping in the appropriate IP lookup cache and index.
- mapIn(): Take an inbound packet and determine which internal IP address it belongs to and retrieve all state needed.
- mapOut(): Take an outbound packet and determine what headers are needed to deliver it to the proper destination and retrieve all state needed.

- freeMapping(): Clean up state information from index and cache for a specific mapping (IP and tunnel headers).
- fwdIn(): Take an inbound packet, lookup its mapping info, de-encapsulate, and format the packet for local delivery.
- fwdOut(): Take an outbound packet, lookup its mapping info, encapsulate it for tunneling to a remote destination.
- listen(): Await inbound connections on a network port.
- timeout(): Check to see if any tunnels have been inactive for long enough that we can reclaim and free their state.

Private Methods

- writePkt(): Given a set of headers or a new IP address encapsulate of de-encapsulate and return a routable packet.
- createNewIP(): Perform the platform specific operations of allocating a new IP address in the local subnet.
- replaceIp(): Replace the source IP of the decapsulated packet with an IP from the TUN subnet before forwarding it to other hosts.

A.3.2 Member Variables

- m_tTimeout: time_t - This member is the amount of time an entry is allowed to be idle before it can be considered for removal.
- m_iMaxIn: int - This member is the maximum number of inbound mappings that may be kept.

- m_iMaxOut: int - This is the maximum number of outbound mappings that may be kept.
- m_oInCache: LruCache - This member maintains a lookup of TunnelEntry objects based on the IP address the inbound packet came from.
- m_oOutCache: LruCache - This member maintains a lookup of TunnelEntry objects based on the IP address the outbound packet came from.
- m_oOutboundQueue: TunnelQueue - This queue enqueues packets received from the TUN/TAP interface after they are encapsulated.
- m_oInboundQueue: TunnelQueue - This queue enqueues packets received from the listening port of NATTT after they are decapsulated and source IP replaced by replaceIp()
- m_sListenFd: socket - This is the socket on which NATTT sends/received encapsulated packets.
- m_hTunFd: HANDLE (Windows), descriptor (UNIX/Linux) - This is a handle to the TUN/TAP device.

A.4 win_tun_mgr

This section describes the component, which is described by Figure A.5.

This component represents the Windows-specific implementation details of the tunnel manager component.

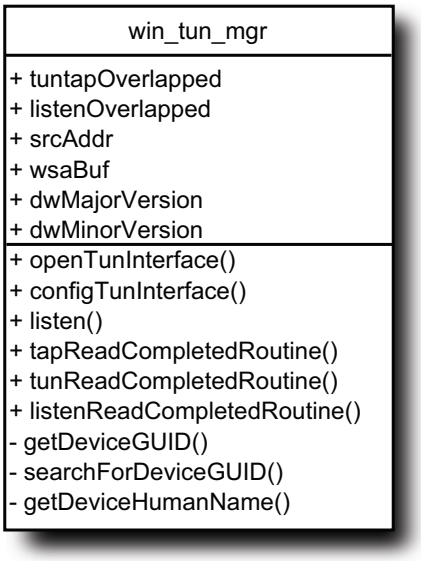

Figure A.5: Component diagram: win_tun_mgr

A.4.1 Methods

Public Methods

- openTunInterface() - Windows-specific code to open a TUN/TAP interface.
- configTunInterface() - Assign the TUN/TAP device with the subnet from configuration file, retrieve it's MTU value.
- listen() - Await inbound connections on the TUN/TAP device and the listening socket. Uses Windows overlapped I/O.
- tapReadCompletedRoutine() - Callback function that is called when reading from TAP device is completed.
- tunReadCompletedRoutine() - Callback function that is called when reading from TUN device is completed.
- listenReadCompletedRoutine() - Callback function that is called when reading from listening socket is completed.

Private Methods

- getDeviceGUID() - Get the GUID (Globally Unique ID) for the TUN/TAP device. Invokes searchForDeviceGUID().
- searchForDeviceGUID - Search the Windows registry for the GUID.
- getDeviceHumanName() - Get the name of the network adapter as the user sees it (i.e. Local Area Connection 2 etc) as opposed to a GUID.

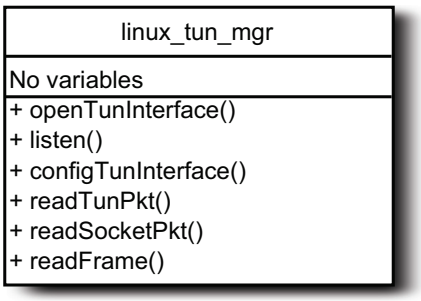

Figure A.6: Component diagram: linux_tun_mgr

A.4.2 Member Variables

- tuntapOverlapped : OVERLAPPED - Required by Windows to perform asynchronous/overlapped IO on the TUN/TAP device.
- listenOverlapped : OVERLAPPED - Required by Windows to perform asynchronous/overlapped IO on the listening socket.
- srcAddr : SOCKADDR - Contains source address of the packet just read from the listening socket.
- wsaBuf : WSABUF - Contains the data just read from the listening socket.
- dwMajorVersion : DWORD - Contains the Windows major release number.
- dwMinorVersion : DWORD - Contains the Windows minor release number.

A.5 linux_tun_mgr

This section describes the component, which is described by Figure A.6.

This component represents the Linux/UNIX-specific implementation details of the tunnel manager component

A.5.1 Methods

Public Methods

- openTunInterface() - Linux/UNIX specific code to open a TUN/TAP device and retrieve a handle to it.
- listen() - Wait for incoming connections on TUN/TAP device and the listening socket.
- configTunInterface() - Assign the TUN device with the subnet from the configuration file, retrieve its MTU.
- readTunPkt() - Retrieve one packet from the TUN device.
- readSocketPkt() - Retrieve one packet from the listening socket.
- readFrame() - Retrieve one Ethernet frame from the TAP device.

A.6 TunnelEntry

This section describes the TunnelEntry component, which is described by Figure A.7. This is a very simple abstract data type (ADT). It encapsulates the set of headers needed to encapsulate an outbound IP packet, and the local address used to deliver traffic to a local client. The goal of this class is to simple wrap the logic that determines how many headers (i.e. levels of NAT) are needed and to actually encapsulate and de-encapsulate packets.

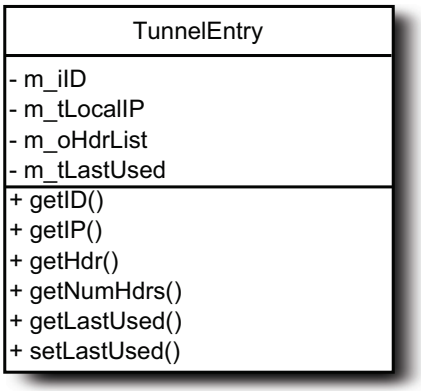

Figure A.7: Component diagram: TunnelEntry

A.6.1 Methods

Public Methods

- getID(): Simple accessor that returns the ID of this entry.
- getIP(): Simple accessor that returns the local IP address that corresponds to this tunnel.
- getHdr(): Takes an ordinal postion and returns the header there.
- getNumHdrs(): Returns the number of headers.
- getLastUsed(): Returns the time of the last access.
- setLastUsed(): Sets the last access time.

A.6.2 Member Variables

- m_iID: int - Internal ID of this entry.
- m_uLocalIP: unit32_t - Local IP address of this tunnel.
- m_oHdrList: list¡¿ - List of headers to enapsulate packets for this tunnel.
- m_tLastUsed: time_t - Time of last use.

A.7 tun_hdr_t

This section describes the tun_hdr_t struct, which is described by Figure A.8. This component is a very simple data structure that defines a single header needed to encapsulate a single layer of a NAT3 tunnel. For example, with a single NAT box,

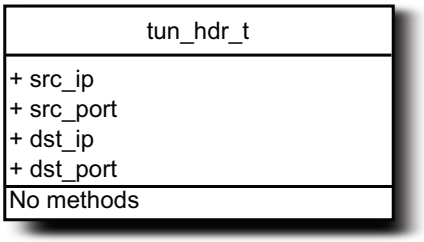

Figure A.8: Component diagram: tun_hdr_t

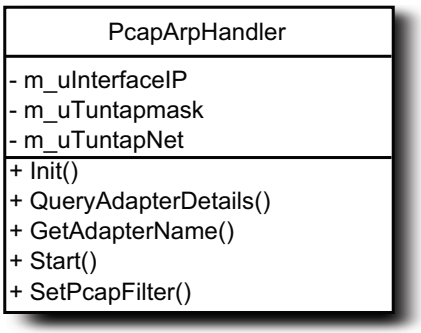

Figure A.9: Component diagram: PcapArpHandler

2 tun_hdr_t structures are needed (one for the outermost IP header, and one for the internal header).

A.7.1 Member Variables

- m_uSrcIP: uint32_t
- m_uSrcPort: uint16_t
- m_uDstIP: uint32_t
- m_uDstPort: uint16_t

A.8 PcapArpHandler

This section describes the component, which is described by Figure A.9.

This component represents the ARP handler, which enables the TUN/TAP device to receive packets on the TUN subnet when NATTT runs on a host. PCAP filtering system is used to listen for ARP requests on the TUN subnet. When such

ARP requests arrive, this component responds to the requests. Therefore packets destined to the TUN subnet arrive on the host.

A.8.1 Methods

Public Methods

- init() - Initialize the PCAP packet filtering system
- QueryAdapterDetails() - Show a list of adapters to the user and ask which adapter to use - in both NATTT server and client modes. For NATTT server, the adapter on which the NAT is located is chosen.
- GetAdapterName() - Given an IP address, find the name of the adapter on which it is present.
- Start() - Start the packet filtering and ARP response process.

Private Methods

- SetPcapFilter() - Generate the ARP filtering command and apply it to the PCAP library so that this component only sees ARP requests for IPs on the TUN subnet.

A.8.2 Member Variables

- m_uInterfaceIp : uint32 - The IP of the interface on which the listening socket is bound. This is the same interface on which PCAP listens for ARP requests.
- m_uTuntapMask : uint32 - The netmask assigned to the TUN device.
- m_uTuntapNet : uint32 - The subnet assigned to the TUN device.

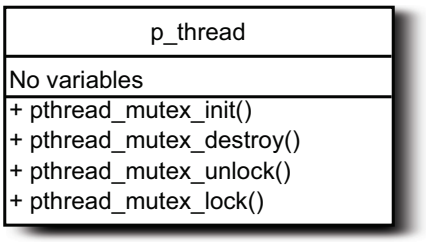

Figure A.10: Component diagram: p_thread

A.9 p_thread

This section describes the component, which is described by Figure A.10.

This component provides a POSIX interface to some Windows mutex functions.

A.9.1 Methods

Public Methods: The methods below are POSIX interfaces to their equivalent Windows functions.

- pthread_mutex_init()
- pthread_mutex_destroy()
- pthread_mutex_unlock()
- pthread_mutex_lock()

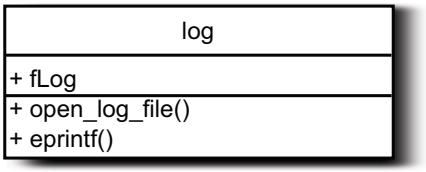

Figure A.11: Component diagram: log

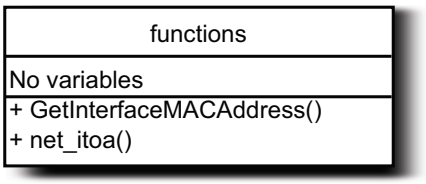

Figure A.12: Component diagram: functions

A.10 log

This section describes the component, which is described by Figure A.11.

This component represents the logging mechanism in NATTT.

A.10.1 Methods

Public Methods

- open_log_file - Log events to the log file specified by the filename parameter. It is possible to redirect the logging to /dev/null or to stderr by passing the filenames as 'none' and 'stderr' respectively.
- eprintf() - Write string to log file.

A.10.2 Member Variables

- fLog : FILE* - A file handle to the log file.

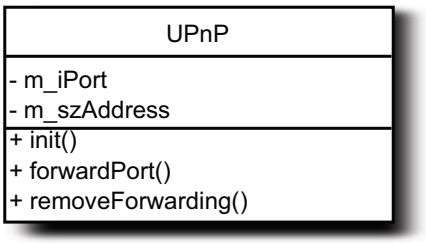

Figure A.13: Component diagram: UPnP

A.11 functions

This section describes the component, which is described by Figure A.12.

This component represents some utility functions available for both Windows and Linux/UNIX platforms.

A.11.1 Methods

Public Methods

- GetInterfaceMACAddress() - Get the MAC address of the interface whose name is passed as parameter.
- net_itoa() - Convert an integer representation of an IP address to a string representation.

A.12 UPnP

This section describes the component, which is described by Figure A.13.

This component represents the Linux/UNIX implementation of port forwarding via UPnP. It uses the miniupnpc library.

A.12.1 Methods

Public Methods

- init() - Initialize the UPnP system. Search for a NAT and retrieve its IP address.
- forwardPort() - Forward a port on the UPnP-enabled NAT.
- removeForwarding() - Remove a port forwarding entry on the UPnP-enabled NAT.

A.12.2 Member Variables

- m_iPort : uint16_t - The port to be forwarded.
- m_szAddress : char[] - The IP address to which the above port should be forwarded.

A.13 DnsResolver

This section describes the DnsResolver component, which is described by Figure A.14.

The resolver is a very loose wrapper around the DNS library and the DnsQuery class. The resolver's main task is to read bytes from the wire to feed into the DNS library which returns fully-parsed objects representing those DNS queries. It

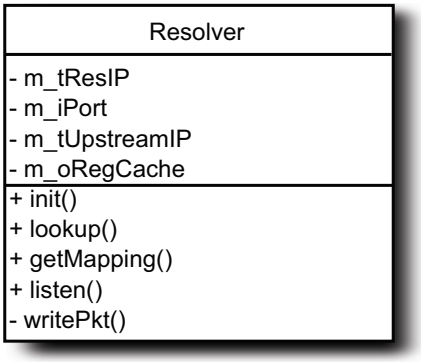

Figure A.14: Component diagram: DnsResolver

then feeds those queries into objects of the DnsQuery class, which will be described in a later section. That class will have the resolver send further packets as required by the NAT3-specific DNS logic.

A.13.1 Methods

Public Mehtods

- init(): This method reads the system resolv.conf file and prepares the object to accept DNS queries.
- listen(): This method edits the resolv.conf file to interpose the NAT3 resolver between the system resolver libraries and the local caching resolver. It causes the daemon to bind to the DNS socket and calls read_loop() to accept DNS queries.

Private Methods

- select_loop(): This method is a basic select() loop, which calls read_loop() and write_loop() depending on the read/write state of the file descriptor.
- read_loop(): When the DNS socket is ready to be read, this method continues to read packets from it until the recvfrom system call would block. Each time it reads a packet, it calls the DNS library to parse it and hands the parsed packet to read_packet() to handle it.
- write_loop(): If there is a queue of DNS packets waiting to be sent and the UDP socket becomes ready to write, this method will dump as many packets from the queue as it can before the socket would block.

- read_packet(): Once a properly parsed packet is received, this method handles it. This is the method acts as a marshaller between DnsQuery objects and the DNS packets from the wire.
- try_send(): This method will attempt to send a DNS packet if there is no packet queue. It will enqueue it if there is any problem sending.
- enqueue(): Add packet to queue for sending.
- writePkt(): Once the socket becomes ready to write, this method is used to write enqueued data.

A.13.2 Member Variables

- m_uResIP: uint32_t - IP address of IP to listen on
- m_uPort: uint16_t - Port to listen on
- m_iFD: int - File descriptor used for network.
- m_uUpstreamIP: uit32_t - IP of upstream caching resolver.
- m_oReqCache: LruCache - Cache of outstanding queries.

A.14 DnsQuery

This section describes the DnsQuery component, which is described by Figure A.15.

The DnsQuery component encapsulates the NAT3-specific DNS logic needed by this application. It responds to error replies to questions for A resource records by generating questions for NAT3 records. If those are properly received, it asks the TunnelManager to set up a mapping. It will then construct a DNS response

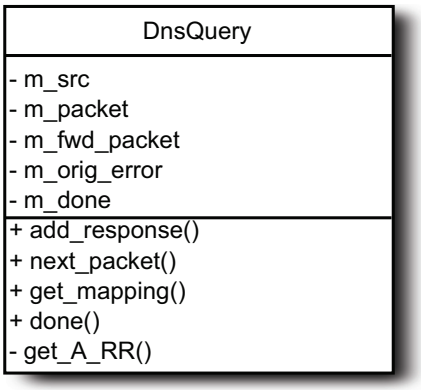

Figure A.15: Component diagram: dns_query

with an A record for this local mapping. It returns both this DNS response and the mapping itself to the resolver.

A.14.1 Methods

Public Methods

- constructor: The constructor takes a socket address and a question packet and encapsulates them.
- add_response(): This method takes a parsed DNS packet that represents a response to a previous packet. The DnsQuery object then uses the data in this packet to create a "next packet" to send, encapsulating the high-level NAT3-specific DNS logic.
- next_packet(): This method returns a wire-encoded DNS message and a destination socket address representing the next packet to be sent.
- get_mapping(): This methods returns the mapping given by the TunnelManager to the DNS resolver if one has been made.
- done(): True when a query has been completed.

Private Methods

- get_A_RR(): Returns the first A resource record from the question section of the input packet, if one exists.

A.14.2 Member Variables

- m_src: Socket address representing the source of the original packet.

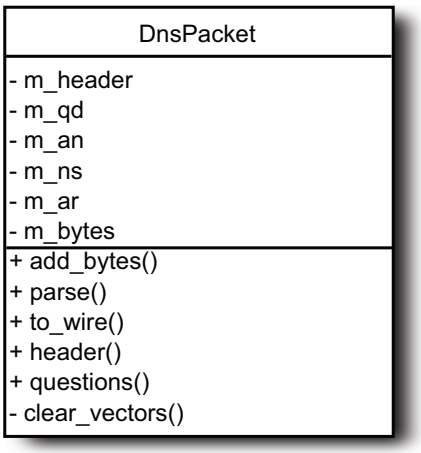

Figure A.16: Component diagram: DnsPacket

- m_packet: Original DNS reply packet that was received from the caching resolver.
- m_fwd_packet: Packet to be forwarded to the caching resolver or original questioner.
- m_orig_error: Original error packet (i.e., NXDOMAIN) in case no NAT3 RR could be found.
- m_done: Flag used by done() method.

A.15 DnsPacket

This section describes the component, which is described by Figure A.16.

This component represents a high-level view of a DNS packet. It encapsulates all the header information as well as the RRs that were received from the wire.

A.15.1 Methods

Public Methods

- add_bytes(): This method is called by the resolver when bytes have been received from the network. It can be called as many times as necessary and will continue to add bytes to the buffer. For a UDP packet, this will only be called once. However, this can be called arbitrarily many times for a TCP packet.
- parse(): Once all the data has been received from the wire, calling this method parses the packet and returns a boolean value depending on the success or failure of the parse.
- to_wire(): This method converts the message from high-level objects into the canonical wire representation of a DNS packet.
- header(): This method returns the header section of the DNS packet.
- questions(): This method returns a list of the question RRs from the packet.

Private Methods

- clear_vectors(): This method frees the memory associated with the RRs and is used by the destructor.

A.15.2 Member Variables

- m_header: A DnsHeader object representing the parsed headers.
- m_qd: Question RRs from the original DNS packet.

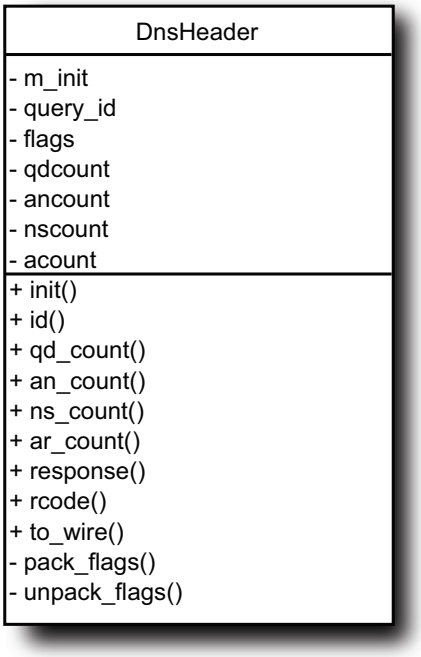

Figure A.17: Component diagram: DnsHeader

- m_an: Answer RRs from the original DNS packet.
- m_ns: Authoritative NS RRs from the original DNS packet.
- m_ar: Additional RRs from the original DNS packet.
- m_bytes: Buffer of bytes from the network.

A.16 DnsHeader

This section describes the DnsHeader component, which is described by Figure A.17.

This class holds the header information for a DNS packet, providing convenience methods for various data.

A.16.1 Methods

Public Methods

- init(): Loads the header from a wire representation.
- id(): Get the ID of the query.
- qd_count(): Get the query count.
- an_count(): Get the answer count.
- ns_count(): Get the authoritative NS count.
- ar_count(): Get the additional count.
- response(): Is this a response?
- rcode(): Get the RCODE.
- to_wire(): Convert to the wire format.

Private Methods

- pack_flags(): Pack flags into wire representation.
- unpack_flags(): Unpack flags into a common C-style structure.

A.16.2 Member Variables

- m_init: True if packet has been initialized.
- query_id: Query ID of the packet.

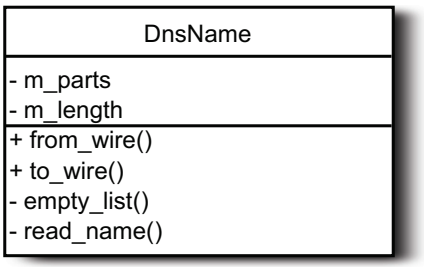

Figure A.18: Component diagram: DnsName

- flags: Header flags.
- qdcount: Question count.
- ancount: Answer count.
- nscount: NS count.
- arcount: Additional count.

A.17 DnsName

This section describes the DnsName component, which is described by Figure A.18.

This class implements the semantics of a DNS name. Due to the use of DNS-style compression, handling of DNS names is non-trivial.

A.17.1 Methods

Public Methods

- from_wire(): A factory method which parses a name from the wire and returns an object of the DnsName class on successful parse.
- to_wire(): Takes a name length and an optional pointer and converts the name into a compressed wire representation.

Private Methods

- empty_list(): Frees memory associated with the object.
- read_name(): Reads a name from a packet, handling compression as needed.

A.17.2 Member Variables

- m_parts: List of the period-delimited parts of the DNS name.
- m_length: Length of the uncompressed name.

A.18 DnsRR

This section describes the component, which is described by Figure A.19.

This represents an RR of any type for which we do not have a specific implementation. It handles the high-level functions of a DNS RR packet.

A.18.1 Methods

Public Methods

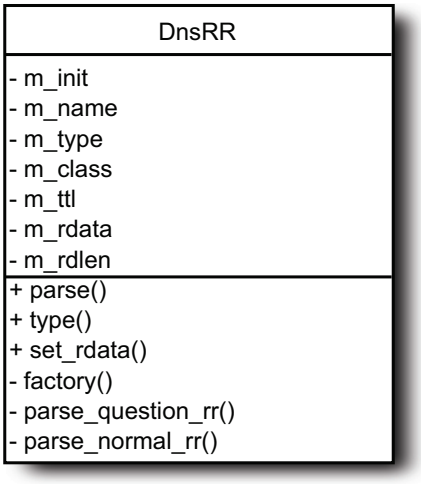

Figure A.19: Component diagram: DnsRR

- parse(): Takes a wire representation of a DNS packet and parses it into an RR. This is a factory method, and will return an object inheriting from DnsRR depending on the type of the RR.
- type(): Returns the numeric type of the RR. A list of symbolic constants for common types is included.
- rdata_valid(): Returns true if the RDATA is of a valid format (which depends on the underlying RR type).
- set_rdata(): Set the RDATA section to a bag of bytes.

Private Methods

- factory(): Static factory method that returns a new RR of a numeric type, used by parse_question_rr() and parse_normal_rr().
- parse_question_rr(): Parse an RR from the question section, returns an RR of the proper type if no error occurs.
- parse_normal_rr(): Parse an RR from any section other than the questions section. Returns RR of the proper type if no error occurs.

A.18.2 Member Variables

- m_init: True if the object has been init.
- m_name: DnsName object representing the name of the packet.
- m_type: Numeric type of the RR.
- m_class: RR class (as defined by RFC1035).

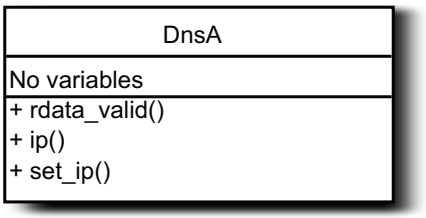

Figure A.20: Component diagram: DnsA

- m_ttl: TTL of RR (-1 if question RR).
- m_rdata: Binary RDATA.
- m_rdlen: Length of RDATA.

A.19 DnsA

This section describes the DnsA component, which is described by Figure A.20.

The DnsA is a subclass of DnsRR which implements a common A resource record.

A.19.1 Methods

Public Methods

- rdata_valid(): Check if the data is valid
- ip(): Get the IP
- set_ip(): Set the IP.

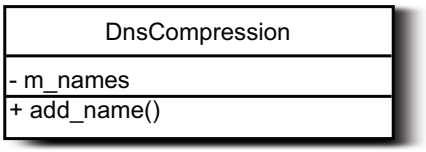

Figure A.21: Component diagram: DnsCompression

A.20 DnsCompression

This section describes the DnsCompression component, which is described by Figure A.21.

DNS name compression is tricky business, and this class aims to simplify it by providing a compression context.

A.20.1 Methods

Public Methods

- add_name(): This method takes a name and returns the number of leading name parts that remain after compression. It also returns a pointer if any compression occurs. It records metadata for use in future add_name() method calls.

A.20.2 Member Variables

- m_names: A mapping of names and sub-names to DNS pointers. For the purposes of compression, the names are actually stored backwards.