ÉCOLE NATIONALE SUPÉIEURE DE TÉLÉCOMMUNICATIONS

POLITECNICO DI TORINO
Faculty of information engineering
Specialisation in informatic engineering

Final project report

Peer-to-Peer Reliable Connectivity Across Network Address Translator

A Software Approach to Solving NAT Traversal Problem

Tutors:

AJMONE Marco
DAX Philippe
GORGES Didier
RABRET Laurent

Student:
GABALLO Luca

YEAR 2004-2005

Acknowledgements

During the course of my thesis work, there were many people who were instrumental in helping me. Without their guidance, help and patience, I would have never been able to accomplish the work of this thesis. I would like to take this opportunity to acknowledge some of them.

I'm grateful to my supervisor, **Didier Gorges**, the person who guided me during my six-month internship, whose expertise, understanding, and patience, added considerably to my experience. He provided me with educational direction and technical support. A very special thanks goes out to **Laurent Rabret** for his scientific, technical and moral support. I appreciate his vast knowledge and skill in many areas. Many thanks to Emmanuel Breton for the wonderful collaborative experience and all the team. Their creative ideas, talent, and help has been more than I can possibly thank for. I would like to express my gratitude to **Prof. Philippe Dax**, from the ENST Institute, for his encouragement and guidance during my work. There are no words to thank **Prof. Marco Ajmone** from the Politecnico di Torino, who accepted to remotely follow my thesis work from Italy.

There are many people who deserve to be thanked for their love, friendship, help, support, or even simple physical proximity during these long years of school and study. In this limited space I will try to do so, even if it's hard to properly express my gratefulness.

Let me start from my family. I owe you everything, and fear I can't love you enough in return for that. Thanks to you all for your continuous and ever-caring support, I've always felt your presence so near to me. Tanks to my father and my mother, but in particular to my sister Serena, my little treasure. A special thanks to my brother Graziano and to Nadia, the persons who have taught me the important value of a family.

Most importantly, I would like to thank all my friends. Ilan, who taught me that 80 heart contractions per minute are not enough in order to totally profit by this life. Alessandro, I don't know his secret, but he's able to transform every problem in something not drastic at all to find again your smile. Serena, the North Star, a really good model for me. The only person capable to menage three earthquakes like we are. Consuelo, the good friends are not never enough. I'm the youngest, you all are my second family.

A special thanks to the *Pizzaconnection* team, Carlo, Davide and Luca, and its honorary members, Silvia, Sara, Nacho and Tono, who have shared with me this great experience in Paris. Tanks to all my friends at the Politecnico di Torino, who, with me, get through these five hard years. On with my old dear friends in Lecce, Annalisa, Alessandra, Emanuela, Valeria, Gianni, Roberto, Totó. It's refreshing to see that physical distance cannot break deep, heart-felt friendships. Thanks for being there when I need you.

A new shiny star illuminates my heart. *Hope* is the meaning of her name, hope as desire of love, or as the belief that it is obtainable. Thank you my dear Amal.

Tanks to Salento, my native land, Torino, and Paris. Tanks to the many people who were close to me during this hard but pleasant years - you know who you are.

The best tanks to God because graduate school isn't the most important thing in life, but good friends, good times, love and happiness are.

Abstract Peer-to-peer applications need direct connection between nodes. Network Address Translation (NAT) cause well-knows difficulties since the peers have no globally valid IP address. Some NAT traversal approaches are emerging, but they suffer from several problems. This work analyses the NAT functionalities, the UDP/TCP communication through the NATs, analyses the different approaches, try to find a good solution for the NAT traversal problem and implements it.

Contents

List of Figures

List of Tables

The Company

France Telecom

France Telecom is one of the world's leading telecommunications carriers, with 121.5 million customers on the five continents (220 countries and territories) and consolidated operating revenues of 46.1 billion euros for 2003 (23.2 billion euros for 1st semester 2004). Through its major international brands, including Orange, Wanadoo, Equant and GlobeCast, France Telecom provides businesses, consumers and other carriers with a complete portfolio of solutions that spans local, long-distance and international telephony, wireless, Internet, multimedia, data, broadcast and cable TV services. France Telecom is the second-largest wireless operator and Internet access provider in Europe, and a world leader in telecommunications solutions for multinational corporations. France Telecom (NYSE: FTE) is listed on the Paris and New York stock exchanges.

France Telecom R&D

France Telecom puts technological innovation at the heart of its concerns. With 4200 researchers and 7500 patents, its R&D is the engine of its innovation capability, in France and abroad. Its role is to anticipate technological revolutions and new uses, providing innovation to offer customers the best from telecommunications, simultaneously imagining today the technologies which will be part of their daily life tomorrow. Its R&D's results have put the Group in the leading position in the world in terms of telecommunications research and development. Established on 17 sites, including 9 abroad (London, San Francisco, Boston, Tokyo, Varsow, Beijing, Seoul, New Delhi, Canton), the researchers of France Telecom are involved in the design and the development of approximately 70% of the products of the group.

Chapter 1

The Context: The SOLIPSIS Project

This work is born in the SOLIPSIS project, a pure peer-to-peer system for a massively shared virtual world. In SOLIPSIS there is no server at all: it only relies on end-users' machines. Traverse the NAT is an exigence for peers in the SOLIPSIS network to exchange data in a reliable mode. It is not just a SOLIPSIS necessity but involve all the peer-to-peer applications.

This chapter presents the SOLIPSIS world, to better explain the source of the problem and for a properly comprehension of the solution and implementation requirements.

1.1 Solipsis

A shared virtual world is a computer-generated space used as a metaphor for interactions. Entities, driven by users or by computer, enter and leave the world, move from one virtual place to another and interact in real-time. The SOLIPSIS system is a network of peers, where peers are connected entities sharing a virtual space. It intends to be scalable to an unlimited number of users and accessible by any computer connected to the Internet. It does not make use of any server and is solely based on a network of peers. [1]

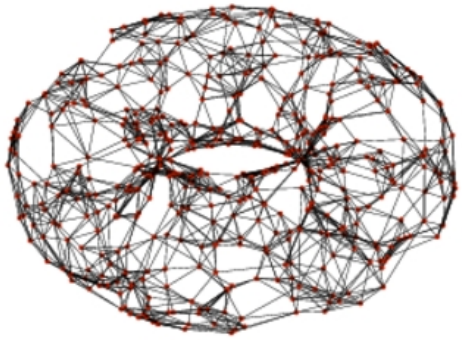
In the SOLIPSIS system, each participating computer runs a specific software that holds and controls one or several peers. These peers implement the entities of the virtual world and "perceive" their surroundings. Peers can be lite pieces of software and Solipsis aims to be accessible to low end computers at 56kbs and to mobile wireless devices and not only to full featured broadband connected engines. Connected peers may exchange data like video, audio, avatars movements or any kind of events affecting the representation of the virtual world.

SOLIPSIS is a system intended to create a massively shared computer- generated space. This virtual world may welcome entities, lite pieces of software running on any computer connected to Internet. The entities, driven by users or by computer, enter and leave the world, move from one virtual place to another and interact in

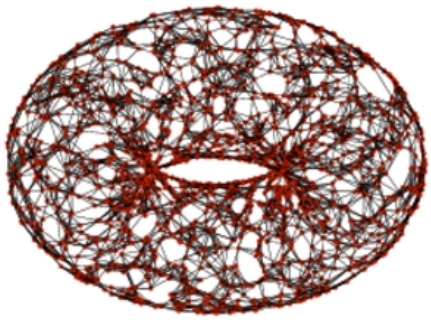
real- time.

In comparison with other existing systems for shared virtual world, SOLIPSIS is a fully *scalable* system. That is, the behaviour of SOLIPSIS is not altered when the number of active entities varies several orders of scale. Moreover, SOLIPSIS ensures the coherency of the virtual world, in the sense that it guaranties that a virtual scene is identically perceived by all entities observing it.

The virtual world is the surface of a torus (Figure 1.1). Each entity has a position on the torus. The SOLIPSIS protocol provides all the material for communication between entities. Especially, entity characteristics and virtual events can be easily transmitted from one entity to another. The SOLIPSIS protocol also ensures that the virtual world is still unique by guarantying the connexity of the communication graph modeling the network of participating nodes. [26]



500 entities



2000 entities

Figure 1.1: Solipsis network growth simulation. Simulation assumes that entities are uniformly distributed in the virtual world

Chapter 2

The Problem and the Solution Requirements

2.1 The NAT Traversal Problem

TCP connections between two machines are required to build peer-to-peer (P2P) applications. The users can connect to the Internet from different places (so, they have no fixed network address) and, often, they are behind a Network Address Translation (NAT) box. The problem is that computers connect to the Internet via NAT boxes can make outbound connection to the public Internet but cannot receive inbound connections.

NAT boxes allow several computers, in a private network, to share one public IP address and their communication to the public Internet pass through the NAT box, that maintains a table of mappings and use port translation in order to determine which computer should receive the response. This cause many difficulties for a peer-to-peer communication, since the peers cannot be reachable at any global valid network address. The new Internet address architecture, where there is a global address realm and many private address realms interconnected by NATs, make it difficult for two nodes on different private networks to communicate to each other directly.

To make the new services based on P2P, like teleconferencing or on-line gaming, completely accessible, it's very important, and urgent, to implement a technique that consents to different protocols to traverse the NATs. The objective of this work is to study different approaches to pass the NATs and in particular: the scalability, the reliability, the mobility, the standard interfaces, and the security of these approaches. This work might give us a possible approach to solve the problem, not only on an hypothetical level but also on a practical level.

2.2 Solution Requirements

The principal requirements that the solution must have are:

- Scalability: a peer-to-peer application is intended to be open to a theoretically infinite community. A completely scalable system allows a number of instances to grow several orders of scale without any behaviour problem. The solution must satisfy this important requirement.
- Reliability: the aim is to find a solution that allows to exchange data in a reliable way. The ideal example is the normal TCP communication for data transfer.
- Standard interface: the use standard interfaces for network (e.g., Sockets) allows a simpler implementation and the use of many network packages/libraries. It's important that wherever possible, the application will use the native OS TCP stack. This is to avoid increasingly complex protocol stacks, but more importantly, because TCP stacks have been, over the years, carefully optimised for high performance and congestion friendliness.
- Multi Operative System context: the application must be compatible with the BSD-style socket API. This imposes some limitations, for instance, the API depends on the kernel to construct packets and send them on the physical medium. The application can run on various machines, like normal computers, but also on mobiles, PDAs, etc., with several operative systems like Windows, UNIX, etc.
- Security: NATs are often deployed to achieve security goals. The solution must not affect the security properties of these devices.
- Mobility: communication between peers that move from network to network. SOLIPSIS, like much other P2P applications, is conceived to be implemented on mobile terminals too, like mobile phones or PDA, and the solution must be independent from a particular NAT configuration. The implementation must don't need any NAT reconfiguration, like the additions of port mappings, permission UDP or turn on UPnP [10].

Chapter 3

Network Address Translation (NAT)

Network Address Translation is a method by which IP addresses are mapped from one address realm to another, providing transparent routing to end hosts. The need for address translation arises when a network's address cannot be used outside the network or because it is a not valid public address, or because the address must be kept private from the external network. Address translation allows transparent routing beetween two hosts in different networks by modifying end-node addresses en-route and maintaining state for these updates [20]. This chapter attempts to describe the operations of NAT devices and to define the terminology used to identify various kind of NAT and use them.

3.1 NAT Binding

To understand a NAT traversal connection the comprehension of the concept *session* is fundamental. A *session endpoint* for a TCP or UDP session is a couple (IP address, port number), and a session is identified by two session endpoints. A session can be defined as a 4-tuple (source IP, source port, target IP, target port). Address translations performed by NAT are session based and will include translation of incoming as well as outgoing packets belonging to that session. Session direction is identified by the direction of the first packet in that session.

There are two principal NAT functions that interferes with a peer-to-peer application and thus requires NAT transversal support: address translation and the filtering of the unsolicited traffic. In this chapter these functionalities will be analysed and a list of the NAT types will be presented.

3.1.1 Address and Port Mapping

In order to enable many private hosts behind the NAT to share the use of a smaller number of public IP addresses (often just one) the NAT modifies the IP-level and often the transport-level header information in packets flowing across. The users

behind a NAT have no unique, permanently IP address to use in the public Internet, and can communicate only with the temporary public address (IP and port) that the NAT assign them dynamically. Establish a communication between two peers that reside behind two different NAT (in two different private network) is very hard, because neither of them has a permanent and valid public IP address that the other can reach at any time, so in order to establish a peer-to-peer connection, the hosts must rely on the temporary public address assigned them by their NAT as a result of a prior outgoing client/server style communication sessions [23].

Discovering, exchanging, and using these temporary public endpoints requires that the two host collaborate through a well-known node on the public Internet with a protocol that allows applications to obtain external communication through the NAT.

When an internal host opens an outgoing session through the NAT, the NAT assigns an external public IP address and port number for this session. Every subsequent incoming packet from the extern endpoint can be received by the NAT, translated and forwarded to the internal endpoint. This mapping between the internal IP address and port tuple and the IP address and port is valid for all the session's duration.

For many applications, and in particular for peer-to-peer, is fundamental to know the NAT behaviour with multiple simultaneous session binding, to allow a high connectivity. The behaviour depends by the criteria used in the address mapping for a new session. Like showed in Figure 3.1 two session are established between the same internal address X:x and two different addresses in the external network. When the internal endpoint sends from address and port X:x to the external endpoint Y1:y1 to establish a connection, the NAT creates a mapping for the session identified by the tuple (X, x, Y1, y1) using the public address and port N1:n1. If the internal endpoint creates a second session with Y2:y2, this session is mapped on the NAT with the public address and port N2:n2 [22]. The relationship between N1:n1 and N2:n2 is critical and determines the NAT behaviour. The next paragraph exposes these behaviours and, consequently, the classification of the different NAT types.

NAT types for TCP communication. There are four different ways NAT boxes implement the mapping and filtering 8using the classification in [21]):

- *Independent*: all requests from the same internal IP address and port are mapped to the same external IP address and port. So the NAT reuses the mapping for subsequent session initiated from the same IP address and port (X:x). Specifically, for the mapping showed in Figure 3.1, N1:n1 equals N2:n2 for all values of subsequent endpoint Yi:yi.
- *Address Dependent*: all requests from the same internal IP address and port are mapped to the same external IP address and port only when the destination IP is the same, regardless of the external port. Specifically, N1:n1 equals N2:n2 if, and only if, Y2 equals Y1 for any port y.

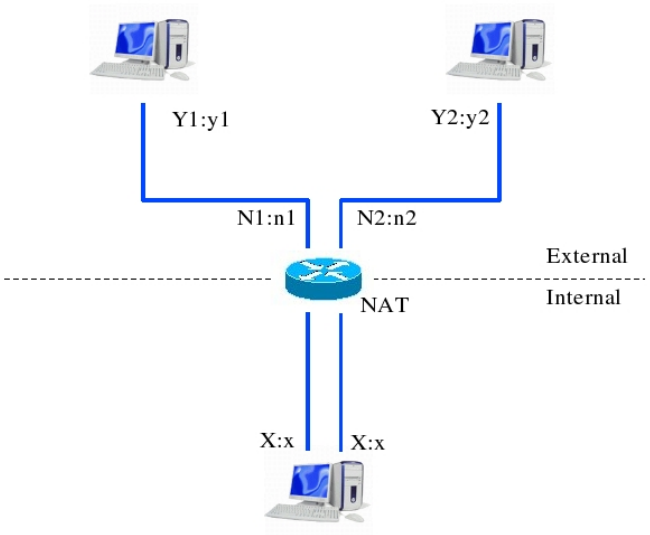


Figure 3.1: NAT address and port mapping

- *Address and Port Dependent*: it is like an Address Dependent NAT, but the restriction includes port numbers. The NAT reuses the port mapping for subsequent sessions initiated from the same internal transport address (X:x) only for sessions to the same external IP address and port. Specifically, N1:n1 equals N2:n2 if, and only if, Y2:y2 equals Y1:y1.
- *Session Dependent*: a session dependent NAT is one where all sessions from the same internal IP address and port, to the same destination IP address and port, are mapped with a different external IP address and port. Specifically, N1:n1 not equals N2:n2 even if Y2:y2 equals Y1:y1. Thus, with this mechanism, two subsequent sessions have not the same 4-tuple.

3.1.2 Port Assignment

The *port preservation* is the attempt of the NAT to preserve the port number used internally when creating a mapping to an external IP address and port.

Referring the mapping showed in Figure 3.2, if the port n is already in use and no other external IP addresses are available, the NAT switch to the Network Address and Port Translator (NAPT) mode. The *port preservation* behaviour does not guarantee that the external port n will always be the same as the internal port x but only that the NAT will preserve the port if possible.

The difference in allocation (n2 - n1) between a fresh external binding (N2:n2) when the last port allocation was (N1:n1) is called NAT's *Binding Delta*. It may be constant or random and the tests have showed that the delta value is usually 1 or 2

and rarely random. If the NAT is an Independent type, the Binding Delta value is 0. Assembling the delta value with the NAT binding types exposed previously, the NAT type obtained are: Address$_\delta$, Address and Port$_\delta$, Session$_\delta$, where the subscript 'δ' usually assumes the value of 0, 1, 2 or '$\Re$' for random.

Some NATs assign the same external IP and port to two different internal transport addresses even in the case of collision and rely the mapping from the external endpoint (Y1:y1, Y2:y2). This *port overloading* fails if the two internal endpoints are establishing sessions to the same external destination (e.g., a SIP proxy, a web server, etc).

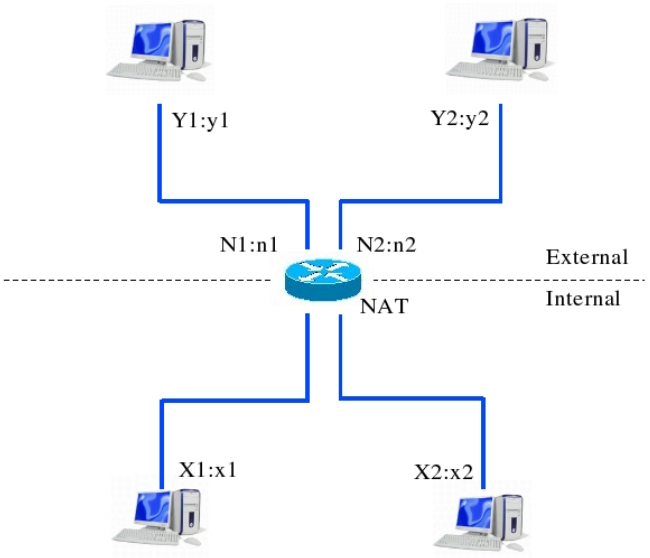


Figure 3.2: The mapping in a Network Address and Port Translator (NAPT) mode

If a NAT preserves the parity of the port, i.e., an even port will be mapped to an even port number, and an odd port will be mapped to an odd port number. Specifically, for the Figure 3.2, if the external port n, assigned for the internal port x, is even if and only if x is even, then the NAT preserves even-odd port parity. If the NAT assign address N1:n1 and N1:(n1+1) for two connection initiated from internal ports X1:x1 and X1:(x1+1) then the NAT preserves *next-higher port parity*, also referred *port contiguity preservation*. The *port contiguity preservation* can be usefull for the RTCP=RTP+1 [1] rule in the RTP protocol but this port reservation is problematic since it is wasteful and a NAT cannot distinguish in a reliable way between UDP packets where there is or not contiguity rule.

The last point for the port assignment behaviour is the binding range allocated by the NAT. The IANA defined three ranges: *well-known* from 0 to 1023, *registered* from 1024 to 49151, and *dynamic/private* from 49152 to 65515. The use of an

[1] Real Time Protocol (RTP) and Real Time Control Protocol (RTCP)

already registered port can cause undesirable effects. The NATs' behaviour for range port allocation is variable. Some NATs allocate an external port binding that is in the same port range as internal port, other NAT devices may prefer to use the dynamics range first or, possibly, avoid the actual registered ports in the registered range. Other NATs preserve the port range if is in the well-known range. [21] [22]

3.1.3 Binding Lifetime

A NAT keeps the mapping alive for an idle period. After this period the allocated binding address is released. The length of this idle period and the way the mapping timer is refreshed vary and could depend on the connection state and the NAT implementation. For a TCP connection the idle timer can be short when the connection is in the SYN_SENT or LAST_ACK state and long in the ESTABLISHED state.

The binding may be released upon receiving certain packets such as TCP RST packets or ICMP errors for a connection.

Some NATs do a mapping N:n for several sessions (X1:x1; Yi:yi). If this mapping is refreshed for all session on that mapping by an outbound traffic, the NAT is said to have a NAT *Mapping Refresh Scope* of *Per mapping*. Otherwise we have a *Per session* Mapping Refresh Scope and the mapping is refreshed only on a specific session on that particular mapping by an outbound traffic. For some connectionless protocol, like UDP, the NAT can have different *refresh direction* behaviour. If the mapping is refreshed for an outcoming packet the NAT has an *Outbound Refresh* behaviour of *true*. If the mapping is refreshed for an outcoming packet the NAT has an *Inbound Refresh* behaviour of *true*. For NAT that refresh the mapping for incoming and outcoming packets both properties are *true*.

3.2 Filtering of Unsolicited Traffic

The filtering function allows the NATs to drop incoming session that are deemed unsolicited. The most common filtering policy is to permit a communication session to cross the NAT if and only if it was initiated from the private network. All packets arriving from the public Internet, like an attempt by another peer to establish a connection, are dropped because they are not part of an existing communication session, and the NAT cannot know the destination host in the private network. Every TCP connection must be often previously initiated by a node in the private network. This is not a guarantee that the NAT will accept the inbound packet. More specifically, using the mapping showed in Figure 3.1, we can classify the NAT behaviour in four different criteria for endpoint filtering, like exposed in [21] and [22]:

- *Open*: The NAT does not filter any packets.

- *Endpoint Independent*: The NAT filters out only packets not destined to the internal address and port X:x, regardless of the external IP address and port source. Specifically, for mapping in Figure 3.1, if there is a connection between X:x and Y1:y1, mapped on NAT as N1:n1, the NAT routes the incoming SYN packets with destination N1:n1 to the internal address X:x for all values of source address Yi:yi.

- *Endpoint Address Dependent*: In this case, the NAT will filter out packets from external IP Y destined for the internal endpoint X:x if X:x has not sent packets to Y previously, regardless of the external port. In other words, as showed in Figure 3.1, if a connection exists between internal address X:x and external address Y1:y1, NAT routes incoming SYN packets to X:x, only for the Y1 IP source address but for any value of the source port y.

- *Endpoint Address and Port Dependent*: This behaviour is similar to the previous, except that the filtering depends of the external port too. The NAT will filter out packets from external endpoint Yi:yi destined for the internal endpoint X:x if X:x has not sent packets to Yi:yi previously. NAT will route incoming SYN packets to X:x, only for the Y1:y1 source address

Moreover, a NAT may filter out some TCP packets viewed as insolecited traffic even if they come from the IP and port to which a packet was sent recently. For example, if a SYN was sent to the external address Y1:y1 through the NAT that maps this connection with the address N1:n1, an incoming SYN from endpoint Y1:y1 to the destination N1:n1 may be filtered out. Like SYN, all other TCP packets from Y1:y1 to N1:n1 are filtered out, except for a TCP SYNACK or TCP RSTACK.

There are different NAT's reactions to an unsolicited packet. The NAT may generate an ICMP error or TCP RST packet to inform the sender, or simpler, drop the packet silently. The behaviour, in front of an unsolicited packet, is important for the NAT traversal, specially if we attempt to establish a TCP connection between endpoint behind two different NATs. An ICMP error or a TCP RST packet may release the mapping for a particular tuple (X:x, Y:y) as explained in section 3.1.3. An estimated 93% of the NATs simply dorp silently the inbound SYN.

3.3 NAT Mapping and Filtering for UDP Communication.

The terminology presented for NAT behaviours is not globally accepted, some other definitions can be found and is valid only for TCP protocol. For UDP, as there are not session establishment, the filtering and the mapping is not extremely different but the terminology is not the same. The four treatments observed, like reported in STUN documentation [16], are:

- *Full Cone*: all requests from the same internal IP address and port are mapped to the same external IP address and port and any external host can send a packet to the internal host, by sending a packet to the mapped external address.
- *Restricted Cone*: 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 Y) can send a packet to the internal host only if the internal host had previously sent a packet to IP address Y.
- *Port Restricted Cone*: the implementation is like in a restricted cone NAT, but the restriction includes port numbers. Specifically, an external host can send a packet, with source network address Y:y, to the internal host only if the internal host had previously sent a packet to IP address Y and port y.
- *Symmetric*: all requests from the same internal IP address and port, to a specific destination IP address and port, are mapped to the same external IP address and port. If the same host sends a packet with the same source address and port, but to a different destination, a different mapping is used. Furthermore, only the external host that receives a packet can send a UDP packet back to the internal host.

The equivalence with the NAT behaviour in presence of TCP is reported in Table 3.1

UDP NAT type	**TCP Binding**	**TCP Endpoint Filtering**
Full Cone	Independent	Independent
Restricted Cone	Independent	Address Dependent
Port Restricted Cone	Independent	Address & Port Dependent
Symmetric	Session	Address & Port Dependent

Table 3.1: Equivalence for NAT Beauvoir with UDP and TCP

3.4 Hairpinning Behaviour

If two host are behind the same NAT, or behind multiple levels of NATs, and want to exchange traffic, a NAT that supports hairpin can route the traffic coming from an internal address, translating the external destination address to the internal value. Specifically, following the mapping in Figure 3.2, if the internal host X1:x1 sends packets to the external address N:n2, that is an active mapping for the internal address X2:x2, the NAT routes the packets to X2:x2 with source address either the original internal source address X1:x1, or translates it to the external mapped address N:n1. A non-translation of the internal source address to the external mapped address may cause problems to applications that expect an external IP address and

port. Section 4.6.1 shows some examples where NAT hairpin is fundamental for communication.

Many NAT do not yet support hairpin translation, and the message sent from X1 to X2 is dropped, but hairpin is more common as NAT vendors become aware of this issue.

3.5 Packet Manipulation

A NAT may manipulate packets by modifying bytes that not need to be changed for successful operation or by not changing bytes that should be changed. In particular, a NAT may change the IP header of the packet traversing the NAT: modifying the TCP sequence number adding a random offset, and subtracting the same offset from the acknowledgement numbers in incoming packets; changing the time-to-live value to something different of its value decremented by one. It may even manipulate payloads by translating bytes that look like the internal or external transport address encoded in the message payload. It may not correctly translate IP packets in the payload of ICMP packets. These modifications can cause problems in a TCP connection attempt that tries to traverse NAT.

Classification	**Values**
NAT Binding	Independent Address$_\delta$ Address and Port$_\delta$ Session$_\delta$
Endpoint Filtering	Open Independent Address Address and Port
Response	Drop TCP RST ICMP
TCP Seq#	Preserved Not Preserved

Table 3.2: NAT behaviour classification

3.6 NAT Classification

The different categories distinguishing various NATs are reported in the Table 3.2. Table 3.3 shows the data about the different percentages of the NAT distribution on the market [6]. The proportions for the various binding behaviour, previous

presented, show that the majority is Independent, but the most important data is the percentage of the NAT with a random delta value: there are practically no NAT with a random binding delta. It is for this reason that we have focalised our attemption to traverse just the NAT with a not random delta value.

NAT Binding	**%**
Independent	75.9%
Address_1	2.5%
$\text{Address and Port}_1$	18.7%
Session_1	2.9%
$\text{Address}_{\Re}$	0.0%
$\text{Session}_{\Re}$	0.0%

Table 3.3: NAT binding types observed on the worldwide market

3.7 IETF guidelines: NAT Behavioural Requirements

NATs have been created to solve the exhaustion of IPv4 addresses, like the new standard IPv6. Unfortunately, the commercial world has not embraced the IPv6, conceived and promoted by the Internet Engineering Task Force (IETF) and Internet Architecture Board (IAB). This solution come to solve definitively the critical problem of the IP addresses' exhaustion. Like exposed in [7], IETF discouraged the use of NATs for years, seen just as a short term solution, and standardised many protocols that did not work through NAT (e.g., mobile IP and SIP). No guideline for NAT implementation have been developed, and this make difficult to deploy P2P protocols. Some guidelines have been lately published (e.g., [22]) to standardise the next generation of NATs, and allow a good support for P2P based applications.

3.8 The NAT security behaviours

A critical point on NAT specification is security. Restricted filtering ostensibly has the benefit of protecting applications on a internal host from traffic the internal host is not expecting. This behaviour interferes with application and it seems that provides very little real security benefits. IETF is working on a standard for NATs, to increase the predictability and compatibility with applications. The main argument is the high restrictive filtering policies proposed by IETF, that interferes with applications and seem a firewall standard more than a Network Address Translation behaviour.

Chapter 4

NAT transversal: some general approaches

The basic problem with TCP through NATs is that, in a normal situation, there is an endpoint that listens for incoming connection and another that initiates the connection. Unfortunately, with NAT presence, the TCP session requires an initial outgoing packet to assign the NAT port mapping before any incoming packets can be received. Popular P2P applications have faced the NAT traversal problem in different ways.

Some solutions suggest a re-configuration of NAT to forward incoming request on specific ports (e.g, Kazaa), that is not acceptable because several computers behind the NAT may run the same application and user may not have administrative access or technical capability to configure the NAT. Another solution is to route the traffic through central servers (e.g., BitTorrent) that results quite expensive and is not scalable or through enduser instances of the application that are non-NATed (e.g., Skype) [6]. These solutions don't satisfy our solution requirements. Other approaches came out recently. They are exposed here in this section. The chapter presents other methods that have been used to build the final approach.

4.1 TCP over UDP

A possible solution for applications to transfer data over UDP, through the NATs, is to re-engineer TCP/IP stack encapsulated inside UDP datagrams protocol. This was the initial solution for problem and the object of this work. A carefull analysis has revealed the problems with the application of this solution. This approach, in fact, incurs overheads introduced by encapsulation, decapsulation and transmission of extra header data. The main problems are that the UDP packets will be handled at the application level, that requires a context switch, with possible loss of packets. Every segment's dispatch requires a read/write operation, and thus the context switch, while TCP can do large write reducing OS overhead. Additionally, the

timeouts and ACK posting could be less efficient than a kernel implementation. Moreover, it have to be considered, that adding an application layer over the re-engineered TCP, will require multi-threading, flow control and buffer management, that could reduce the effective througput. These and others inefficiencies brought to look for a better solution. [21] [2]

4.2 UPnP and MIDCOM

The Universal Plug-and-Play (UPnP) [10] and the MIddlebox COMmunication (MIDCOM) architecture offer pervasive peer-to-peer network connectivity. The architecture is a distributed, open networking architecture that leverages TCP/IP and the Web to enable seamless proximity networking in addition to control and data transfer among networked devices. They are a new standard to support the ability to plug devices into a home network. An application can configure a NAT box to forward back the incoming request received on specific ports. Unfortunately, there are high security problems and many NAT boxes turned off these protocols. Furthermore, WiFi hotspots tend not to enable them. In general, an application developer cannot depend on the existence of these protocols or that they are enabled in the NAT. Something more deployable is preferable. Nevertheless, UPnP support can be easily added to the solution proposed in this work.

4.3 IPv6

IPv6 is the new standard that specifies 128 bit IP addresses. It is possible that the new longer IPv6 addresses will reduce the need for a NAT box. However, as the adoption of the IPv6 addresses is slow, the NAT boxes are used to bridge between IPv4 and IPv6 networks. In addition, due to the critical importance of network security, firewalls with a policy of filtering unsolicited incoming traffic are likely to remain commonplace even after the transition to IPv6.

4.4 STUN: Simple Traversal of UDP protocol Through NATs

STUN is a protocol that allows users to discover if there is a NAT between them and the public Internet and, in this case, the type of the NAT. It allows to discover if there is an UDP firewall too. With STUN, a peer-to-peer application can determine the public network address allocated to it by the NAT, and require no special NAT behaviour and no changes to NATs. The STUN's objective is to provide a mechanism for NATs traversal and allows applications to work through the existing NAT infrastructure. This section resumes the STUN protocol defined in RFC 3489 [16]

4.4.1 Terminology

In the STUN protocol there are two principal entities: a STUN client, that generates the STUN requests, and the STUN server that receives the STUN requests, and send STUN responses.

4.4.2 Overview of operations

To know behind what type of NAT the client is and to discover his public network address, it starts the STUN discovery procedure sending the initial *Binding Request* to server using UDP. A *Binding Request* that arrives to the server may have traversed one or multiple NAT levels, and it source address will be the mapped address created by the NAT closest to the server. The server puts this address in a *Binding Response* message and sends it back to the source IP address and port of the request. The client compares the local IP and port with these send by the server. If the addresses are the same, between the client and the open Internet there are not NAT boxes. In the case of a mismatching one or more NATs are present.

If the NAT is a full-cone, the IP and port mapped by the NAT are public and any host can use them to communicate with the user behind the NAT. But the user is not sure that the NAT is a full-cone and if everybody can join it on this address. For this reason, it's necessary to follow in the STUN discovery procedure.

The client sends a second *Binding Request* from the same IP and port address but to a different IP address, to a second STUN server that reply with the public IP and port address that NAT mapped for this communication. If they are different from those in the first *Binding Response*, the client knows it is behind a Symmetric NAT. Otherwise, the client just knows that the NAT is not a Symmetric type and it must continue the discover procedure.

It sends a *Binding Request* with flags that tell the server to send a response from a different IP address and port than the request was received on. If the client receives this response, it knows it is behind a full cone NAT. Otherwise the NAT can be a

port restricted cone NAT or just a Restricted Cone NAT.

To discover it, the client ask to STUN server, to sending a *Binding Response* from the same IP address than the request was received on, but from another port. If a responce is received, the client is just behind a Restricted NAT. The Figure 4.1 shows the flow for the type discovery process. The different types are be explained in section 3.1.1.

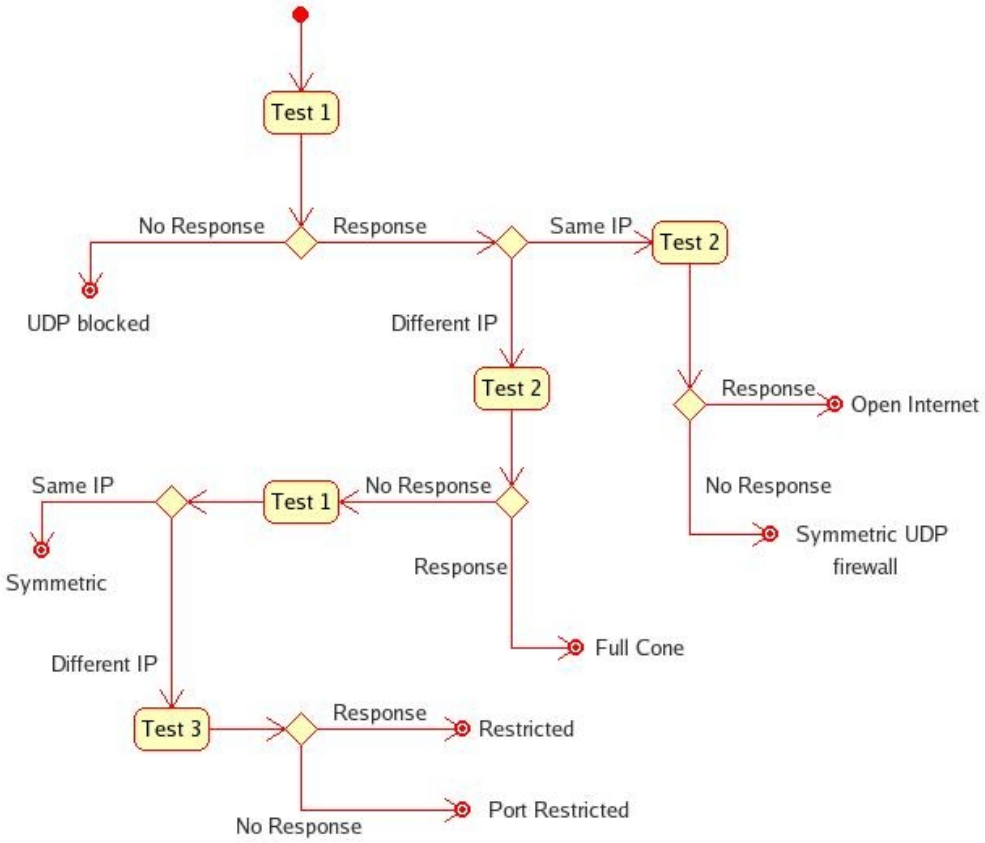


Figure 4.1: STUN - Flow for for type discovery process

4.5 STUNT: Simple Traversal of UDP Through NATs and TCP too

The STUN protocol allows applications to discover the presence and type of NAT and learn the binding allocated by the NAT without requiring any changes to the NATs themselves. However, STUN is limited to UDP and does not work for TCP. The behaviour of NATs when it comes to TCP is more complex.

The protocol described here, Simple Traversal of UDP Through NAT and TCP too (STUNT), extends STUN allowing applications behind a NAT to discover the NATs behaviour for TCP packets and to learn the transport address binding allocated by the NAT. Using STUNT, applications can establish raw TCP sessions with other NAT'ed hosts without using encapsulation, tunneling, relaying or a userspace stack, and without forfeiting any of benefits of using TCP. Like STUN, STUNT requires no changes to NATs and works with a large majority of NATs currently present in the network architecture.

4.5.1 Overview of operations

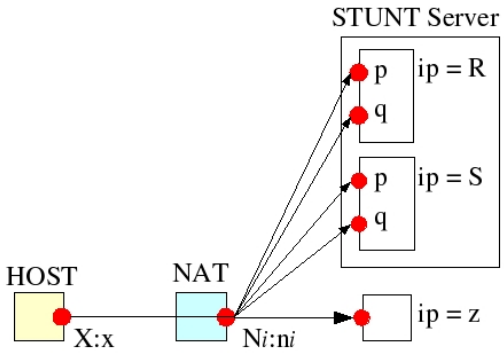


Figure 4.2: STUNT - binding behaviour

A STUNT client contacts one STUNT server multiple times to check if the binding changes. If it does not then the client checks if the binding changes for a different server at the same IP address but different port, and finally whether it changes for a third server at a different IP address and port. Once the NATs binding behaviour is known the client can find or predict the binding for new connections through a simple algorithm.

A client typically also learns the filtering behaviour of the NATs to decide between TCP Traversal techniques. It does so by testing the NATs response to incoming TCP

SYN packets in various situations.

The Figures 4.3 and 4.4 show how the client can accomplish useful tasks to discover the binding and endpoint filtering behaviours.

Binding Behaviour: In the Figure 4.3 there are the operations to discover the NAT's binding behaviour.

The client initiates Test 1, where it establish a TCP connection to the server address R:p, bound to local address X:x (Figure 4.2). It sends a *Binding Request*. The server puts the source address seen N1:n1 in a *Binding Response* message and sends it back to the source IP address and port of the request. The client compares the local IP and port X:x with these send by the server. If the addresses are the same, between the client and the open Internet there are not NAT boxes.

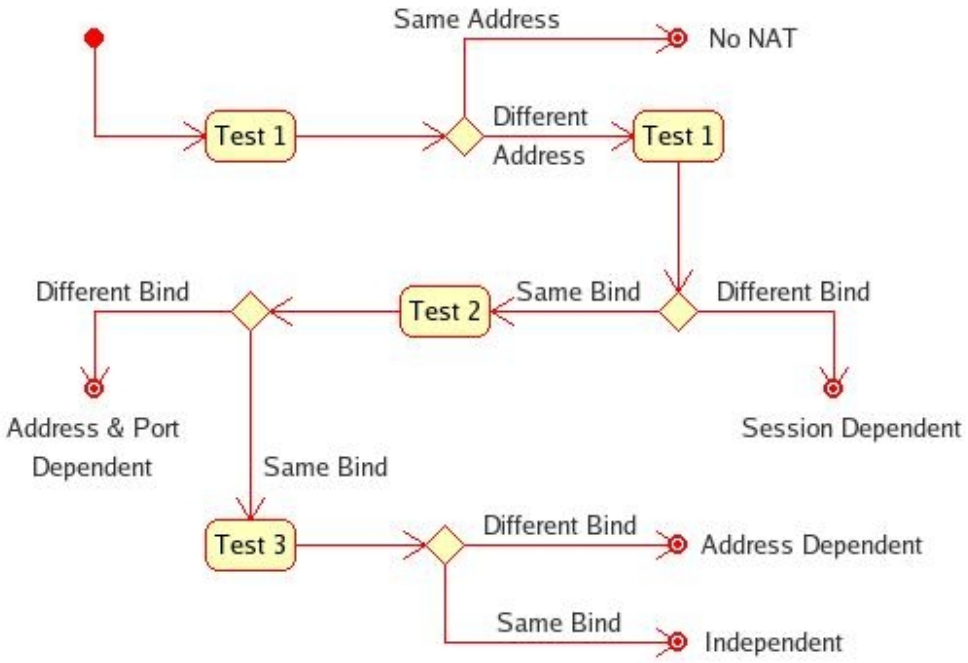


Figure 4.3: STUNT - Binding behaviour

In the case of a mismatching one or more NATs are present. The client then repeats Test 1 and compares the address N2:n2 sent by server with N1:n1, if they are different the binding is session dependent.

Otherwise the client start Test 2 that is similar to Test 1 but connects to server

R:q. It learns its binding, call it N3:n3. If N3:n3 is not the same as N1:n1 the binding behaviour is Address and Port Dependent.

If the mapping did not change Test 3 is executed, which is like Test 1 with S:q as the server address. If the new binding is different from N1:n1 then the binding behaviour is Address Dependent, otherwise it is independent of the destination.

Endpoint Filtering Behaviour. To know the endpoint filtering behaviour the client executes the tasks in Figure 4.4.

It establish a TCP connection with STUNT server and initiates Test 1 where the server sends an initial SYN from an alternate IP and port to the client's IP but to a random port. If the SYN is received by the client then the NAT has a Open filtering behaviour.

Otherwise Test 2 is done, where the client establish a TCP connection with the server, send a *Packet Request* message. Then the server replies sending a SYN packet from a different address and port. If the SYN is received then NAT allows any host to contact the client once a mapping is established, thus the NAT has an Endpoint Independent behaviour.

If the test fail, the client initiates Test 3 which is identical to test Test 2 but the SYN is sent by the server from the same address and a different port. If the client receives the packet then the NAT filter is Address Dependent, otherwise Address and Port Dependent.

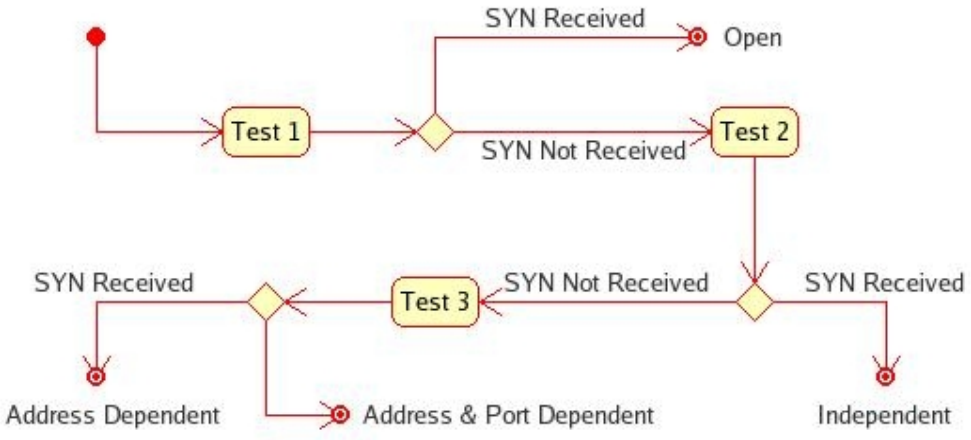


Figure 4.4: STUNT - Endpoint filtering behaviour

STUNT Improvements. These tasks, however, don't allow to discover the configuration where the NAT doesn't accept incoming SYN at all, even if a previous

SYN has been sent to the destination address and port. It's for this reason that is suggested a STUNT variation, adding a fourth test, where the client establish a TCP connection with server to communicate with it and send a SYN packet to an alternate port k on STUNT server. The packet must have a low time-to-live and to not reach the server. An ICMP error occurs. At this point the client asks to the server to send a SYN packet from port k. If the client receives the packet then the NAT filter is Address and Port Dependent, otherwise the NAT doesn't accept incoming SYN. The STUNT protocol allows to perform various tests combining arranging the STUNT messages in different ways. Other tasks can be implemented to test the NAT reaction to an incoming ICMP message or RST error. These test could be usefull to better choose the right method to be used to traverse the NAT.

4.6 Hole Punching

4.6.1 UDP Hole Punching

This approach enables to establish a direct UDP session between two nodes, even when both may lie behind different NATs, with the help of a well-known third machine, the *connection broker*. A detailed description can be found in [3].

The connection broker must have a public address to allow the NAT'ed peers to contact it and helps them to discover the other peer's public address. UDP hole punching relies on the properties of common firewalls and NATs to allow peer-to-peer applications to *punch holes* and establish direct connection with each other. This technique works good with UDP, but it will be seen that, with some modifications , it can also work with TCP. In this section will be shown how hole punching works, because it will be used to implement the entire architecture for TCP communication across NATs.

Here are shown three specific scenarios, and how applications can be designed to handle all of them rightly. In the first situation, representing the common case, two nodes desiring direct peer-to-peer communication reside behind two different NATs. In the second, the two peers reside behind multiple levels of NAT and the last scenarios shows two clients that reside behind the same NAT, but do not necessarily know that they do.

Peers behind different levels of NATs: in UDP hole punching, the external port mapped by each NAT is learned by a third party, the server S in the Figure 4.5, assisting the connection attempt.

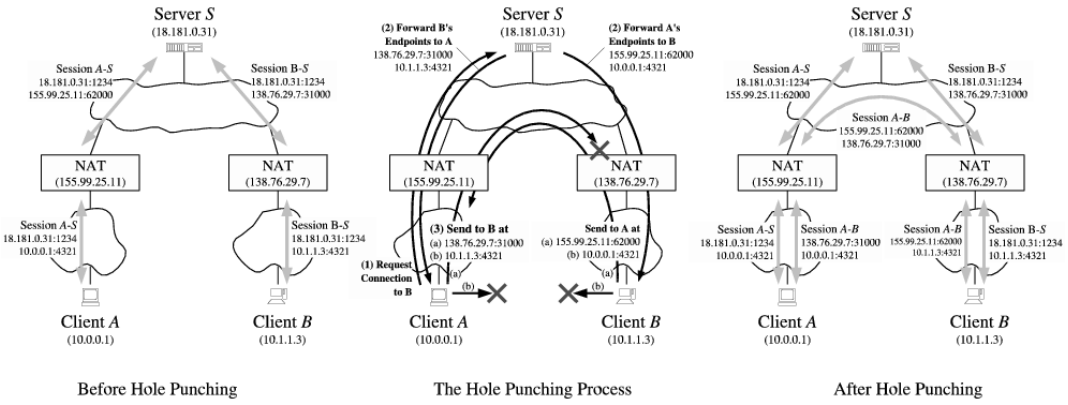


Figure 4.5: UDP Hole Punching: peers behind different NATs

Both parties, client A and B, are registered to the server. Client A, that want to communicate with B, in UDP, send a request to the server to learn its NAT mapping, client B's NAT mapping, and inform B. Both parties behind NATs, send

a UDP packet to the correct external port of the other peer. This create the necessary NAT port mappings. Once the first message from A and B have crossed their respective NATs, holes are open in each directionand UDP communication can proceed normally.

A and B try to send messages to their respective private addresses too. The message sent to these endpoints will reach either the wrong host or no host at all.

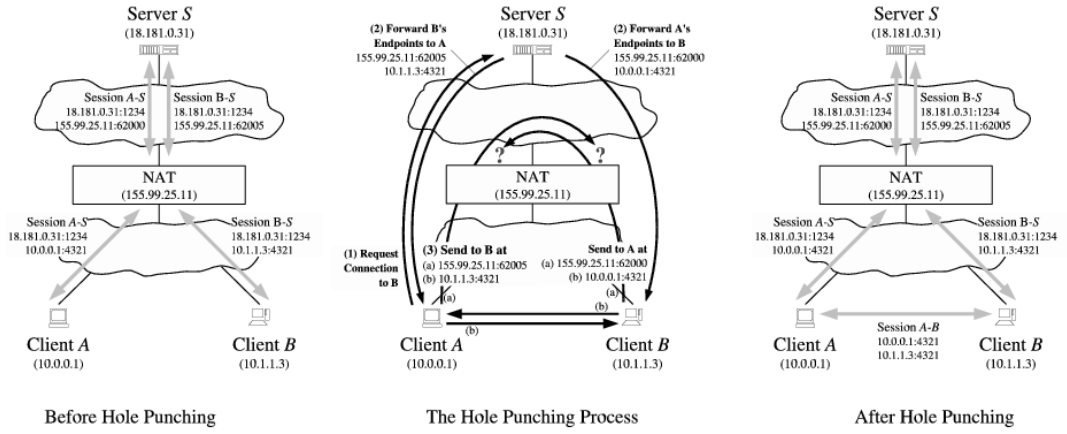


Figure 4.6: UDP Hole Punching: peers behind the same NAT

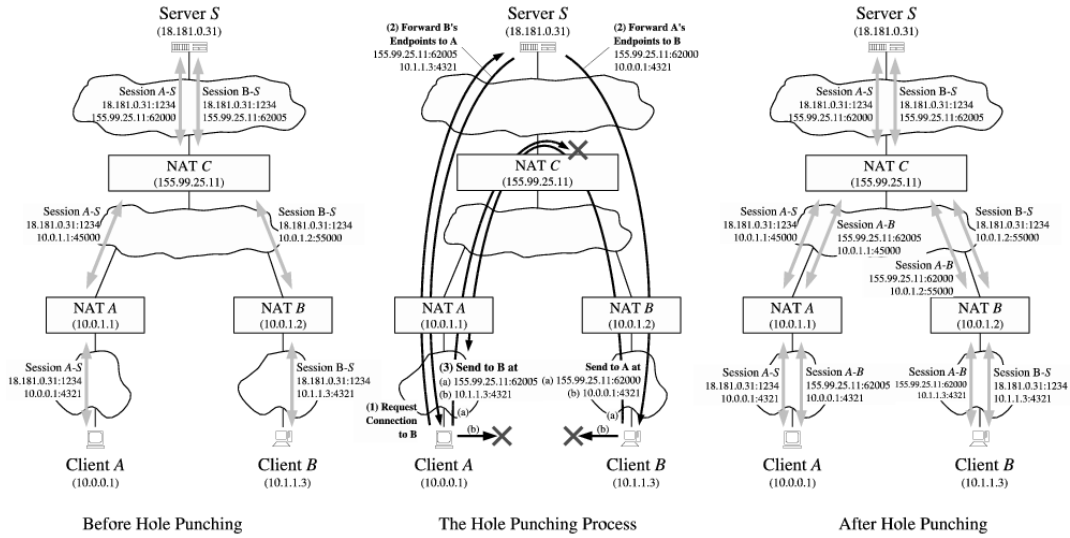


Figure 4.7: UDP Hole Punching: peers behind multiple levels of NAT

If the clients reside behind the same NAT, as the scenario in Figure 4.6 shows, the messages directed at the private endpoints reach their destination. The message

directed to the public endpoints may or may not reach their destination, depending on whether or not the supports *hairpin* translation as described in section 3.4.

In some topologies involving multiple NAT devices, two clients cannot establish an *optimal* p2p route between them. The clients have no choice but to use their global public addresses as seen by server, and rely on NAT C, Figure 4.7, providing hairpin translation.

4.6.2 TCP Hole Punching

To establish a peer-to-peer TCP session is necessary to use a more refined approach, although this approach is very similar to the UDP hole punching. In a traditional TCP connection one party must be the initiator (send the initial SYN packet) while the other listen for the initiation. The listening peer will be prevented from seeing the initial SYN because the SYNs will dropped at the listening peer's NAT. So, in order to establish a TCP connection between two hosts behind NATs, each NAT must believe that the connection is initiated from the endpoint in its private network. To obtain this, some approaches are proposed in the next chapter.

Chapter 5

The Solution

Various techniques have been developed for traversing NAT/firewall with UDP, because of the difficulty to establish a TCP connection. This difficulty is due to the asymmetric nature of the TCP protocol. In this chapter will be explained different approaches that have been formulated, implemented and tested to arrive to the final solution.

Independently, other authors had worked on TCP connectivity through NAT and had similar results. Severals approaches developed by other authors are exposed in chapter 4, and we have centralised our attemption on four particular solutions: the NUTSS framework [5], a SIP-based approach to UDP and TCP network connectivity, the NatTrav solution [8], based on standard communication interfaces and stacks, the NATBlaster method [9], a TCP connectivity technique avoiding spoofing, and the P2PNAT [3], a peer-to-peer communication across NAT that takes advantage of the simultaneous open scenario defined in the TCP specifications [17].

5.1 Network Architecture

The Figure 5.1 shows the net architecture implemented in the different approaches. Like explained in chapter 2.1, the problem comes up when two endpoints want to establish a TCP connection and both are behind a NAT box. As shown in this chapter we can use a third-party globally reachable in the public Internet, the Connection Broker (CB). The Connection Broker concentrates functionalities like (i) discovering NAT configuration and mapping through the STUN/STUNT protocols, (ii) allowing an UDP connectivity between endpoint using the Hole Punching protocol and (iii) facilitating the TCP connection establishment through the NATs. The Connection Broker can be replicated to allow availability and scalability. We can have, thus, more than one CB and every endpoint involved in a connection could be connected to a different CB. A CB connectivity and information integrity is required in the solution implementation.

In the SOLIPSIS environment, such as in every peer-to-peer world, each node in

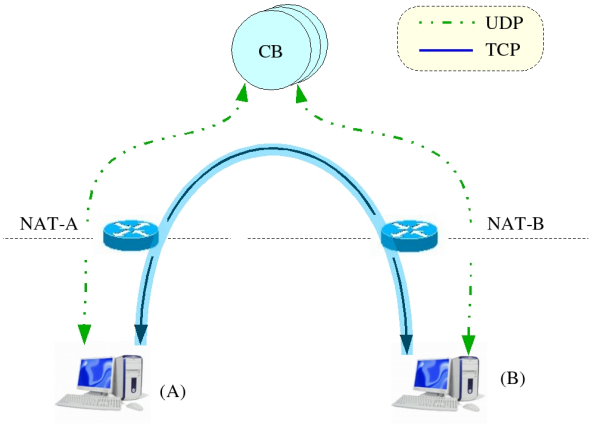


Figure 5.1: Network Architecture

the peer-to-peer network, that has valid public IP address, can be a potential CB, or, more generally, a Super Node to help other NATed endpoint to establish a NAT traversal connection.

5.1.1 Experiment Setup

A test implementation has been developed to test the several methods exposed in this chapter. A complete description of the code implemented is available in the Chapter 6. As show in Figure 5.1, the clients are behind a NAT, while the Super Node is in the public network. For tests, the STUNT server and the Connection Broker were on the same machine, with multiple IP addresses. This machine was a SUN with a Linux system. The clients have been tested on a PC either with Windows 2000, either with Linux. The NATs used were Linksys router, specifically, Wireless-G VPN Broadband Router. These NAT presented some common characteristics for TCP traffic, like *independent* binding behaviour and doesn't accept at all incoming SYN.

5.1.2 Puncher Protocol

All the methods presented in this chapter need a previously UDP data exchange to recuperate all the required endpoints' information. To establish an UDP traversal data communication, we have implemented the Hole Punching approach, exposed in the section 4.6.1. The Figures 5.2, 5.3 and 5.4 show the message sequences to allow UDP communication between endpoints and Connection Broker.

Specifically, the Figure 5.2 shows how the host A, register itself to Connection Broker. At this point, host A is reachable in UDP by Connection Broker, until the UDP session is maintained alive by endpoint with *Keep Alive* messages.

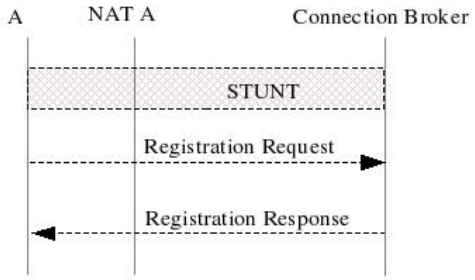


Figure 5.2: Registration to the Connection Broker

If endpoint A wants to establish a TCP connection with another endpoint B, it sends a *Connection Request* to Connection Broker, that contacts the other endpoint, waits for response and sends it to initiator (Figure 5.3).

For a direct communication between A and B, a hole punching procedure is started with a *Lookup Request* to know the NAT UDP mapping for B, to which will be sent several *UDP Hole Punching* message until a *UDP Hole Punching* response message (Figure 5.4). From this point, A can contact directly B, with a *Connection Request* message.

If a UDP communication between endpoints is given, the procedure is reduced to the *Connection Request* and *Response* (messages 6 and 7 in Figure 5.4).

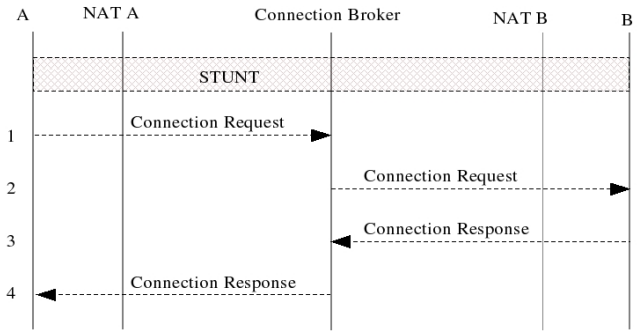


Figure 5.3: Hole Punching Protocol through Connection Broker

We will refer to all these tasks as Hole Punching task, like shows Figure 5.5

After the hole punching achievement, has been hold all the informations to start the TCP NAT traversal method. The next section discuss the different possible approaches to establish the TCP connection.

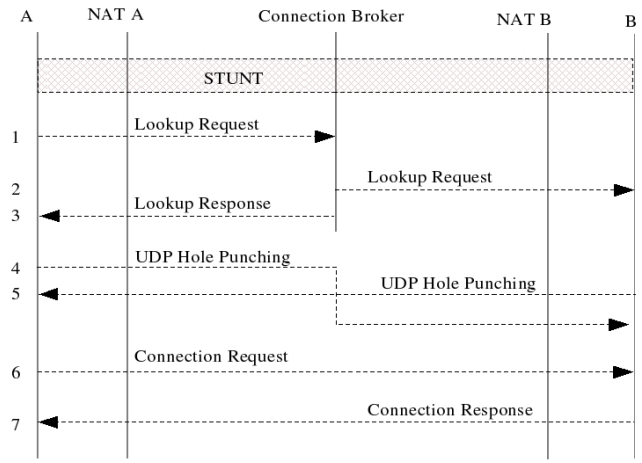


Figure 5.4: Hole Punching Protocol for directly UDP communication between endpoints

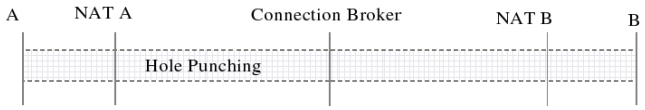


Figure 5.5: Hole Punching Protocol

5.2 TCP NAT Traversal Methods

This section exposes the various approaches implemented in the final solution. For every approach, a message sequence is presented. The Figure 5.6 shows the convention used in these message sequence.

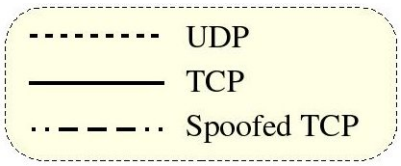


Figure 5.6: Convention used in the message sequence

5.2.1 Behaviours Observed by NAT and Application

What the NAT observes and how it reacts with the different approaches during TCP connection attempts depends on the timing, TCP implementation and NAT

configuration. The NAT can see different inbound and outbound packet sequences and, for every one of them, it can reacts in different modes. The different sequences observed for the tests are tied to the endpoint filtering behaviours of the NAT exposed in paragraph 3.2.

Sequence	Filtered
outbound SYN - inbound SYNACK	0%
outbound SYN - inbound SYN	16.1%
outbound SYN - inbound ICMP - inbound SYNACK	4.5%
outbound SYN - inbound ICMP - inbound SYN	22.2%
outbound SYN - inbound RST - inbound SYNACK	17.1%
outbound SYN - inbound RST - inbound SYN	27.9%
outbound SYN - outbound SYNACK[2]	-

Table 5.1: Packet sequences and percentage of NATs not accepting them

- *Inbound SYN*: a SYN packet used to initialise a TCP connection is consired by the major number of NAT an unsolicited traffic and it isn't forwarded to the internal endpoint, that usually has no globally valid public address. In accord with to the NAT classification in paragraph 3.2, only the *Open* type allows to a SYN packet to traverse the NAT and enter in the local area network (LAN). More important, in this case, is to consider the NAT reaction to an unsolicited SYN packet, because, like showed in Figure but5.1, NAT-B message reply to a SYN packet generated by endpoint A, will be an incoming packet for NAT-A. NAT-B may reply with a TCP reset response (RSTACK), or an ICMP unreachable error, or may drop the packet silently and a timeout will be raised.
- *Low time-to-live value*: the initial SYN packet may have a low time-to-live to avoid the destination's NAT response. In this case an ICMP error is sent by network between NAT-A and NAT-B. If the time-to-live is not reduced, we will be in the previous situation where the destination's NAT will see an inbound SYN. Several tests have showed that the majority of the NAT, to an inbound SYN, replies dropping the packet silently. It for this reason that an approach that leaves the time-to-live unchanged is more effective [1].
- *Inbound ICMP error and reset response RSTACK*: an inbound packet, like ICMP error or RSTACK, may cause the release of the mapping on the NAT or reducing the binding lifetime.
- *A not normally packet sequences*: various non-standard sequences, as an outbound SYN followed by an inbound SYN, or an outbound SYN followed by

[2] There are no statistical results for this sequence at the moment.

[1] In our implementation, we choose to leave the time-to-live unchanged to obtain better results. A future improvement can make the method's chosen adaptable with the NATs to be crossed

outbound SYNACK, may be seen as not normally. The NAT filtering behaviour may filter out this messages and, additionally, release the mapping for the addressee endpoint.

The various packet sequences and the percentage of NATs not accepting them is showed Table 5.1 [6]

5.2.2 STUNT-2 and P2PNAT Methods

The STUNT-2 method is the simplest to implement, but it is the most effective and doesn't require many user issues.

The message sequence in Figure 5.7 (a) shows this approach (proposed in [21] and [8]). An endhost A sends out a SYN packet (message 1); this packet is blocked on the other side by the NAT-B that sees it as an unsolicited traffic. The sender A than abort the connection attempt and creates a passive TCP socket on the same address and port. The endpoint B then initiates a regular TCP connection sending a SYN packet (messages 2, 3 and 4).

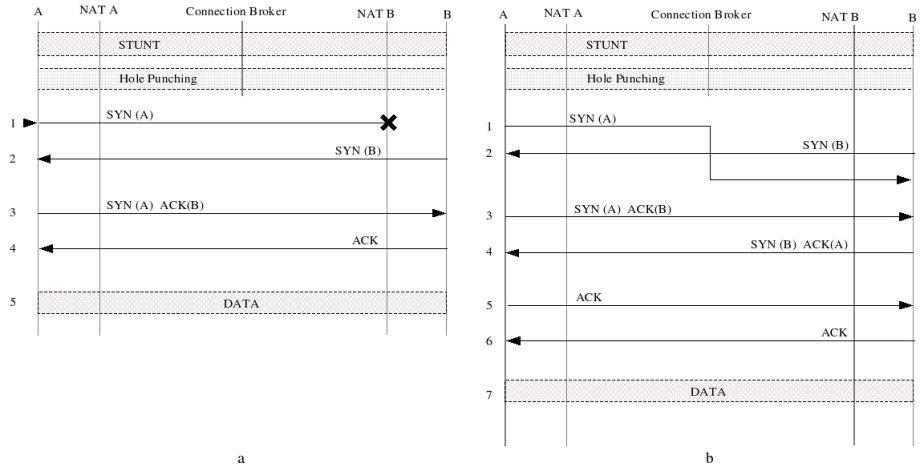


Figure 5.7: The (a) STUNT-2 and (b) P2PNAT methods

Method's issues(STUNT-2). The NAT must accept the sequence outbound SYN followed by an inbound SYN that is not a normal sequence seen in a TCP client/server connection, so the NAT may block it. Moreover, NAT-B can reply with an ICMP unreachable error or with an RSTACK packet that may bring NAT-A to not accept the following incoming SYN or, definitively release the session's mapping. If, instead, the SYN packet reaches the destination host, it could trigger unforeseen transitions, that could be favourable if if it triggers a TCP simultaneous-open or it may be determinant if it confuses the stack or NAT.

Implementation's issues. Because standard sockets APIs usually associate TCP sockets with individual TCP sessions rather than with a local port as with UDP, the application would have typically open multiple TCP sockets - one listen socket and one or more connect-sockets - and explicitly bind them to the same local port, using a special socket option usually named SO_REUSEADDR. This option indicates that the rules used in validating addresses supplied in a *bind* call should allow reuse of local addresses, except when there is an active listening socket bound to the address.

It is not possible, thus, to obtain a simultaneous client and server bound on the same local address. In our implementation we have initially bound a client to send the initial SYN packet (message 1), subsequently close the socket and opening a new socket to listen forthe incoming SYN (message 2). We have founded our implementation on the NAT behaviour to keep alive the mapping for the session for several seconds (approximately 56 seconds) in the SYN_SENT state (see section 3.1.3).

The second method presented in this paragraph, called P2PNAT, is a variation of the first approach to reduce the NAT rejection of the sequence of an inbound SYN following an outbound SYN. It takes advantage of the simultaneous TCP connection specified in [17].

As the Figure 5.7 (b) shows both endpoints, A and B, send a SYN packet, initiating a TCP connection (messages 1 and 2). Normally, if the two endpoints are well synchronised, the outcoming SYN packet cross the NAT before that the incoming SYN packet reaches the same NAT. Specifically, like showed in Figure 5.7 (b), the SYN(A) crosses the NAT-A before that the SYN(B) reaches the NAT-A, because the distance between A and NAT-A is smaller than distance between NAT-A and NAT-B. When one end's SYN arrives at the other end's NAT it can be dropped for two reasons. The first can be because it is arrived before the other end's SYN leaves that NAT. The second reason is that NAT not accept at all an incoming SYN, after an outcoming SYN too. The first endpoint's stack ends up followinf TCP simultaneous open while the other side follows a normal open. In this case the message sequence is like the first method presented before, the STUNT-2 method.

Method's issues. This approach requires that at least one NAT accepts the sequence of an inbound SYN after an outbound SYN. Moreover, it requires a tight synchronisation between endpoints and no drastic actions if RST packets or ICMP errors are received coming from destination's NAT. If, instead, a NAT responds with a TCP RST packet to an inbound SYN, this approach devolves in a packet flood until the timeout expires. In addition, not all the operative systems support the TCP simultaneous open, like Windows XP SP1 and earlier

5.2.3 STUNT-1 Method with Spoofing

With this method, parallelly proposed in [5] and [21], we try to establish a TCP connection traversing NATs that not accept the sequence outbound SYN - inbound SYN. Both endpoints, A and B, send an initial SYN (message 1 and 3 in Figure 5.8), that is blocked on the other side as an unsolicited traffic.

The endpoint learn the initial TCP sequence number by listening for the outbound SYN over a RAW socket. Both endpoints communicate these numbers to a globally reachable Connection Broker, CB, (messages 2 and 4). The CB can create and send the SYNACK response for A and B, spoofing the source address (messages 5 and 6). The ACK messages complete the TCP three way handshake (messages 7 and 8).

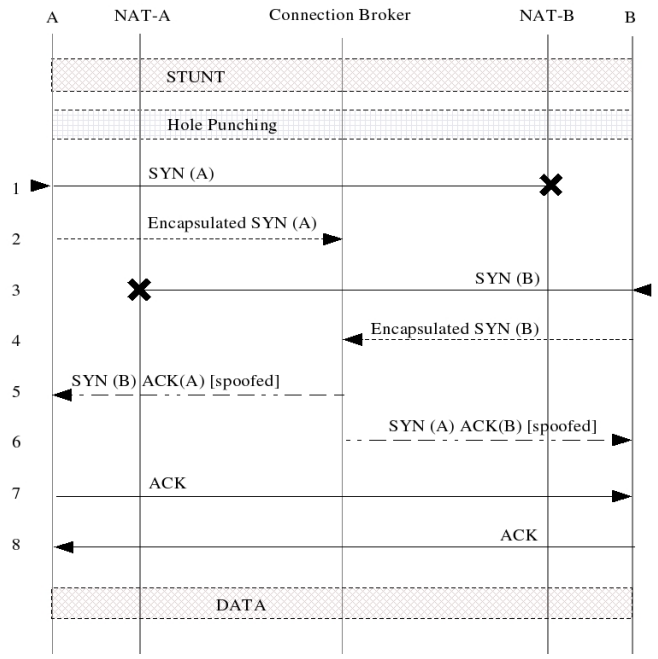


Figure 5.8: The STUNT-1 method's message sequence

Method's issues. The NAT may change the TCP sequence number of the initial SYN such that the spoofed SYNACK based on the original sequence number appears as an out-of-window packet when it arrives at the NAT. Moreover, it requires a third-party to spoof a packet from an arbitrary address, that can be dropped by network filters. Statistics show that more than 70% spoofed traffic is detected by network controls. Both the NATs may drop the inbound SYN silently, or reply with an ICMP error or a RSTACK packet that may compromise the connection establishment. It needs, in addition, to open a RAW socket on endpoints' machine, which requires

superuser privileges on that machine.

Implementation's issues. In this approach we need to listen for outbound packet. A normally RAW socket catches only inbound packet. For this reason, in our implementation, we have used the *libpcap* [29] [30], that allows applications to capture and transmit network packets bypassing the protocol stack.

5.2.3.1 STUNT-1 with Spoofing from Well-Known Machine

To limited the spoofing detection problem, this method comes as a variation of the previous method that requires spoofed packets to be injected from Connection Broker to any endpoints. The message sequence is similar to that showed in the STUNT-1 approach, with an adjunctive step to choose the right machine that can spoof packet without network detection. In this approach, the Hole Punching session allows a directly UDP session between endpoints [2], like showed in Figure 5.4.

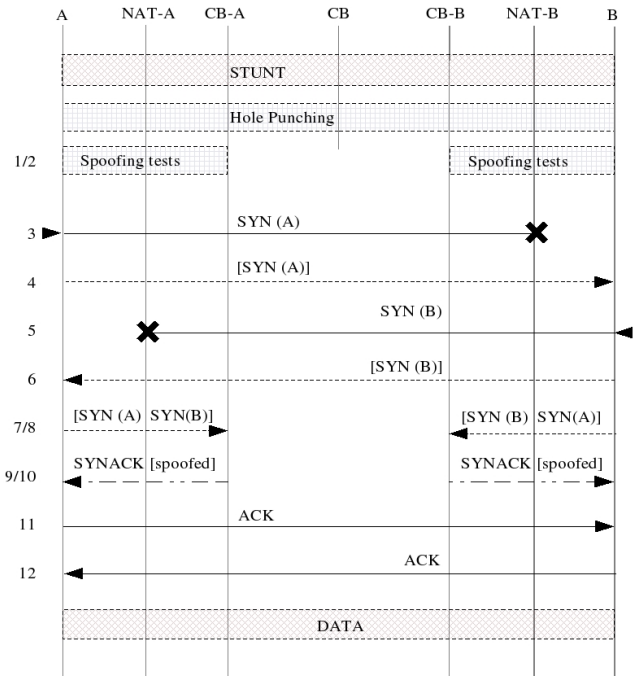


Figure 5.9: The STUNT-1 method with Spoofing from Well-Known Machine - message sequence

Before to start the connection attempt, each endpoint starts a *test spoofing procedure* to find a Super Node that can spoof packets to the sender of the request forcing the packet's source address to the remote endpoint (messages 1 and 2). Both

[2] The same result can be obtained using the Connection Broker to exchange the UDP messages between endpoints. For more clarity and simplicity we have preferred a directly UDP session.

endpoints send the initial SYN packet that is blocked on the destination's NAT (messages 3 and 5), learn the initial sequence number through a RAW socket and communicate these numbers to the other endpoint using the directly UDP session (messages 4 and 6). With messages 7/8 each endpoint communicates via UDP his initial sequence number, the remote sequence number and address to the Connection Broker choose that craft a SYNACK packet and spoofs it to the endpoint (messages 9/10). The ACK messages complete the TCP three way handshake (messages 11 and 12).

Method's issues. Like in the previous approach, the NAT may change the TCP sequence number of the initial SYN such that the spoofed SYNACK appears as an out-of-window packet. It requires a third-party to spoof a packet and, even if we have solved the network detection problem, is not certain that each endpoint can find a Super Node allowed to spoof a packet from the address requested. NAT's reply to an inbound SYN with an ICMP error or a RSTACK packet may compromise the connection establishment. A RAW socket must be opened on endpoints' machine, which requires superuser privileges.

5.2.3.2 Some considerations on spoofing approach

Spoofing has been used as the basis for a whole set of recent attacks. Additionally to the high detection of the spoofed traffic, that reduce the effectiveness of these methods, implement an application that make spoofing can be dangerous because it is very hard to test the security level, and thus, it could become a good target for attacks. Anyway, we have implemented and tested this solution.

5.2.4 NATBlaster Method

To solve the spoofing problem, here is exposed a possible solution, presented in [9] and called NATBlaster. We suppose that both the NATs involve in the communication don't accept incoming SYN packet, after an outcoming SYN packet, thus the method P2PNAT, presented in paragraph 5.2.2, cannot be applied. Each endpoint sends out the initial SYN packet listening the sequence number through a RAW socket. Both the SYN packet are dropped by the NATs. The two endpoints exchange the sequence number (messages 2 and 4) through an UDP connection and each sends a manipulated SYNACK packet the other expects to receive through a RAW socket. Each endpoint manipulates the SYNACK packet just setting the ACK sequence without modify the source address, and thus, without spoofing, and sends it to the other endpoint (messages 5 and 6). Once the SYNACK packets are received, the connection setup is completed with the ACKs exchange.

Method's issues. As seen in the STUNT-1 approach, this method fails if the NAT changes the sequence number of the initial SYN packet and if the NAT doesn't allow

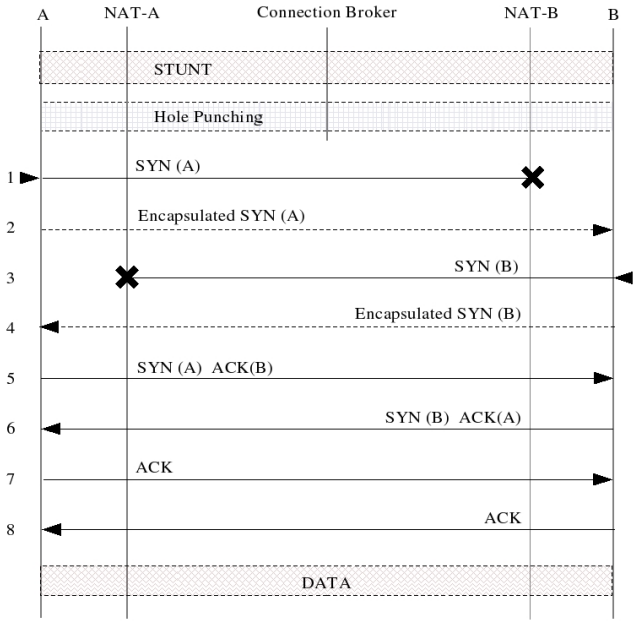


Figure 5.10: The NATBlaster method - message sequence

an outbound SYNACK packet immediately after an outbound SYN, a not normally sequence of packets. A RAW socket must be opened on endpoints' machine to inject the crafted SYNACK, which requires superuser privileges. Additionally, some restrictions on modifying IP header are introduced in Windows XP SP2.

Implementation's issues. Like in STUNT-1 methods, this approach requires the *pcap* library for outbound packet capture.

5.2.5 Low Time-To-Live Approaches

All the approaches previous presented can be slightly modified setting a low value of the IP time-to-live field, to avoid possible negative effects for incoming SYN packet on public NAT side. The SYN packet sent will be dropped in the middle of the network and an ICMP unreachable error will be returned.

As explained in paragraph 5.2.1, an ICMP packet may cause problem on NAT, which may releases the mapping for the session, or not accept following packets. The figure 5.2, for each methods previous exposed, shows the version with a low time-to-live value. These versions present the same problems seen before, except that the response to the first outbound SYN is always an inbound ICMP message. Normally, the solution with no low time-to-live value is more effective because the majority of the NATs drop the incoming SYN silently. We can thus take advantage

from this approach when we want to traverse the addresse's NAT that reply with a RSTACK packet not accepted by sender's NAT.

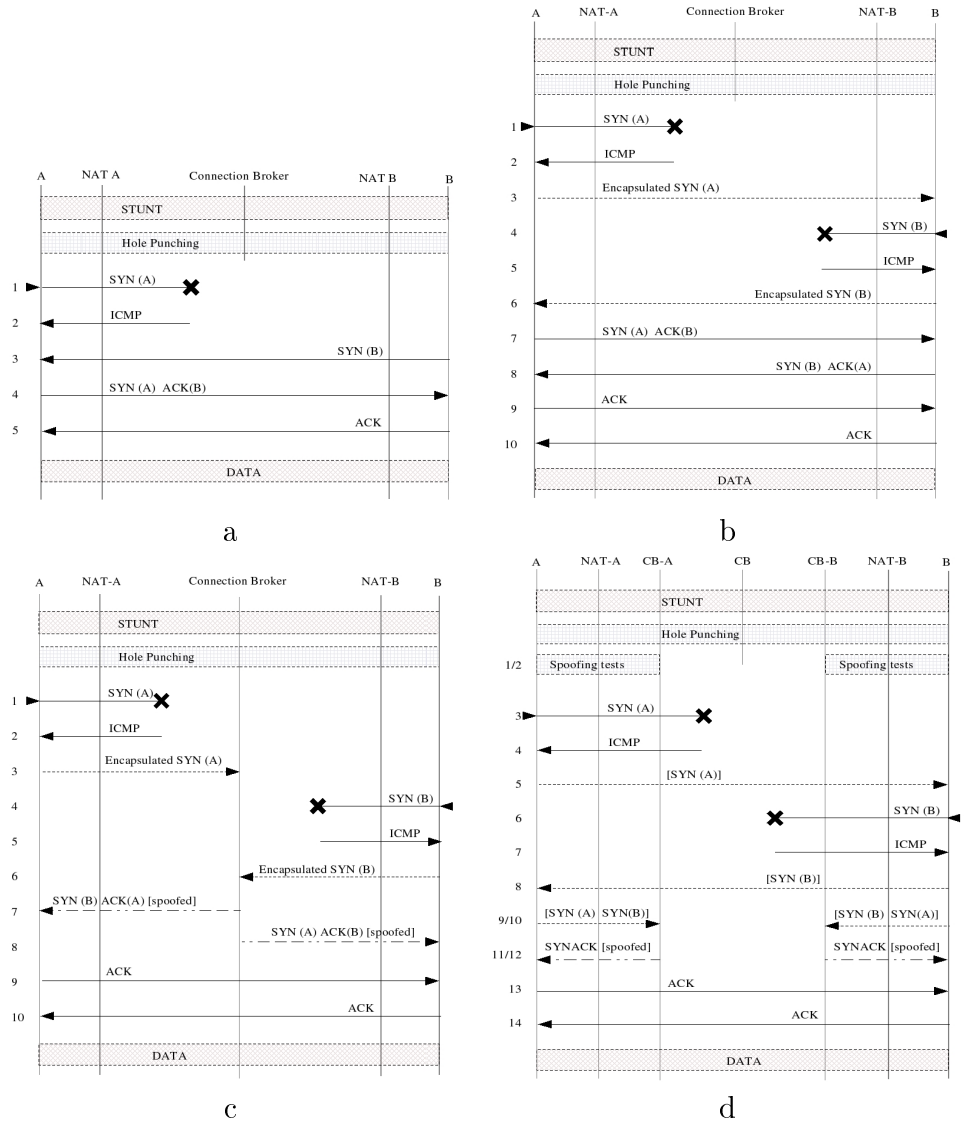


Table 5.2: Low Time-To-Live Approaches: (a) STUNT-2, (b) NatTrav, (c) STUNT-1, (d) STUNT-1 spoofing from well-known machine

Method's issues. Set the time-to-live is not well supported by all operative system. Windows XP, for example, does not allow this operation. Special drivers are thus required to set TTL under some systems.

Implementation's issues. To set the time-to-live value the IP-TTL socket option can be used. This require to determine a TTL large enough to cross its own NAT and low enough to not reach the other end's NAT. Such a TTL does not exist when the two outermost NAT share a common interface.

5.2.6 How to Choose the Right Method

The troubles encountered in the approaches' implementation and tests are resumed in Table 5.3, presented in the Guha and Francis work at Cornell University [6]. This table, combined with the STUNT discovery procedure, exposed in section 4.5, can help us to choose the right method to traverse the NATs and establish the TCP connection between the two NATed hosts.

Approach	NAT/Network Issues	UNIX Issues	Windows Issues
STUNT-2	- RST error - SYN-out SYN-in		
P2PNAT	- TCP simult. open - Packet flood		- TCP simult. open[2]
STUNT-1	- RST error - TCP Seq# changes - Spoofing	- Admin priv.	- Admin priv. - TCP simult. open[2]
NATBlaster	- RST error - TCP Seq# changes - SYN-out SYNACK-out	- Admin priv.	- Admin priv. - RAW sockets[1] - TCP simult. open[2]
STUNT-2 with low TTL	- Determining TTL - ICMP error - SYN-out SYN-in		- Setting TTL
STUNT-1 with low TTL	- Determining TTL - ICMP error - TCP Seq# changes - spoofing	- Admin priv.	- Admin priv. - Setting TTL
NATBlaster with low TTL	- Determining TTL - ICMP error - TCP Seq# changes - SYN-out SYNACK-out	- Admin priv.	- Admin priv. - Setting TTL - RAW sockets[1]

Table 5.3: Implementation, NAT and Network issues with the various TCP NAT traversal apporaches

The criteria that we have followed to choose the right approach are:

[1] Post WinXP SP2.
[2] Pre WinXP SP2.

- NAT binding behaviour
- NAT filtering behaviour
- NAT response to sequence outbound-SYN inbound-SYN
- NAT response to sequence outbound-SYN outbound-SYNACK
- NAT response to inbound ICMP or RST error message
- Operative System and access rights on machine

The library deployed and exposed in the next chapter, implements the methods discussed here. The implemented order for the various method is:

1. STUNT-2: it is the simplest with minimun requirements by NAT, network and endhost. If at least one NAT accept inbound SYN this is the right method.
2. P2PNAT: it has the same characteristics than STUNT-2, except for TCP simultaneous open support. It is used if no information are provided about NAT behaviour for the inbound-SYN outbound-SYN sequence.
3. STUNT-1: this method is very effective for NAT's point of view. It is, in fact, transparent for the NAT, but requires spoofing that has a low possibilities to traverse the network. STUNT-1's strong restriction is the strictly dependence from the operative system and user privileges.
4. NATBlaster: It reduces the STUNT-1's spoofing problems, but requires that the NAT accepts the sequence outbound-SYN outbound-SYNACK. Like STUNT-1, it is strictly dependent from operative system and user privileges.

5.3 Results

Using data presented in [6], some of which have been shown previously, we can calculate the percentage of NAT that we can traverse with the methods implemented. The Figure 5.11 shows that, without using the methods that require spoofing, in particular STUNT-1, we are capable to traverse the 94% of the NAT on the world-wide market. If we apply the limitated effectiveness of STUNT-1 method to the remaining 6% we can reach the results of 96% of the existing NATs. If we set a low TTL value we obtain worse performances.

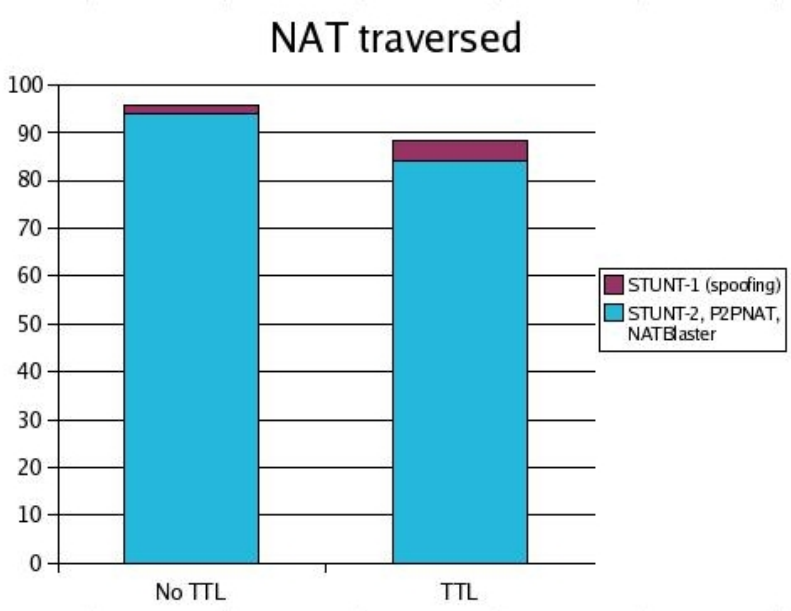


Figure 5.11: Prevision of the NATs traversed percentage with the modules implemented

Chapter 6

Implementation of NTCP library

After several tests, that have revealed either the various NAT behaviours either the advantages and disadvantages for every approach, we have implemented a library to allow the NAT traversing: the NAT Traversal TCP (NTCP) library. This chapter shows the library structure and the functionalities supplied.

6.1 The Python Language and the Twisted Library

The chosen language to implement the library is *Python*. It is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries. A Python implementation is portable: it runs on many brands of UNIX, on Windows, OS/2, Mac, Amiga, and many other platforms. Using python we have been able to quickly implement the code in a simple way. Moreover, the project SOLIPSIS, the final adressee of the *ntcp* module, is written in python.

With Python we have been able to have use of the Twisted module. Twisted is an asynchronous networking framework written in Python, supporting TCP, UDP, multicast, SSL/TLS, serial communication and more. It provides the tools to write client and server networking applications.

Although his advantages, Twisted has the disadvantage to be an extremely high level module. It doesn't allow to approach some important details for network programming, in particular some socket options.

Since SOLIPSIS uses the Twisted module, the *ntcp* package developed contains a wrap for the standard Twisted functionalities to set up TCP connection between endpoints.

6.2 NTCP Implementation

The main *ntcp*'s classes are shown in the Figure 6.1. There are four principal sections: the NAT Connectivity classes, the STUNT protocol, the Hole Punching protocol and the NAT traversal approaches implementation. Here a description of every section. For the complete API documentation and a report on NAT traversal see the Appendix C

- *NAT Connectivity*: This section is the interface with the application, it's, in fact, the highest level of the module. It contains all the functions to discover NAT informations (type and mapping) and force a connection through NATs with the Super Node Connection Broker's help or by a directly UDP connection with the remote endpoint. Specifically, the function *natDiscovery()* discovers NAT presence and NAT behaviours. The functions *listenTCP()* and *connectTCP()* wrap the Twisted functions for the classic client/server instances. The developer that intends to use the *ntcp* module have to call just these three function for a basic service. The *holePunching()* function allows to establish a direct UDP communication between two endpoints.
- *STUNT implementation*: This is the STUNT protocol implementation. It provides a client and a server implementation. The client will be integrated and used by the *NAT Connectivity* section.
- *Hole Punching Protocol implementation*: The hole punching protocol, whose operations have been exposed in the section 5.1.2, is implemented in the Puncher-Protocol classes. The *Puncher* class is the endpoint side of the protocol, while the *SNConnectionBroker* is the implementation for the Connection Broker that is completely independent, and it could be integrated in the endpoint side for a full peer-to-peer paradigm.
- *NAT Traversal Approaches*: the *ConnectionPunching* class contains all the methods' implementation. Specifically, this class, from the informations supplied by *NATConnectivity* module, can make a TCP connection between a NATed endpoint and a not NATed endpoint, between endpoint in the same local area network, or between two NATed hosts, using the methods STUNT-1, STUNT-2, NATBlaster and P2PNAT.

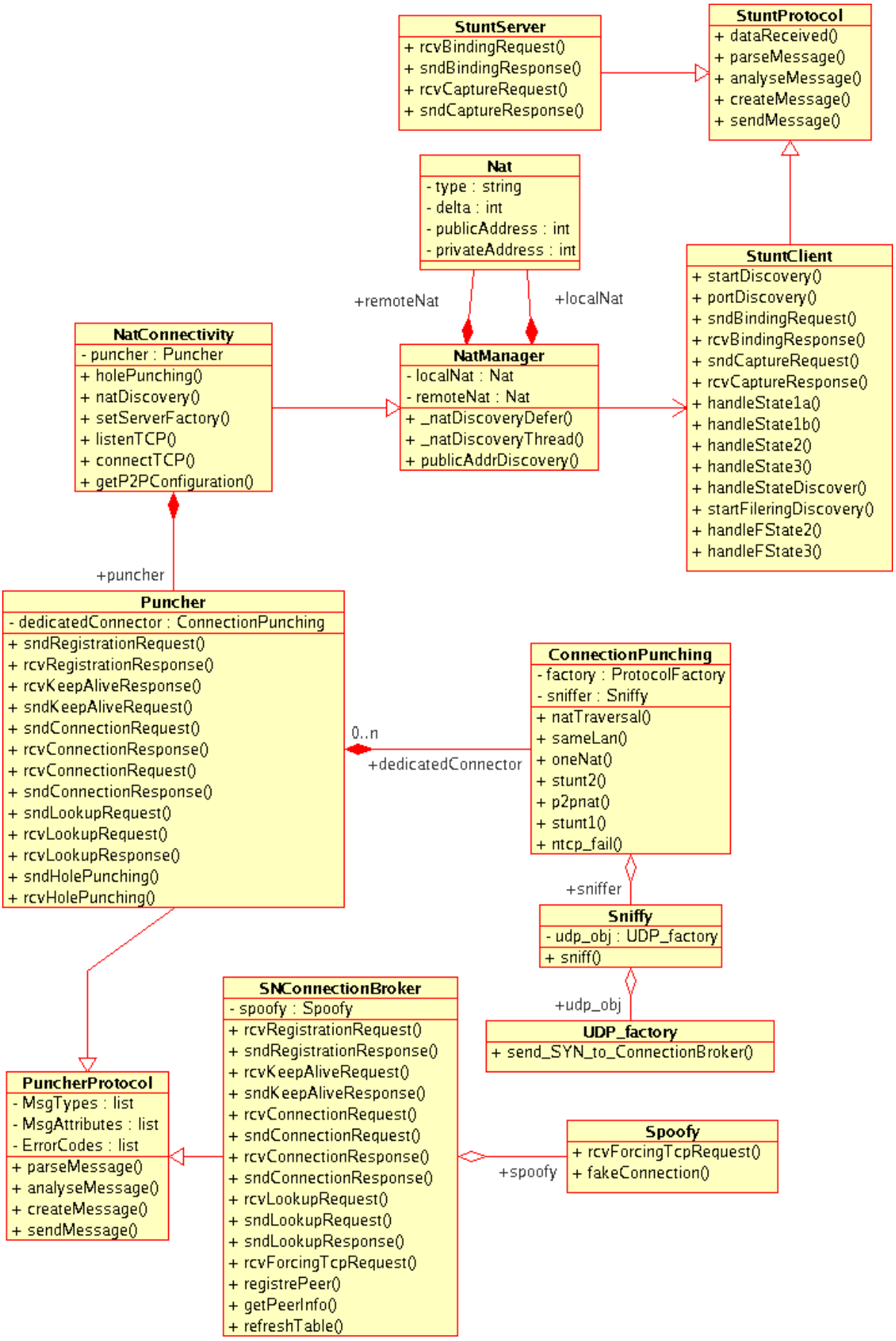


Figure 6.1: UML diagram of the NTCP module

Appendix A

Download Source Code and Documentation

Repository. The project is hosted in the BerliOS repository:

```
http://developer.berlios.de/projects/ntcp/
```

Anonymous SVN Access. The project's BerliOS Developer SVN repository can be checked out through anonymous (svnserver) SVN with the following instruction set. To see the list of the subdirectories available in the repository use the web-based SVN repository access with ViewCVS or WebSVN.

```
svn checkout svn://svn.berlios.de/ntcp/trunk
```

Developer SVN Access via SSH. Only project developers can access the SVN tree via this method. SSH2 must be installed on your client machine. Substitute developername with the proper value. Enter your site password when prompted.

```
svn checkout \
svn+ssh://developername@svn.berlios.de/svnroot/repos/ntcp/trunk
```

The subdirectories for documentation or source code only are respectively:

- /ntcp/trunk/doc
- /ntcp/trunk/p2pNetwork

Appendix B

Hardware and software requirements

B.1 General Requirements

- Python (`http://www.python.org/`)
- Twisted (at least 1.3) (`http://twistedmatrix.com/projects/core/`)

B.2 Special Unix Requirements

Connection Broker side:

- UNIX: the Connection broker require UNIX system for spoofing section
- libpcap (at least 0.9.1)
- Pcapy: python library for spoofing (modification of address, SYN, ACK, TTL, ...)
  `http://oss.coresecurity.com/projects/pcapy.html`
- Impacket: python library to sniff and listen for outgoing packets
  `http://oss.coresecurity.com/projects`

Peer side: if you want to implement spoofing method you need this package. Otherwise the NTCP operations will be reduced to the methods without spoofing. (PS: Spoofing methods require administrator privilege on the machine too!)

- libpcap (at least 0.9.1)
- Pcapy: python library for spoofing (modification of address, SYN, ACK, TTL, ...)
  `http://oss.coresecurity.com/projects/pcapy.html`
- Impacket: python library to sniff and listen for outgoing packets
  `http://oss.coresecurity.com/projects`

B.3 Special Windows Requirements

- Python (at least 2.3): some socket options are not available in the previous version
- winpcap (at least 3.0): the packet capture library for Windows
  `http://www.winpcap.org`
- Impacket: python library to sniff and listen for outgoing packets
  `http://oss.coresecurity.com/projects`

Appendix C

Installation Instructions, Getting Started Tips, and Documentation

C.1 Installation

There is no installation procedure at now. So download the source code (see download section), put it to the desired folder, and make the PYTHONPATH environment variable to point it. Execute this command or put it in your *profile* file.

```
export PYTHONPATH=\$PYTHONPATH:/directory_to_ntcp/ntcp/
```

C.2 Modules' structure

The source code in the /ntcp/branches/beta/ntcp is organised like exposed:

```
/ntcp/branches/beta/ntcp
 |
 |__ connection: some usefull class to wrap twisted structure
 |
 |__ punch: Hole Punching protocol implementation
 |          (Connection Broker and Puncher)
 |
 |__ stunt: STUNT porotocol implementation (client and server)
 |
 |__ test: tests for NTCP connectivity.
 |         Look here if you are looking for some example
 |
 |__ NatConnectivity.py: the most important file.
                         It gives you the access to the NTCP functions
```

C.3 Getting started

To know how to use ntcp library in your code look at the examples in the ntcp/test/ directory; the file simulator can help you.

C.4 Start Server and Connection Broker

For particular settings see the Configuration files section.

- STUNT server: start the python file in your branch ntcp/stunt/StuntServer.py
- Super Node Connection Broker: start the python file in your branch ntcp/punch/SNConnectionBroker.py

C.5 Configuration Files

Before to start use the ntcp library, you have to configure the configuration file.

Endpoint side:

- *ntcp/stunt/StuntClient.py*

```
DefaultServers = [\\
    ('p-maul.rd.francetelecom.fr',3478),\\
    ('new-host-2.mmcbill2',3478),\\
]
```

- *ntcp/p2pNetwork.conf*

```
# ConnectionBroker for peer configuration

[holePunch]
ConnectionBroker: p-maul.rd.francetelecom.fr:6060
```

STUNT Server side: For STUNT server you need two different Ip addresses. These addresses can be on the same machine or on two different machine. On UNIX system, infact, you can configure two different Ip address on the same ethernet device.

- *ntcp/p2pNetwork.conf*

```
[ stunt ]
# STUNT Server configuration
stuntIp : 192.168.1.201
stuntPort : 3478
# If the two server are on the same machine
# it needs of the second interface's name
otherStuntIp: 192.168.1.203
otherStuntPort: 3479
```

C.6 API Documentation

For the complete API documentation and a report on NAT traversal see the documentation in /ntcp/branches/beta/doc/ directory

Bibliography

[1] Solipsis, France Telecom - R&D Division
http://solipsis.netofpeers.net/wiki2/index.php 7

[2] Almost TCP over UDP (atou), Tom Dunigan, 2004-01-12
http://www.csm.ornl.gov/~dunigan/net100/atou.html 22

[3] Peer-to-Peer Communication Across Network Address Translators, Bryan Ford, Pyda Srisuresh, Dan Kegel, 2005-02-17
http://www.brynosaurus.com/pub/net/p2pnat/ 29, 33

[4] Connection à travers des pare-feux, Laurent Viennot
http://gyroweb.inria.fr/~viennot/enseignement/dea/stages2004-2005/firewall.html

[5] NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity, Saikat Guha, Yutaka Takeda, Paul Francis, Cornell University and Panasonic Communication, USA
http://nutss.gforge.cis.cornell.edu/pub/fdna-nutss.pdf 33, 40

[6] Characterization and Mesurement of TCP Traversal Through NATs and Firewalls, Saikat Guha, Paul Francis, Department of Computer Science, Cornell University, USA
https://www.guha.cc/saikat/pub/draft-imc05-stunt.pdf 18, 21, 38, 45, 46

[7] Is the Internet Going NUTSS?, Paul Francis, Cornell University, November-December 2003 http://www.cs.cornell.edu/People/francis/ieee-nutss.pdf 19

[8] TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem, Jeffrey L. Eppinger, Institute for Software Research International School fo Computer Science, Cornegie Mellon University, January 2005, CMU-ISRI-05-104
http://reports-archive.adm.cs.cmu.edu/anon/isri2005/CMU-ISRI-05-104.pdf 33, 38

[9] NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs, Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, Adrian Perrig, Information Networking Institute, Carnegie Mellon University, Pittsburgh, USA
http://www.andrew.cmu.edu/user/ggw/natblaster.pdf 33, 42

[10] UPnP - Universal Plug and Play Internet Gateway Device, Microsoft Corporetion, November 2001
http://www.upnp.org 10, 22

[11] MIDCOm Protocol Semantic, M. Stiemerling, J. Quittek and T. Taylor, June 2004
http://www.ietf.org/rfc/rfc3989.txt

[12] Transfert de fichier, TFTP, Laurent Viennot
http://www.enseignement.polytechnique.fr/profs/informatique/Laurent.Viennot/majReseau/2004-2005/td4/index.html

[13] Architecture de gestion de traversé de protocoles au travers des pare-feu et des routeurs NAT, JOEL BQO-Lqn TRAN, Université de Sherbrooke, September 2003

[14] Dive Into Python
http://diveintopython.org

[15] Twisted Documentation
http://twistedmatrix.com/projects/core/documentation/howto/index.html

IETF - Request for Comments

[16] rfc 3489: STUN - Simple Traversal of UDP Through NATs

[17] rfc 793: TCP - Transmission Control Protocol

[18] rfc 768: UDP - User Datagram Protocol

[19] rfc 1928: SOCKS Protocol Version 5

[20] rfc 2663: IP Network Address Translator (NAT) Terminology and Considerations 16, 23

IETF - Internet Draft

[21] STUNT: Simple Traversal of UDP Through NATs and TCP too, Saikat Guha, Cornell University, 11 December 2004
https://www.guha.cc/saikat/pub/draft-imc05-stunt.pdf 33, 39

[22] NAT Behavioral Requirements for Unicast UDP, C. Jennings, Cisco System, 11 April 2005

[23] Application Design Guidelines for Traversal of Network Address Trankators, B. Ford, P. Srisuresh, D. Kegel, February 2005

[24] NAT Classification Results using STUN, C. Jennings, Cisco Systems, October 24, 200411

[25] Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, J. Rosenberg, Cisco Systems, 17 July 2005. 12, 15, 22, 38, 40

[26] SOLIPSIS Node Protocol 12, 15, 19

Tool

[27] Python
`http://www.python.org/` 12

[28] Twisted
`http://twistedmatrix.com/products/twisted`

[29] LIBPCAP `http://www.tcpdump.org/`

[30] WINPCAP `http://www.winpcap.org/` 8

[31] Emacs
`http://www.gnu.org/software/emacs/`

[32] Subversion
`http://subversion.tigris.org/` 41

[33] RapidSVN
`http://rapidsvn.tigris.org/` 41

Mailing list

[34] Twisted-Python mailing list
`http://twistedmatrix.com/cgi-bin/mailman/listinfo/twisted-python`

[35] p2p-hackers mailing list
`http://zgp.org/mailman/listinfo/p2p-hackers`