NAT TRAVERSAL

MIDHUL VARMA

Abstract

With the increasing usage of mobile devices and the rise of the Internet of Things, the number of devices connected to the internet has skyrocketed. This has lead to a spike in the rate of depletion of Internet Protocol v4 (IPv4)'s limited pool of 4,294,967,296 addresses. In response to this scarcity, local networks especially wireless ones are being increasingly privatized using middleboxes with special functionality called Network Address Translation (NAT), which allows multiple hosts to share the same public IP address. NAT temporarily solves the address space exhaustion problem, but is not a fully transparent solution as it breaks the functionality of certain applications, especially those which require peer to peer (P2P) communication. The primary goal of this article is to examine these problems and study techniques which can be used to overcome them. It starts off by describing the basic working mechanism of NAT, different possible flavours of NAT and the problems that they create for certain types of applications. Following this different kinds of NAT traversal techniques are discussed at the conceptual level. After that some standardized protocols used for NAT traversal are overviewed. Finally the NAT traversal functionality of some real life applications is briefly described.

CONTENTS

1 WHY, WHAT, WHERE NAT?

1.1 *Working Mechanism*

Network Address Translation is a broad term and can be defined as a method by which IP addresses are mapped from one address realm to another, providing transparent routing to end hosts. Strictly speaking there are several classes of NAT: Traditional NAT, Bidirectional (or) two-way NAT, Twice NAT & Multihomed NAT. The class of NAT which is of interest in this report is a subclass of Traditional NAT called Network Address Port Translation (NAPT). This is by far the most widely deployed class of NAT and is used by middleboxes such as WiFi routers in home or public areas to privatize local networks. Hence in the remainder of this report the term NAT is used to refer to NAPT only and no other class of NAT.

In a typical NAT configuration, a local network uses one of the designated ″private″ IP address subnets. A router on that network has a private address in that address space. The router is also connected to the Internet with a ″public″ address assigned by an Internet service provider. As traffic passes from the local network to the Internet, the source IP address in each packet is translated on the fly from a private IP address to the public IP address by the router. Doing just this is not going to be sufficient, because when a destination sends packets back to the router, it will have no way of determining which host of the private network to forward the packet to. (as any of them could have contacted the destination). Hence, the router needs to maintain some state for every outgoing packet it translates. Using only information from the packets' network layer headers (source and destination IP address) to maintain this state is not efficient, as multiple hosts could be communicating with the same destination (a popular website perhaps). Routers equipped with NAT try to use information from transport layer headers to maintain more specific state information. Traditional simple routers are purely layer 3 devices, while NAT routers additionally need the ability to understand layer 4 headers. State information is stored in the router by creating ″sessions″ for packet flows which pass through it. The mechanism for creating, maintaining and destroying these sessions is protocol dependent:

- UDP packets (User Datagram Protocol): In addition to source and destination IP addresses, UDP packets also contain source and destination port numbers. The router extracts the four tuple (source IP, source port, destination IP, destination port) from every packet it receives from the private network. It then creates a session based on this 4 tuple (if one doesn't already exist) and then assigns a public port for this session. For every outgoing packet whose 4 tuple matches that of the created session, the router modifies the packet's source IP address and port to the public IP address of the router and the public port which was mapped to the corresponding session, before forwarding it to the destination. The destination now thinks that the packet actually came from the (public IP address, port) pair and hence sends reply packets with destination set to the (public IP addres, port) pair. On receiving these reply packets the router can decide which session they belong to based on the port on which they are received, and then forward them to the right host in the private network. Essentially, a session represents a flow of packets and the router is multiplexing packet flows originating from different source hosts in the private network, by assigning a port number to each flow. Once a session is created it cannot be stored forever, as the router has a limited pool of port numbers (only 65536). Since UDP is a connectionless protocol, it is not possible for the router to determine when a flow of packets between a source and destination will terminate. Hence, routers usually implement timers, and destroy sessions after no packets corresponding to them are observed for a fixed timeout period.

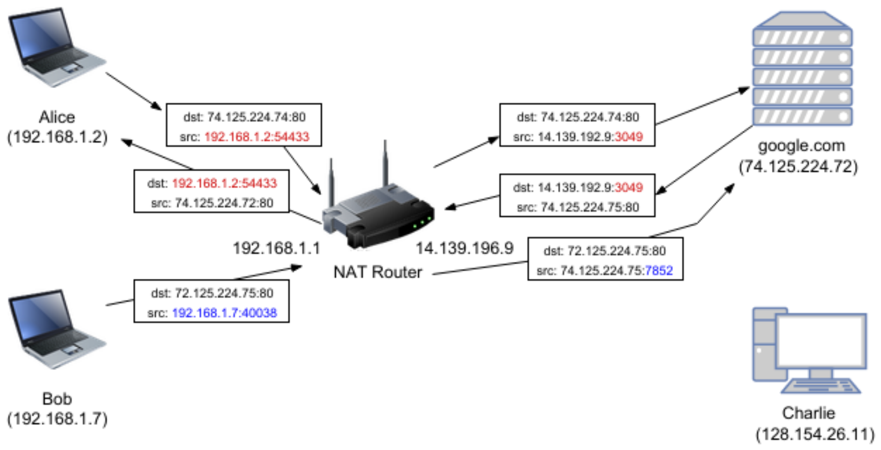
- TCP packets (Transmission Control Protocol): TCP packets also contain source and destination port information just like UDP, hence the same kind of session mechanism as described above is used. Unlike UDP, TCP is connection oriented, and the router can exploit this to determine when to create and destroy sessions. For example, a TCP

SYN packet would mean a new TCP connection is being created, so the router creates a session based on the packet's 4 tuple. When TCP FIN packets are observed, the router knows the present connection has been ended and can choose to destory the corresponding session. In this way, sessions of TCP flows are handled more effectively, than UDP ones (which are maintiained using timers).

- ICMP (Internet Control Message Protocol): Hosts inside the private network may try to PING servers in the public internet. To support this the NAT router has to also handle ICMP packets. Unlike TCP/UDP, ICMP packets do not contain any port information. There is an identifier field in ICMP query messages which is set by the Query sender and returned unchanged in response message from the Query responder. This can be used by the router to maintain sessions just like port numbers are used for TCP/UDP packets.

Note: The "session"s which are being refered to here are internal to the router and have nothing to do with the applications running on the end hosts.

Consider the following example scenario as shown in:



Alice and Bob are in a private network (192.168.1.0/24) and they connect to the internet through a WiFi router who's private IP is 192.168.1.1 and public IP is 14.139.192.9. Now Alice sends a packet from port 54433 to one of google's servers (74.125.224.72). The router receives this packet, and creates a session corresponding to the 4 tuple (192.168.1.2, 54433, 74.125.224.72, 80) and allocates the port 3049 to it. Now it modifies the IP address of the packet to it's public IP, and sets the port as 3049 and send it to the server. The server receives this modified packet, and responds to it. The response packet reaches the router, and looking at the port number (3049) the router associates this packet to the previously created session, from which it determines the right destination is Alice (192.168.1.2, 54433), to which it finally forwards the packet. Packets can flow in this fashion from Alice to the server as long as the session is alive in the router. Now while Alice's session is alive, if Bob tries to communicate with the same server on port 40038, the router will create another session and give it a different port (7852). All packets coming from the server to the router on port 3049 are sent to Alice, and all those which come on port 7852 are sent to Bob.

For daily life applications such as web browsing, email, etc. where the initial attempt for communication is made by hosts in the private network, NAT is transparent. In the above discussed example Alice feels that she is sending/receiving packets to/from the server directly and the server feels that it is sending/receiving packets to/from the router. Neither of them are aware of the NAT that is going on in the middle. As a consequence they do not require any special configuration/settings to work with NAT, unlike in the case of HTTP proxy servers, which require clients to configure proxy settings in their web browsers. This transparency is a major reason for the large scale deployment of NAT middleboxes.

1.2 *NAT flavours*

Unlike most network technologies used in the Internet, Network Address Translation (NAT) unfortunately has no well defined and/or globally accepted standard. Device manufacturers tend to implement custom mechanisms which can vary form one another in several aspects. This makes it difficult to categorize and discretely label NAT mechanisms. For the purposes of NAT traversal it is useful to categorize NAT mechanisms based on two important characteristics: (Endpoint here means IP address, port pair. Private endpoint refers to the initial source address, port of an outgoing packet before translation. After translation, the new source address, port is referred to by the term Public endpoint).

- **Mapping** of private endpoints to public endpoints
  - Endpoint independent (Cone NAT)
  - Endpoint dependent (Symmetric NAT)
- **Filtering** of incoming packets at a given public endpoint
  - Open (No filtering)
  - Address restricted
  - Port Restricted

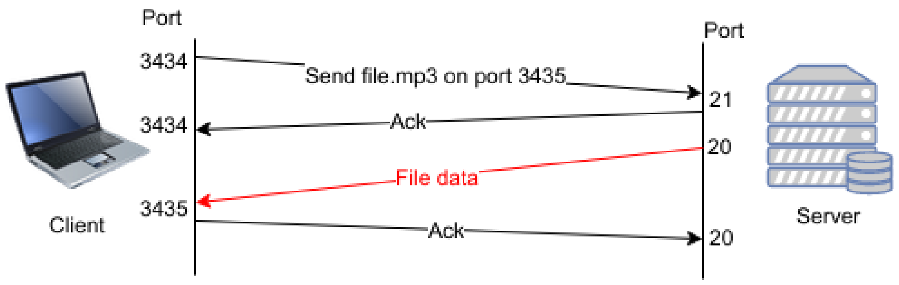
Mapping is essentially the process of allocating a port number to a given session. While doing this allocation NAT middleboxes use a mapping scheme which can in general depend on the 4-tuple (source address, source port, destination address, destination port). If the mapping is independent of destination address and port and depends only on the source's address and port, it is called an **endpoint independent mapping** scheme i.e all the presently active sessions corresponding to a given private IP address, port pair are mapped to the same port number. In essence it is as if the NAT device assigns a public port to every active private endpoint. In the previous example scenario, let's say Alice while communicating with the server also starts sending packets to another host on the Internet (128.154.26.11:21). The router will create a new session for this flow, but if it is using an endpoint independent mapping scheme, the port allocated to this new session will also be 3049 (same as that of the previous session). This particular characteristic is a minimum requirement for the success of any reasonable NAT traversal technique as will be described further in this article. Fortunately a majority of the NAT devices found in the wild are of this type, particularly those which are found in home/personal Wifi routers. In Endpoint Dependent mapping schemes, NAT devices may map different sessions from the same private address port pair to different port numbers. Such devices popularly known as Symmetric NATs are not very common and are sometimes found in NATs used by large organizations for their internal networks.

Note: In endpoint independent mapping, it is not necessary that every private address, port pair is given a permanently fixed public port. It is sufficient if at any given point in time, all active/alive sessions corresponding to a private endpoint are mapped to the same port number. Let's say Alice in the previous example, after finishing all sessions on port 3049 shuts down her computer and restarts it the next day. Now if she tries to send packets to a server, the new session which is created could very well be mapped to a different port say 5513.

The other important characteristic for the classification of the behaviour of NATs is the set of rules which they use to filter packets which are received from the outside network. Again considering the example in the above figure, when Bob sends packets to the Google server (74.125.224.72:80), a session is created with port number 7852. All packets sent from the server to the router with destination port as 7852 will be forwarded to Bob. Now what if Charlie (128.154.26.11) another host on the Internet tries to send a packet to the router with destination port 7852. If the router allows this packet and forwards it to Bob, it is not doing any filtering (Open). If the packet is dropped, since the session is meant for Bob and the server and not Charlie, then the NAT is said to be using Address Restricted Filtering. Now let's say the NAT is using Address Restricted filtering. This time the packet comes from the server itself (74.125.224.72) but from a different source port (32), which is different from the source port corresponding to the session (port 80). If the router drops even this packet, it is said to be using an even stricter filtering criteria called Port Restricted filtering.

1.3 *Evils of NAT*

NAT does offer transparency for most client server style applications (client inside the private network and server out in the public Internet) but is definitely not completely transparent. It breaks the Internets original uniform address architecture, in which every node has a globally unique IP address and can communicate directly with every other node. In fact it doesn't even offer transparency for all kinds of client server applications. To see why consider as an example the File Transfer Protocol (FTP) used in active mode.



As shown in the figure, the client which needs a particular file, requests for it by making a TCP connection to the server on standard port 21 asking it to send the desired file on port 3435. The server acknowledges this, and then makes another TCP connection to port 3435 of the client, and sends the file data over this connection. After the data is received the client acknowledges. Now consider what would happen if there was a NAT middlebox in between the client and the server. The connection highlighted in red in the figure, would not be possible since it is initiated by the server, and there is no session in the middlebox associated with it. This illustrates how NATs break the functionality of applications which explicitly make use of address or port numbers in their payloads and require connections initiated from the server's end. Well at least in the case of FTP users have the backup option of using the passive mode (where instead of the server the client initiates the connection for data transfer) which is NAT friendly. In general it is possible to make any client server application work over NAT by ensuring that all connections are initiated by the client. The situation is worse for peer to peer applications such as VoIP, video conferencing and multiplayer games. This is because if 2 peers are both behind NAT middleboxes, neither of them can initiate direct communication with the other, and without that there cannot be any flow of packets between the two peers. The remainder of this report is focused on describing solutions to this problem.

2 SOLUTIONS

The problems which NATs create for applications can be solved by either modifying the NAT mechanism at routers themselves or by implementing additional NAT traversal functionality in the application's code.

2.1 *Application Level Gateways (ALG)*

When NAT device manufacturers realized these problems and their implications, they started trying to "augment" the mechanisms that their devices were using to make certain commonly used applications like FTP, BitTorrent etc. work. Deep packet inspection of all the packets handled by the NAT device over a given network makes this functionality possible. An ALG understands the protocol used by the specific applications that it supports i.e each application requires its own custom ALG. For example an FTP ALG would update NAT with appropriate data session tuples and session orientation so that NAT could set up state information to allow incoming connections from FTP servers on the client specified ports. ALGs are not a suitable solution for applications whose functionality can be updated frequently (for example a VoIP service provided by a company, the company could make regular changes/improvements to the protocols their application uses). Such out of date ALGs not only break the application but also make it very difficult for other application level NAT traversal techniques to work. Moreover, it is usually not easy to update/modify software built into routers and hence using ALGs could be a bad idea. VoIP/peer to peer service providers generally recommend their customers to disable any ALGs in their routers.

2.2 *Port forwarding*

It is possible to configure some NAT devices to forward all packets that they receive from external hosts on a specific port to a specific private address, port pair. Most personal/home WiFi routers support such static port forwarding i.e users can login to the router's admin panel and add fixed port mappings. This makes it possible for hosts in the internal network to receive inbound connections from the external world on ports which have been forwarded. However such static forwarding is not very helpful for general purpose NAT traversal, as it does not offer much flexibility and requires user intervention. Some NAT routers have special functionality which enables hosts in the private network to communicate with the router and dynamically allocate ports for forwarding. The Internet Gateway Device (IGD) Standardized Device Control Protocol which is part of the Universal Plug and Play (UPnP) family of protocols makes it easy for hosts in the private network to do the following:

- Learn the public (external) IP address
- Enumerate existing port mappings
- Add and remove port mappings
- Assign lease times to mappings

If a client is behind a NAT which supports this protocol, it can send IGD requests to the NAT router to obtain public port mappings for its private ports making it possible to receive incoming connections. Port Control Protocol (PCP) and its predecessor NAT-Port Mapping Protocol (NAT-PMP) are standardized protocols which offer similar functionality. Unfortunately by the time these protocols were standardized a huge number of NAT devices were already deployed throughout the Internet, and only a tiny fraction of NAT devices today support protocols like these.

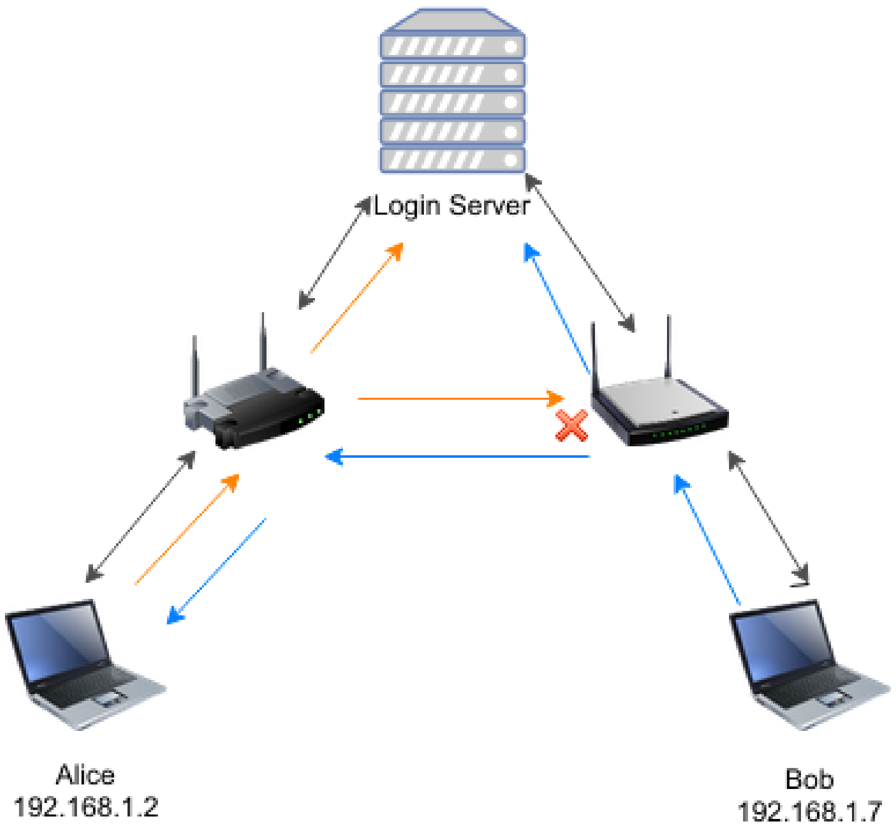
Depending on routers/manufacturers for providing NAT traversal functionality is clearly not a viable solution for applications that need to be deployed to users on a large scale. This is what motivates the development of techniques which applications can use to traverse NATs WITHOUT any help from the device/router.

2.3 *Hole punching*

Hole punching is an important general technique used by several applications and protocols for reliable NAT traversal. It can be used to establish flows of UDP packets or TCP connections between two peers who are both located behind NATs. To understand hole punching we shall first see how it can be used for UDP packets

2.3.1 *UDP hole punching*

Let us consider the example of a simple VoIP application to understand how UDP hole punching works. The situation is illustrated in the following figure:



There is a server on the public Internet which maintains a list of peers who are online along with their details. When a user wants to sign in, he/she opens a connection with the login server. We shall assume that this connection is kept open as long as the peer is online, so that there is always an open bidirectional channel of communication between the server and the peers. Since these connections are initiated by the peers the NAT middleboxes cause no trouble. We shall also assume that the NAT middleboxes in the figure translate private addresses using an endpoint independent mapping scheme. Later in the discussion we will return to this point. We now consider two peers Alice and Bob who are online. Lets say Alice wants to make a call to Bob. Both of them can communicate with the server but cannot send packets to each other directly. The hole punching process makes this possible and consists of the following steps:

1. Alice first creates a UDP socket and binds it to a particular port on her computer. Using this socket she sends UDP packets to the server which contain a call request to Bob. This creates a session in the router to which Alice is connected, which is mapped to a particular port.

2. Upon receiving these packets the server first records their source IP address and source port number. These are nothing but the public IP and port corresponding to Alice's private IP and port pair (for that particular session in the NAT middlebox). The server then checks if

Bob is online. If he is, it uses the open connection which it has with him to forward Alice's call request to him, along with Alice's public IP and port as recorded by the server.

3. Bob now knows Alice's public IP and port mapping. If he wants to answer the call he too creates a UDP socket and binds it to some port on his computer and uses it to send a positive acknowledgement to the server, which leads to the creation of a session in Bob's router.

4. The server again records his public IP and port and sends them to Alice along with a positive acknowledgement.

5. Now both the peers have each others public address, port pairs. They try to send UDP packets to each other using these public addresses,ports i.e Alice sends a packet to Bobs public address,port and Bob sends a packet to Alice's public address,port. Depending on the filtering rules of the NAT middleboxes the following cases could occur: - If the NAT middleboxes have no filtering rules (Open), both the packets will be allowed through and reach the opposite peer - If the filtering is either Address restricted or Port restricted (which is more strict than the former), the exact timing of events determines what happens:

    - Alice's packet crosses her own NAT and reaches Bob's NAT before his packet crosses it. In this case Alice's packet is dropped by Bob's NAT as it has not yet seen any recent packets sent from Bob to the source address of Alice's packet. Later, Bob's packet reaches Alice's NAT and is allowed as to pass through as it has already seen a packet sent from Alice targeted to the source address of Bob's packet. (This is the case depicted in the figure)
    - Same as the case above except Bob's packet reaches Alice's NAT first
    - Both the packets cross their respective NATs before the other's packet reaches them. In this case both the packets are allowed to pass through

In any case at least one of the packets reaches it's destinations. Also both NATs have created mappings to allow packets from the opposite ends to pass through. It's as if the peers have "punched holes" through their respective NATs by sending outgoing packets. The peers can now freely stream voice to each other over UDP.

It is important to note that peers must use the same port with which they initially sent packets to the server for all following peer to peer communication, as using a different source port might lead to a different port mapping on the router which the other peer does not have knowledge about. At the beginning of this example we had assumed that the NAT middleboxes use an endpoint independent mapping scheme. If either one of them was a symmetric NAT (endpoint dependent) it could create different public port mappings for outbound packets from the same private address,port targeted to the server and the opposite peer, making the hole punching process fail. Hence, endpoint independent mapping is a necessary condition for the success of Hole punching. Luckily as mentioned before, very few NAT devices use endpoint-dependent mapping schemes and this gives UDP hole punching a reasonable success rate.

2.3.2 *TCP Hole punching*

The basic concept behind TCP hole punching is essentially the same as that of UDP hole punching, except the implementation details make it more complicated. Lets again use the previous simple VoIP example application and try to implement TCP NAT traversal. Our goal in the example scenario would then be to establish a valid TCP connection between Alice and Bob.

As in the UDP case Alice and Bob will have to create TCP sockets and use them to connect to the server. The server will then send them each others public address, port pairs. They must then attempt to make TCP connections to these public address, port pairs from the same private address, port pairs they had used to make the initial TCP connection to the server. Everything might seem to be fine at first sight, but a closer look will reveal some problems:

1. Once a UDP socket is created, it can be used to send UDP packets to different destination endpoints (address, port pairs). Additionally it can also be used to receive packets which have their destination as the port which the socket is bound to. These operations can happen simultaneously in the sense that for example first one packet can be sent to a particular destination, then one can be packet from some sender can be received following which another packet can be sent to a different destination. TCP sockets do not offer this flexibility, as they were designed with a client server model in mind and are bound to the specifications of the Berkeley Sockets API. The sockets API specifies that sockets are of 2 kinds:

   - Active sockets for initiating connections to given destinations using the *connect()* system call
   - Passive (Listening) sockets which wait for connections from active sockets using the *accept()* system call. On receiving a connection request the *accept()* system call creates a new socket bound to the same port as the listening socket and uses it for the new connection. Since there are multiple sockets bound to the same port, incoming packets are multiplexed among the sockets based on their source address,port.

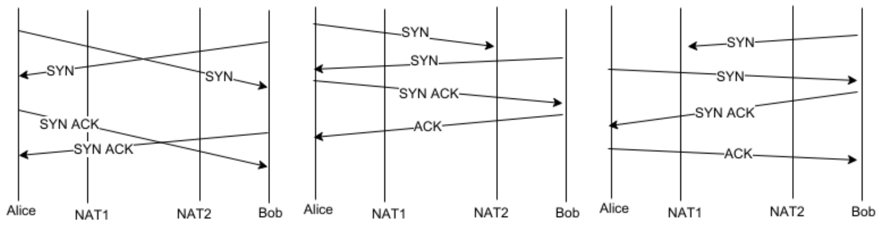
   A single active socket can be used to make a connection to a single destination at a time. To make another connection to a different destination through the same port, a new active socket has to be created but when attempting to bind this socket to the port of the previously existing one the OS returns a exception typically saying that the socket is already in use. This functionality of simultaneously connecting to two different destinations from the same port is required by Alice and Bob (one to the server and one to the opposite peer). They also require the ability to listen for new connections on the same port. Fortunately, all major operating systems support a special TCP socket option, commonly named *SO_REUSEADDR*, which allows the application to bind multiple sockets to the same local endpoint as long as this option is set on all of the sockets involved. Hence, Alice and Bob must use this option on all the sockets which they create for this purpose and bind them to a single port.

2. Normally TCP connections are established using a 3-way handshake sequence (SYN, SYN ACK, ACK). During the Hole punching process when Alice and Bob try connecting to each others public address,port pairs, they are essentially sending a SYN packet each to each other. Just like in the UDP case as explained previously, depending on the exact timing of events different things can happen but finally at least one of the SYN packets will reach their target destination without begin dropped by any middlebox. Looking at this from the perspective of the peer which received the SYN (let us assume Alice without loss of generality), she had previously sent a SYN to Bob and hence would be expecting a SYN ACK but instead receives a SYN from Bob instead. The TCP State Machine was luckily designed keeping this odd case in mind. It is formally known as a *"**TCP Simultaneous Open**"*. The standard specifies that Alice on receiving that SYN from Bob, will send him a SYN ACK in which the SYN is a replay of the previous SYN

it had sent Bob, and the ACK is an acknowledgement of Bob's SYN. Looking back at the possible cases taking this into consideration we may see the following:

- Alice observes a Simultaneous Open but Bob observes a normal 3-way handshake
- Alice and Bob swapped in the above statement
- Both Alice and Bob see a simultaneous open.

The time sequence of events in the 3 cases are depicted in the following figure:



Just like in the case of UDP, TCP Hole punching also requires the middleboxes to use endpoint-independent mapping, though unfortunately this condition is no where near sufficient to ensure success. There are many other factors which need to be taken into consideration. In the above analysis, we were not considering the behaviour of the NAT boxes with respect to TCP. Some of the problems which could come up are:

- The NAT box might enforce that a SYN ACK must be received after a SYN and blocks other any other combination of TCP flags. This can happen when the NAT software is not designed to support simultaneous open.
- Some NAT boxes on receiving a SYN, instead of simply discarding it explicitly respond to the sender with a RST. This would lead to termination of the senders connection attempt, and corresponding destruction of state in the NAT middlebox, hence not allowing the other SYN to come in.
- Some NAT boxes send ICMP error messages which can create problems similar to the ones mentioned in the above point.

Some researchers have published papers proposing schemes which solve these and other problems: STNUT [Guh], NATBlaster [Big+05], P2P NAT, each of which have their own trade offs. The authors of [GF05] undertake a detailed analysis of these and other techniques and perform a practical measurement study in which they "claim" to have achieved 88% average success rate for TCP connection establishment with NATs in the wild. Outside of literature, there don't seem to be any popular peer to peer applications which actually do TCP NAT traversal, and this is mostly due to the complications involved and the lack of understanding within the developer community.

2.4 *Port prediction*

In the above described Hole Punching technique, Alice and Bob are using the login server to find out each others public IP address, port pairs. It is possible to do this without the help of an external server, if the NAT uses a mapping scheme which follows a predetermined pattern. For example some NATs uses a port preserving mapping scheme, i.e the NAT tries to map an internal source port number to the same external source port number (This may not be possible if a conflicting session is already present in the NAT, in which case it has to allocate a different port). If Alice and Bob know each other's public IP address's and have an agreement about which ports they are going to use, they can start sending packets to each other to start the hole punching process. In reality though, they still need some channel to communicate their IP addresses and decided port numbers, and hence some sort of server is still required.

2.5 *Relay*

When the NAT middleboxes involved are Symmetric (endpoint dependent mapping), hole punching may not work. In these unfortunate cases as a fallback mechanism, the VoIP stream can be relayed through a server on the public internet. This leads to heavy loads on the relay server, and high latencies in voice transmission, as the packets are not following the optimal path. Hence, relay should only be used as a last resort, when no other NAT traversal mechanism works.

3 STANDARD PROTOCOLS

The previous sections have given a conceptual description of some NAT traversal techniques, along with some high level implementation details. A real life implementation of NAT traversal would however need to specify several other details such as the exact format of messages used for communication. We have also been discussing NAT traversal with respect to a simple two client one server VoIP application, whereas in reality we might have to deal with far more complex topologies. NAT itself has no standard specifications but in contrast several NAT traversal protocols have been standardized to avoid the redesign and reimplementation of these tedious details by application developers. Some of these useful standard protocols will be discussed in this section.

3.1 *Session Traversal Utilities for NAT (STUN)*

STUN (Session Traversal Utilities for NAT) is a standardized set of methods and a network protocol to allow an end host to discover its public IP address if it is located behind a NAT. The STUN protocol allows applications operating behind a network address translator (NAT) to discover the presence of the network address translator and to obtain the mapped (public) IP address (NAT address) and port number that the NAT has allocated for the application's User Datagram Protocol (UDP) connections to remote hosts. The protocol requires assistance from a third-party network server (STUN server) located on the opposing (public) side of the NAT, usually the public Internet.

STUN is a lightweight clientserver network protocol [J Ra]. Its purpose is to allow an application running on a host to determine whether or not it is located behind a network device that is performing address translation. The basic protocol operates essentially as follows: the client sends a message (known as a binding request) to a STUN server on the public Internet. The STUN server responds with a success response that contains in its payload the IP address and port of the client, as observed from the server's perspective. The result is obfuscated through XOR mapping. This is because some very nasty NAT devices scan the entire payload of packets and try to translate all IP address instances which they see.

STUN messages are usually sent using UDP, but in essence it is transport independent and can even be sent over TCP or TLS if security is required. An application may automatically determine a suitable STUN server for communications with a particular peer by querying the Domain Name System (DNS) for a particular record. Endpoint independent mapping is clearly a necessary condition for STUN to achieve its functionality.

Several free to use public STUN servers which are maintained by some organizations/companies exist on the internet. (For example one of Google's public STUN servers: *stun.l.google.com:19302*).

3.2 *Traversal Using Relay for NATs (TURN)*

TURN is a protocol which aids in the traversal of NATs using Relay servers. It enables hosts to control the operation of the relay and to exchange packets with their peers using the relay. The specification allows a host behind a NAT (called the TURN client) to request that another host (called the TURN server) act as a relay. The client can arrange for the server to relay packets to and from certain other hosts (called peers) and can control aspects of how the relaying is done. The client does this by obtaining an IP address and port on the server, called the relayed transport address. When a peer sends a packet to the relayed transport address, the server relays the packet to the client. When the client sends a data packet to the server, the server relays it to the appropriate peer using the relayed transport address as the source. The steps that a client must follow to send messages through the relay are as follows [J Rb]:

- The client uses TURN commands to create and manipulate an ALLOCATION on the server. An allocation is a data structure on the server. This data structure contains, amongst other things, the **RELAYED TRANSPORT ADDRESS** for the allocation. The relayed transport address is the transport address on the server that peers can use to have the server relay data to the client. An allocation is uniquely identified by its relayed transport address.
- Once an allocation is created, the client can send application data to the server along with an indication of to which peer the data is to be sent, and the server will relay this data to the appropriate peer. The client sends the application data to the server inside a TURN message; at the server, the data is extracted from the TURN message and sent to the peer in a **UDP datagram**.
- In the reverse direction, a peer can send application data in a UDP datagram to the relayed transport address for the allocation; the server will then encapsulate this data inside a TURN message and send it to the client along with an indication of which peer sent the data.

Note that since the TURN message always contains an indication of which peer the client is communicating with, the client can use a single allocation to communicate with multiple peers. Essentially it is as if the relayed transport address is a public IP, port pair which the client owns, and hence can use it to receive packets from other peers who could be behind other NATs.

TURN always uses UDP between the server and the peer. However, the exchange of TURN messages between the client and server can happen over any of UDP, TCP or TLS over TCP.

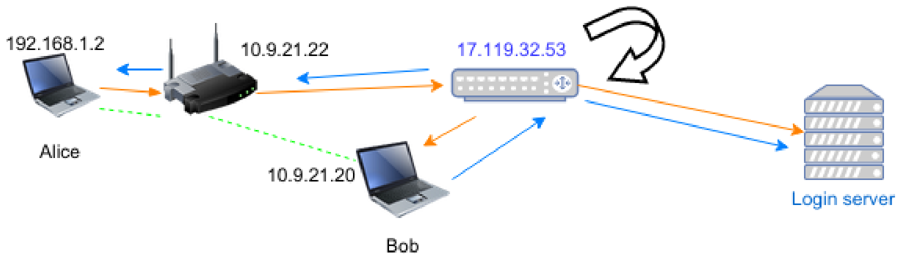
TURN servers have to endure a traffic load which is much higher that what a STUN server would typically have to handle. This is because all the packets exchanged during the entire peer to peer session pass through the TURN server. Hence the cost of setting up and maintaining TURN servers is very high making it the least preferable option for NAT traversal.

3.3 *Interactive Connectivity Establishment (ICE)*

To get some motivation behind the need for the functionality provided by the ICE protocol, we shall further analyze an earlier mentioned point: topologies other than the simple two peers behind two NATs situation which as analyzed in the previous section. To start off simple, lets consider the case of two peers behind the same NAT middlebox, say Alice (IP: 192.168.1.2) and

Bob (IP: 192.168.1.7) who are connected to the WiFi network (192.168.1.0/24) hosted by a NAT enabled router (IP: 192.168.1.1). Now if Alice and Bob want a peer to peer media stream between each other, the optimal path would be to send packets directly to each other over the local WiFi network. Given each others IP address, port pairs how would the figure out whether or not they are in the same local network. A misleading solution is to have them decide by checking whether or not their IP addresses lie in the same subnet. This will not work as they could very well have IP addresses which appear to belong to the same subnet but are actually located in 2 completely different private networks behind NATs. (as private IP addresses are not unique) The only way to know is for Alice and Bob to actually try and send packets to each others private address,port pairs. Success would indicate that they actually belong to the same local network and can then directly send packets to each other. Hence, before attempting to try things like hole punching the peers should first try to send packets to each others private addresses.

Another interesting set of topologies arises when peers are behind multiple levels of NAT instead of just one. Consider the scenario shown in the following figure:



The optimal peer to peer communication path between Alice and Bob is shown in dotted line. If they try to send packets to each others private address, port pairs they will obviously fail. Then they will try the normal Hole punching process by sending packets to each other's public address, port pairs. Note that these public address, port pairs are the ones seen from the perspective of the VoIP server (i.e the ones after translation by the upper level NAT box). So lets say Alice sends a packet to 17.119.32.53:5467 (Bob's public address). The packet would first cross Alice's NAT and then reach the upper level NAT. Upon receiving this packet the upper level NAT is in a slightly odd situation. It has received a packet from the private side of it's network which is destined to NAT device's own public IP address. If the NAT device continues with the translation and sort of reflects the packet back to itself, then the Hole Punching process can continue as usual without any problems. Such behaviour of the NAT device is called "hairpinning" (since the packet follows a path similar to the curve of a hairpin) and is essential for NAT traversal to be successful. Even if the NAT device supports this behaviour, the media packet stream after hole punching takes a suboptimal path. If the NAT device does not support hairpinning, hole punching will not work and peers will have to resort to using a relay (perhaps with TURN).

The above two examples demonstrate the differences in NAT traversal behaviour for different topologies. Hopefully this motivates the need for a more general protocol which can deal with all these different cases. This is exactly what ICE is for. It is a unified protocol which combines several NAT traversal techniques to establish a peer to peer flow of packets along the best possible path.

In the hole punching discussion in the previous section, we had described an example mechanism by which peers were sending call requests and responses through the server. In real life applications this is done using specialized protocols called offer/answer protocols (example: the Session Initiation Protocol (SIP)), whose sole purpose is the establish a session of communication between peers. ICE has been designed so that it can be used on top of any protocol utilizing the offer/answer model, and is commonly used on top of the Session Initiation Protocol (SIP). (i.e it essentially acts as an extension of an offer/answer protocol) The protocol is complex, and in fact took 6 years to gain RFC [Ros] status after it was first proposed. There is not much literature available explaining the design decisions made by the developers of the protocol, but it is known to work well in practice. The following is a vague description of the process to be followed to establish a peer to peer flow:

ICE make use of STUN and TURN protocols as tools during the traversal process, and honors the principle that media should be relayed only in the worst case when there are no other possible options. The clients using the protocol do not require any prerequisite knowledge such as the type of NAT they are behind or the topology of the network to which they are connected.

Each peer has a variety of candidate TRANSPORT ADDRESSES (combination of IP address and port for UDP) it could use to communicate with another peer. These might include:

- A transport address on a directly attached network interface (The peer could be multihomed i.e connected to the Internet via more than one interface with different IP addresses. The interfaces could also be virtual, which is the case when the peer has a VPN connection)
- A translated transport address on the public side of a NAT (a "server reflexive" address)
- A transport address allocated from a TURN server (a "relayed address").

While communicating with another peer some of these may or may not work depending on the exact topology. For instance, if both peers are behind different NATs, their directly attached interface addresses are unlikely to be able to communicate directly. Lets assume two peers who are using a VoIP application, a caller and a callee along with a public VoIP server (for connection establishment). The steps of the ICE protocol are vaguely as follows:

1. **Allocation** - The caller first gathers all of it's candidates addresses. It fetches the directly attached interface addresses using system calls. Server reflexive addresses are fetched by contacting a public STUN server. Relay addresses are obtained by sending TURN ALLOCATE messages to a TURN server.
2. **Prioritization** - The caller assigns priority numeric priority values to each of the candidates fetched in the previous step, using a formula which takes many parameters into consideration. The calculated priority value is a positive integer between 1 and (2ˆ31 - 1). For example, directly attacked network interface addresses are likely to get higher priority than relayed candidates.
3. **Offer encoding** - The gathered candidates along with their priority values are encoded into an offer message of the offer/answer protocol which the VoIP application is using. The offer is then sent to the VoIP server, which then forwards it the the callee.
4. The callee receives the offer and starts his own allocation, prioritization and encoding process similar to Step 1,2,3 in which he collects all his candidates, calculates their priorites and encodes them into and answer message which he sends to the server, which eventually reaches the caller.
5. **Verification** - Now both peers have each other's candidates. Each of them pair their own local candidates with the candidates from the remote party. The list is pruned for duplicates, and at the end both peers will end up having the exact same list of pairs. Then they calculate priority values for each of these candidate pairs using another

formula based on the priority of each of the candidates. The peers then start trying each of the candidate pairs in decreasing priority order with time gaps of 20ms. To do this they send STUN binding requests from the source address to the destination address of the candidate pair. Upon receiving a STUN request from the other peer, they send back STUN responses to source IP address and port of the request. If a response is received the check is considered to be successful.

6. **Coordination + Communication** - After all the checks both the endpoints will have the same list of valid candidates. The caller decides which of the candidate pairs to use and sends a "use candidate" message to the callee who then acknowledges, and finally they start exchanging media packets using the chosen candidate pair.

4 REAL APPLICATIONS

4.1 *Skype*

Skype is a peer-to-peer (P2P) VoIP client that is presently owned my Microsoft. Skype allows its users to place voice and video calls and send text messages to other users of Skype clients. Skype claims that it can work almost seamlessly across NATs and firewalls and has better voice quality than other VoIP clients. It's exact architecture is kept secret and not open to the public. Researchers have studied it's working by analyzing packet traces and other visible communication [BS06]. Skype uses an overlay network in which there are two types of nodes, ordinary hosts and super nodes (SN). Any node with a public IP address having sufficient CPU, memory, and network bandwidth is a candidate to become a super node. It is conjectured that it uses a variant of STUN to perform UDP hole punching. Instead of maintaining a single global STUN server, it decentralizes the functionality across it's super nodes. It also uses a TURN like relay protocol as a fallback option. All peer to peer communication happens over UDP. For tranfer of things like text messages and files, Skype implements it's own reliability mechanisms over UDP

4.2 *WebRTC*

WebRTC (Web Real-Time Communication) is an API definition drafted by the World Wide Web Consortium (W3C) that supports browser-to-browser applications for voice calling, video chat, and P2P file sharing without the need of either internal or external plug-ins. Normally to participate in peer to peer applications, hosts need to have specialized client software (as is the case with Skype). WebRTC essentially eliminates the need for this, by allowing web based applications to directly establish peer to peer communication through the web browser. It features inbuilt NAT traversal functionality. This is achieved using the ICE protocol on top of SIP (Session Initiation Protocol). It can be configured to use public STUN and TURN servers during the ICE process. The bidirectional channel of communication between online peers and login server which is required during call setup is usually implemented using the WebSockets protocol which allows bidirectional communication between clients and servers over TCP. For P2P multimedia streams it uses RTP (Real-time transport protocol) and RTCP (RTP Control Protocol), both transported over UDP. For P2P data channels (like file transfers), it uses the STCP (Stream Control Transport Protocol) which has congestion, flow control and also configurable reliability parameters. STCP data is also sent over UDP.

REFERENCES

[Big+05] Andrew Biggadike et al. "A Perrig, NATBLASTER: Establishing TCP connections between hosts behind NATs". In: *in proceedings of ACM SIGCOMM Asia Workshop*. 2005.

[GF05] Saikat Guha and Paul Francis. "Characterization and Measurement of TCP Traversal Through NATs and Firewalls". In: *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*. IMC '05. Berkeley, CA: USENIX Association, 2005, pp. 18–18. URL: `http://dl.acm.org/citation.cfm?id=1251086.1251104`.

[BS06] S. A. Baset and H. G. Schulzrinne. "An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol". In: *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*. Apr. 2006, pp. 1–11. DOI: `10.1109/INFOCOM.2006.312`.

[Bry] Dan Kegel Bryan Ford Pyda Srisuresh. *Peer-to-Peer Communication Across Network Address Translators*. URL: `http://www.brynosaurus.com/pub/net/p2pnat/`.

[Guh] Saikat Guha. *STUNT - Simple Traversal of UDP Through NATs and TCP too*. URL: `http://nutss.gforge.cis.cornell.edu/pub/draft-guha-STUNT-00.txt`.

[J Ra] P. Matthews J. Rosenberg R. Mahy. *Request for Comments (RFC): 5389*. URL: `https://tools.ietf.org/html/rfc5389`.

[J Rb] P. Matthews J. Rosenberg R. Mahy. *Request for Comments (RFC): 5766*. URL: `https://tools.ietf.org/html/rfc5766`.

[P Sa] K. Egevang P. Srisuresh. *Request for Comments (RFC): 3022*. URL: `https://tools.ietf.org/html/rfc3022`.

[P Sb] M. Holdrege P. Srisuresh. *Request for Comments (RFC): 2663*. URL: `https://tools.ietf.org/html/rfc2663`.

[Ros] J. Rosenberg. *Request for Comments (RFC): 5245*. URL: `https://tools.ietf.org/html/rfc5245`.