Large Scale Symmetric NAT Traversal with Two Stage Hole Punching

I have a feature that I would like implemented but that I do not have the means to implement myself. It is an addendum to the STUN protocol with ordinary UDP hole punching, but for symmetric NAT traversal. It is based on information provided by these three publications:

1. http://tools.ietf.org/id/draft-takeda-symmetric-nat-traversal-00.txt
2. https://www.goto.info.waseda.ac.jp/~wei/file/wei-apan-v10.pdf
3. http://journals.sfu.ca/apan/index.php/apan/article/view/75/pdf_31

Ultimately, this technology would allow for direct peer-to-peer applications to work over large-scale, symmetirc NAT. Mobile applications would benefit because they have high bandwidth (20+ Mb/s upload bandwidth on 4G networks) and high latency, and with this technology it would be possible for them to stream data to one another at full bandwidth (20 Mb/s quality), without the latency hit associated with relaying of data.

*** Background Information ***

First, before reading this paper, it is important to understand how the two major hindrances to peer-to-peer networking connection operate. For the purpose of this paper, they are Port-restricted cone NAT and Symmetric NAT. Port-restricted cone NAT is more common on home and small business internet networks where as Symmetric NAT is more common on large scale networks (large scale nat), such as 2/3/4G cellular networks. Ordinary UDP hole punching works with Port-restricted cone NAT, but it does not work with Symmetric NAT because ordinary UDP hole punching is based on the presumption that the user's public port number (their port number of origin of outgoing

UDP packets from the perspective of other computers outside their local network) and their public IP address (their ip address of origin of outgoing UDP packets from the perspective of computers outside their local network) do not change with subsequent connections made from the same local socket bound to the same private (internal) port on the host machine. That was a very long sentence. For the purpose of this paper, it might be better to use a pictorial representation to explain things that involve senders and receivers with public and private port numbers.

Sample picture of STUN with ordinary UDP hole punching between Client-A and Client-B who have Port-restricted cone NATs:

Client-A vs. STUN-Server

privatePort:		pubPort:		pubPort:		privatePort:
1000	-->	2006	-----> OPEN__	3478*	-->	3478
1000	<--	2006	REPLY <-----	3478	<--	3478

Client-A vs. Client B (Continuation)

privatePort:		pubPort:		pubPort:		privatePort:
1000	-->	2006	----->__OPEN__<-----	3001	<--	1000
1000	-->	2006	<----- HIT ----->	3001	<--	1000
			(Connection established)			

* This picture may be confusing. Let's break down what is happening row by row. Client-A binds to private port (priv:) 1000. From that socket bound to priv port 1000, Client-A sends a UDP packet (we are only dealing with UDP hole punching because it has a higher success rate and simpler implementation than TCP hole punching) to port

3478 on a STUN server. Now port-restricted cone NAT does not allow unsolicited packets in - it is "CLOSED" to unsolicited packets. In order for a packet to be allowed in, the port-restricted cone NAT must perceive the incoming packet as a reply to some outgoing packet that was sent earlier. The way it does this is it looks at the source port, source address, destination port, and destination address of packets that go in and out. If a packet goes out and it has a destination address and destination port different from any of the other outgoing packets, the NAT (router) creates a NAT table entry saying that a reply from that destination port and destination address is expected to arrive from the receiver. In this case, Client-A's NAT sees that a packet is coming from source port 1000 heading for destination port 3478 (we're assuming for the sake of simplicity that the ip addresses are fixed, so Client-A can have any arbitrary ip address), and the NAT creates an "OPEN"-ing (in the direction of the "__" double underscore) for a "REPLY" from the STUN server coming back to public port 2006 on the NAT. These "OPEN"-ings can be represented as non-bidirectional (one-way) NAT table entries.

Now this non-bidirectional NAT table entry exists and the NAT is waiting for a reply. If a reply takes too long to come in, the non-bidirectional NAT table entry may go away and it would be like no packet was sent. In that case, the NAT would change from "OPEN" to "CLOSED" on port 2006 (if you need to hold a pathway OPEN for an extended period of time, a repeated stream of packets would need to be sent to the same destination port and address in order to keep the NAT table entry from expiring). But this hasn't happened yet, so the STUN server can still get a reply through. This REPLY (second row) goes from the destination port specified in the NAT table entry created by the previous outgoing packet back to the source port specified in the NAT table entry created by the previous packets, so the NAT lets it through. This reply contains information that it would need in order to be able to try and reach Client-B. For the sake of simplicity, lets assume this reply tells Client A that Client-B is at ip address

"123.456.789.0" and has been sending out packets from private port 1000 to public port 3001 through it's NAT.

Now the actual hole punch occurs. In the third row, both Client-A and Client-B make "OPEN"-ings for one another in their NAT tables by sending one another packets at roughly the same time. Finally, in the fourth row, these packets go through the openings created in one another's NAT tables and they make it through to one another, ultimately "HIT"-ing their destinations. Once this has happened, they can reply to these packets, resend, go back and forth, and ultimately maintain an active Internet connection with one another. Their NAT table entries have changed from being "non-bidirectional" (one-way) to being "bi-directional" (two way), and the NAT acknowledges that they have a stateful Internet connection between this port and that port, from this ip address to that one. Now all that Client-A and Client-B have to do is communicate back and forth, occasionally sending "keep-alive" packets during quiet periods to make sure their NAT table entries don't expire and get removed from the NAT table.

Now for the difference between symmetric and port-restricted NAT. With Port-restricted cone NAT, if you bind to port 1000 and send an outgoing packet that the NAT maps to public port 2006, and you keep doing that from the same port on your computer, the mapping will consistently stay the same and anyone who tries to connect to you from a different ip can predict your port number because of it. But with symmetric NAT, your public port number changes with every computer you connect to, even if you use the same socket bound to the same address. Other people (in this case Client B) cannot predict your port number. In some cases, they can't even predict your ip address because some ip addresses change with each new outbound connection (not the norm, but it is possible, especially for a NAT with multiple IP addresses that is incrementing to the next one). Because of these changes in port number (and even ip address) for source packets

coming out of a given socket, ordinary hole punching techniques fail for symmetric NAT, also called carrier grade NAT or large scale NAT. The terms "symmetric", "carrier-grade", and "large-scale" often have the same literal meaning when it comes to their behavior, but they have different connotations because some non-service-provider grade NATs can still technically be symmetric. If a NAT that allocated a new port for each connection was small, it would still technically be a symmetric NAT, but if it were a 2/3/4G network tower, it would be a large scale NAT. For the purpose of this paper, I will say that all large scale NATs are symmetric but not all symmetric NATs are large scale NATs (large scale NAT can be considered a sub-classification of symmetric NAT).

Symmetric NAT Example:

Client-A vs. Client-B (Symmetric NATs on both sides)

privatePort:		pubPort:		pubPort:		privatePort:
1000	-->	2017*	-----> OPEN__	3001		1000
1000		2006	__OPEN<-----	3003**	<--	1000

*Client-A opens up port 2017. The packet is dropped when it reaches port 3001.

** Client-B opens up port 3003. The packet is dropped when it reaches port 2006.

Client-A vs. Client-B (Symmetric NATs continued)

privatePort:		pubPort:		pubPort:		privatePort:
1000		2006*	MISS<-----	3003	<--	1000
1000	-->	2017	----->MISS	3001**		1000

* Client-B was expecting port 2006 on Client-A to be OPEN. It was not.

** Client-B was expecting port 2006 on Client-A to be OPEN. It was not.

Client-B undershot by (2017 - 2006) 11 ports.

Client-A undershot by (3003 - 3001) 2 ports.

They both miss.

(No connection established)

***** Symmetric NAT Traversal Implementation *****

In the three papers at the beginning of the paper (which you should read first), the traversal technique was based on first determining if a NAT was symmetric (if it had different public ports or public addresses from the same destination port). If that was the case, Client_A would launch packets with short time-to-live values (between 1 and 5 with a default value of 2) that will not reach their destination in order to create open (non-bidirectional) NAT table entries to allow packets from the public Internet to enter. The other client (Client_B) would then fire packets with long time to live values to make their way into the holes in the NAT table created by the packets with short time-to-live values sent by Client_A. If a short time-to-live packet exits a given port and then a long time-to-live packet enters through that port (heading in opposite directions), a connection (a bidirectional NAT table entry) is created and Client_A and Client_B can send packets back and forth through that port. The NAT table entries created by the short time-to-live packets not only make space in the NAT table for the long time-to-live packets to come through, but also prevents the long time-to-live packets from repeatedly missing, which may inadvertently trigger a security feature called flooding protection. Flooding protection may block packets from a sender who repeatedly sent packets into a port/address where no NAT table entry existed.

***** Migrating from ordinary UDP hole punching to multi-UDP hole punching *****

That is all well and good, but it isn't backwards compatible with the way things are

done now using STUN and ordinary UDP hole punching because ordinary UDP hole punching just gets a single address and port and sends a packet to only that address and that port in order to establish a connection while symmetric udp hole punching always targets many ports. In addition, with the multi-UDP hole punching technique described in the papers, one side (the receiver) must send short time-to-live packets to open the NAT while the other side (the sender) must send long time-to-live packets to penetrate the NAT, but in ordinary UDP hole punching, both sides send long time-to-live packets and it does not matter who is the sender and who is the receiver. Ordinary UDP hole punching is sender-receiver agnostic while symmetric-UDP hole punching is not. This would not be acceptable for a general purpose replacement for ordinary UDP hole punching not only because some applications do not care which side is offering and which side is answering but because having one side not even attempt to send any packets through to establish a connection may reduce the success rate. Ideally, ordinary UDP hole punching ought to be a special case of the multi-udp-hole-punching such that an ordinary UDP hole punch can be achieved by tweaking a variable in the multi-udp-hole-punch algorithm.

More specifically, I want ordinary UDP hole punching to effectively be the same as multi-udp-hole-punching when the breadth (width in terms of number of ports covered) of packets sent out is equal to one. The breadth can be measured by the number of short time-to-live packets sent out on various ports or the number of unique, open NAT table entries created. I want this "breadth" variable to be modifiable depending on whether the user wants an ordinary UDP hole punch (covering a single port) or a narrow band UDP hole punch or a broad band UDP hole punch (covering several hundred ports).

Note that in the traditional multi-udp hole punch technique, hundreds of packets are sent out on either end, with one end sending out short TTL packets and the other side sending out long TTL packets.

The traditional multi udp hole punch algorithm (as described in the above papers, breadth limited to three ports, with irregular increment due to other users on the same router):

Short-TTL				Long-TTL		
privatePort:		pubPort:		pubPort:		privatePort:
1000	-->	2006	OPEN__	7005**		
		2005	MISS <-----	7004	<--	1000
1000	-->	2004	OPEN__	7003		
		2003	MISS <-----	7002	<--	1000
1000	-->	2002	HIT <-----	7001	<--	1000
		2001	MISS <-----	7000	<--	1000
1000	-->	2000	OPEN__	6999		

Key:

--> Short arrow denotes a packet going from a computer to a router without crossing the public Internet.

-----> Long arrow denotes a packet going from one router to another by crossing the public Internet.

- OPEN denotes an open one-way entry in the NAT table (for example, public port 2006 on the top left is OPEN to an incoming packet from port 7005).

- MISS denotes a failed attempt to send a packet into an open one-way entry in the NAT table (for example, a packet sent from public port 7004 on the upper left failed to

get through when it arrived at public port 2005).

- HIT denotes a successful attempt at sending a packet into an open one-way entry in the NAT table (for example, a packet sent from public port 7001 got through to private port 1000 after successfully arriving at public port 2002).

** Note that NATs can increment their external port numbers, decrement, or jump around with each new outbound connection. In this case we assume a steady increment of the external port number with each increment of the destination port number.

So in the above example the receiver opened four ports and the sender had three misses and one hit ultimately establishing a connection at port 2002. Note that if too many MISS-es occur, flooding protection may be triggered, which can block any subsequent packets from the sender for a fixed time duration such as 100ms or 1 seconds.

Note that if simplified to a width (breadth) of one packet, the result looks like ordinary UDP hole punching, where the public (server-reflexive) ip address of the receiver is 2002 and the public (server-reflexive) ip address of the sender is 7001:

```
Short-TTL (receiver)          Long-TTL  (sender)
priv:     public:             public:     priv:
                  HIT
1000 -> 2002       <-----     7001  <- 1000
                 REPLY
                -----> (Connection established)
```

The only difference in between this means of hole punching and ordinary UDP hole

punching is that the receiver is sending a short TTL packet which doesn't reach the sender while the sender is sending long TTL packets which make their way to the receiver. In ordinary UDP hole punching, both sides would be sending packets with long (normal) TTL values.

Note that we have a problem here. One side has to be aware that they are the recipient of the connection (the one who sends the short TTL packet and opens up the hole in their NAT) while the other side has to be aware that they are the establisher of the connection (the one who actually has to get a packet through). In ordinary UDP hole punching both sides are equal but in symmetric UDP hole punching they are unequal. Let's make them equal so they they do not need to be aware of who is the receiver and who is the sender.

To make them equal, both sides can play the role of the receiver by sending packets with short TTL values. Then, both sides can play the role of the sending (or establisher) by sending packets with long TTL values. So we have something like this:

Step #1: Opening the NAT Table

Short-TTL		Short-TTL
priv: public:		public: priv:
1000 -> 2002	__OPEN__	7001 <- 1000

Step #2: Penetrating the NAT

Long-TTL		Long-TTL
priv: public:		public: priv:
1000 -> 2002 ----->		<----- 7001 <- 1000
1000 -> 2002 <-----	HIT	-----> 7001 <- 1000

(Connection established)

So both sides first send one packet with a short TTL value and then one packet with a long TTL value.

But what if we increase the breadth to three packets instead of one? Which port should be targeted first?

Well since the most important port (our best guess) is the public, server-reflexive port, that port should be targeted first so that if flooding protection is activated through subsequent MISS-ed packets, the flooding protection would not block the server-reflexive port and a connection may still be established, although no more packets could be received from the sender until the flooding protection stops. The receiver of the packet could still send back a response to the sender, although the sender may have to wait a second before their packets can pass through again.

So we start with the public, server-reflexive port, and work our way outward, where the width is controlled by the user specified "breadth" variable. So if the width was three, we might do something like this:

Step #1: Opening the NAT Table (breadth = 3)

Short-TTL		Short-TTL
priv: public:		public: priv:
1000 -> 2002	__OPEN__	7001 <- 1000
1000 -> 2003	__OPEN__	7002 <- 1000
1000 -> 2001	__OPEN__	7000 <- 1000

Step #2: Penetrating the NAT (breadth = 3)

Long-TTL		Long-TTL
priv: public:		public: priv:
1000 -> 2002 ----->		<----- 7001 <- 1000

1000 -> 2003 -----> <----- 7002 <- 1000

1000 -> 2001 -----> <----- 7000 <- 1000

* Note that this diagram is technically incorrect and only meant to help convey the idea that we first start with the server reflexive ports (2002 and 7001), that we work our way outward from the server reflexive ports, and that Long-TTL packets will be sent after short TTL packets. The Long-TTL packets are targeting the same ports as the Short-TTL packets, so since they have the same destination as the Short-TTL packets, we can assume that the Long-TTL packets will come from the same public ip addresses and ports as the Short-TTL packets, even if the NAT is symmetric. The reason that the above diagram is incorrect is that even if the destination port is changing in a back-and-forth pattern (7001, 7001+1, 7001-1, 7001+2, 7001-2, etc.), that in no way guarantees that the public port of origin of these packets will follow the same pattern, as it does in the above diagram.

A less unrealistic diagram:

Long-TTL		Long-TTL	
priv:	public:	public:	priv:
1000 ->	2002 ----->		7001
1000 ->	2003 ----->		7002 (7001 + 1)
1000 ->	2004 ----->		7000 (7001 - 1)
	2002	<-----7001	<- 1000
	2003	<-----7002	<- 1000
	2001	<-----7003	<- 1000

* This diagram is still technically incorrect, but it emphasizes the fact that even though the destination ports are changing in a back and forth pattern ((7001, 7001+1,

7001-1), the public ports assigned by the NAT seem to follow a different pattern, in this case they increase by one (2002, 2003, 2004). Note that although in this example the "breath" of packets sent out is three for both short and long time to live packets, it might be a good idea for the "breath" of short time-to-live packets to be "wider" than that of long time-to-live packets to reduce the number of misses and avoid triggering flooding protection. For the short TTL packets to cover a wider range of ports and open more NAT table entries than necessary just in case.

Dealing With Local Addresses

If the address is on the local Internet (i.e. if the destination has the same server-reflexive public ip address as the source), a different procedure must be taken. A TTL value of 2 might cause the short lived packets to reach their destination, which is undesirable. In this case, the TTL should be reduced to one and the local ip address(es) should be checked in addition to the public ip address/port, which may still be reachable in the case or hairpin translation (when public ip addresses/ports are translated by the router to private ip addresses/port when the source and destination have the same public ip address). In the case of nested NAT (where a TTL value of 2 would not be enough to make it through all the layers of routers), it should be possible (their should be a function provided) to allow the user to obtain the number of hops between the source and the destination and to set the TTL value to a non-default number equal to the number of hops divided by two.

Pseudo-code:

Ultimately, this has to be implemented as a native library using Unix sockets so that it can be easily linked to Mac, Linux, Android, and iPhone (but not Windows because

desktop applications do not need to traverse symmetric NAT). It must be able to work with ordinary STUN clients. The native library needs to export an interface with some basic functionality:

```
/*
 * Attempts to make a connection to the given IP address / port via standard UDP hole punching (breadth = 1).
 * This method is the same as ordinary UDP hole punching and is equivalent to obtainSocketTo(IP_address/port, 1).
 */
udpSocket obtainSocketTo(IP_address/port);
void punchThroughTo(udpSocket, IP_address/port);

/*
 * Attempts to make a connection to the given IP address / port via multi UDP hole punching.
 * This method is the same as obtainSocketTo(IP_address/port, int breadth, DEFAULT_TIME_TO_LIVE).
 * @param breadth- min: 1, max: 32768.
 */
udpSocket obtainSocketTo(IP_address/port, int breadth);
void punchThroughTo(udpSocket, IP_address/port, int breadth);

/*
 * Attempts to make a connection to the given IP address / port via multi UDP hole punching.
```

```
 * @param breadth- min: 1, max: 32768. 32768 is half of 65536, so that would cover all port #'s.
 * @param timeTOLive- min: 1, max: 5 (or greater)
 */
udpSocket obtainSocketTo(IP_address/port, int breadth, int timeToLive);
void punchThroughTo(udpSocket, IP_address/port, int breadth, int timeToLive);

/*
 * Calculates the optimal time to live by obtaining the number of hops to the given IP address and dividing by 2
 */
int calculateOptimalTimeToLive(IP_address/port)
...
```

Some basic functions would be required:

1. Set the time to live for outgoing packets...

Linux C code:

```
    const int timeToLive = 2; /* could be some other value  */
    setsockopt(socket, IPPROTO_IP, IP_TTL, &timeToLive, sizeof (timeToLive));
    printf("TTL set to %d \n", timeToLive);

...
```

*** Edge Cases ***

This is all a rough draft. The devil is in the details and specifics about many edge cases,

such as what should be done when the port number is at or near the minimum or maximum port number, are left unaddressed. For example, some routers with multiple addresses are known to move to the next IP address if no port is available. In such a case, if the router is symmetric, the port number is incrementing, and the port number has reached the upper limit of 65535, it may be wise to "wrap around" to port number 1024. Another edge case would be if one device were not behind a NAT and all packets went through. In that case, advanced UDP hole punching with a breadth (width) of 100 ports would result in 100 UDP connections (UDP is connectionless, but it would result in 100 packets being received [and perhaps acknowledged] on 100 different ports). If only one UDP connection were desired, the other 99 ports would have to be ignored.

*** Addendum ***

It should also be possible instead of sending both packets with short TTL values and then packets with long TTL values, to either only send packets with short TTL values or to only send packets with long TTL values. By only sending packets with short TTL values or only sending packets with long TTL values, the interface would be compatible with the original three papers on which this draft was designed. Then again, we have three different ways hole punching can be done:

1. Each side sends a packet with a long TTL value (regular UDP hole punching).
2. One side sends packets with long TTL values and the other side sends packets with short TTL values (sender-receiver model).
3. Both sides send packets with short TTL values followed by packets with long TTL values (my method proposed in this paper).

Plus two more ways, which are based on the original papers and depend on the user having already determined that the NAT is symmetric and that it is either incrementing up from the target port or decrementing down from the target port:

4. One side sends packets with linearly ascending (or descending) port numbers with short TTL values while the other side sends packets with linearly ascending (or descending) port numbers with long TTL values (sender-receiver model).

5. Both sides send packets with ascending (or descending) port numbers with short TTL values and then both sides send packets with ascending (or descending) port numbers with long TTL values (my bidirectional model optimized for linear changes in predicted port number). Note that this method would typically still start at or near the predicted port and work outward, but it would work outward in only one direction instead of both.

A good hole punching implementation should allow for all five methods of hole punching.

Anyway, I would like a 32 bit and 64 bit Linux .so, C-99 standard-library-only source code that can be compiled into a native library for linking with either Android Studios or iOS SDK, documentation for said source code, and testing for three to five types of the hole punching methods (listed above) in order to make symmetric NAT traversal (universal NAT traversal via advanced UDP hole punching) available to everyone.

p.s. The paper "Extended UDP Multiple Hole Punching Method to Traverse Large Scale NATs" suggests that in addition to port prediction based for noticing increasing or decreasing port numbers, a packet capture method may be used to estimate the number

of newly assigned mappings and more accurately guess the port number of the target, decreasing the "breadth" of the packet barrage necessary to penetrate the NAT. If time permits, packet capturing functionality should be built into the library. Note that the "scanning method", which involves pinging every ip address on a private network to count the number of hosts, may not be practical for large scale NATs because they often do not offer small, predictable IP ranges that can easily be "pinged through".

*** Testing and Proving that it Works ***

Testing would require purchasing multiple cell phones, attaching them to multiple different cell phone towers, and using TeamViewer or some other remote desktop application to connect with them and/or with computers pre-installed with Android studio that are connected to said Android devices. Proving that it works would require having those Android devices establish peer-to-peer UDP connections with one another over said cellular networks.

Another method may simply be to obtain multiple phones from multiple different carriers using multiple differet networks and have them punch holes through to one another. Different “breadths”, port/address estimation techniques, and scanning techniques should be used to find one that is practical.