NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity*

Saikat Guha
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
saikat@cs.cornell.edu

Yutaka Takeda
Panasonic Communications
San Diego, CA 92127
takeday@pcrla.com

Paul Francis
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
francis@cs.cornell.edu

ABSTRACT

The communications establishment capability of the Session Initiation Protocol is being expanded by the IETF to include establishing network layer connectivity for UDP for a range of scenarios, including where hosts are behind NAT boxes, and host are running IPv6. So far, this work has been limited to UDP because of the assumed impossibility of establishing TCP connections through NAT, and because of the difficulty of predicting port assignments on certain common types of NATs. This paper reports on preliminary success in establishing TCP connections through NAT, and on port prediction. In so doing, we suggest that it may be appropriate for SIP to take a broader architectural role in P2P network layer connectivity for both IPv4 and IPv6.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Protocol architecture

General Terms

Algorithms, Design, Experimentation, Measurement

Keywords

NAT traversal, NUTSS, STUNT, IPv6 transition

1. INTRODUCTION

With the existence of NAT, and more recently the introduction of IPv6 and its myriad transition mechanisms, establishing network connectivity between IP hosts is more complex than originally envisioned by IPv4. The original IPv4 connectivity model of course is that respondent hosts listen at *transport addresses* (IP address + transport port number), and initiating hosts send packets to those transport addresses. DNS may be used to discover the IP address of the respondent. In some cases, the port numbers are well-known (HTTP, SMTP, etc.), but in others there must be some way for the initiating host to know which port number to use. The IP suite of protocols relegates this problem to each individual application.

The Session Initiation Protocol (SIP) [4] is a middleware signalling protocol that allows IP endpoints to negotiate whether and how they wish to communicate. SIP negotiates various parameters including port numbers, IP addresses, whether to use unicast or multicast, IPv4 or IPv6 (or some other network protocol), and details of the media stream such as type of encoding.

More recently the use of SIP has been extended to include information required to establish direct UDP, and indirect TCP connectivity between hosts behind NAT boxes and firewalls [2]. This extension of SIP, called ICE (Interactive Connectivity Establishment) relies on two new protocols being developed in the IETF, STUN and TURN. STUN [5] allows a host to learn the global IP address and UDP port assigned by its outermost NAT box. This address can be subsequently conveyed by SIP (under the ICE procedures) to allow direct UDP connectivity between hosts. TURN [3] allows a host to select a globally-addressable TCP relay, which can subsequently be used to bridge a TCP connection between two NATed hosts. Unlike STUN, TURN does not allow direct connectivity between NATed hosts.

To be clear, ICE is a usage profile for SIP—it is not a new protocol per se. ICE is used both to discover and convey a list of possible modes of communications (IPv6, public IPv4, private IPv4, STUN, TURN), and to test these modes.

SIP can serve this broader connectivity establishment role in part because SIP uses URIs (e.g. user@domain) to identify and discover end points, thus allowing SIP signalling messages to pass through NATs and convey the necessary information (type of NAT box, private and public addresses and ports, and so on).

We (the authors) have long been impressed with the broad ability of SIP to manage the intricacies and limitations of the network layer, including its ability to maintain sessions across IP mobility events to find users at multiple different machines, and now to deal with NAT. Although SIP is designed for establishing media streams (audio or video), it is general enough to operate with any kind of data flow, and has been used, for instance, to negotiate and maintain TCP connections across IP mobility events.

Given all this, an important question to ask is, why can't SIP be used broadly as the protocol to establish all kinds of P2P [1] communications, both data and media? Why can't operating systems come standard with a simple sockets-like network API that invokes SIP to discover what type of network-layer communications is possible

*This work is supported in part by National Science Foundation grant ANI-0338750

SIGCOMM'04 Workshops, Aug. 30+Sept. 3, 2004, Portland, Oregon, USA.

[1]By P2P, we mean broadly those hosts and applications that are not served well by the private-client/public-server model, for instance because they are behind NATs or firewalls, or because they do not wish to allow any host to communicate with them at any time.

and appropriate, and subsequently return a transport socket to the application?

There are two important criteria for determining whether a given transport or middleware protocol should be standardized, given a standard API, and perhaps even commonly supported by the OS. First, the protocol should provide basic functionality that is useful to a wide range of applications. Second, the protocol should be difficult enough that it is not trivial to provide it separately by each application.

Perhaps one answer to our question is that the socket-listen, DNS, let-each-application-solve-the-problem model has generally been adequate. Whereas this might have once been true, we believe it is no longer true. Establishing connectivity between NATed hosts, or between IPv4 and IPv6 hosts, or for that matter between IPv6 hosts that are connected to IPv4 networks and behind NAT boxes, is both difficult and broadly needed by all kinds of applications. We don't think it is appropriate to have separate distinct solutions for all of these connectivity problems.

Another possible answer to our question, and one that we confront in this paper, is that there is no known solution that provides *enough* functionality to justify a standard approach to all P2P connectivity establishment. In particular, there is a widespread perception that it is impossible in the vast majority of cases to establish direct TCP connections through NAT, and that there are too many cases where even UDP cannot be established through NAT because of the difficulty of knowing what port number the NAT box will assign. We believe both of these answers to be overly pessimistic. Indeed the authors have had preliminary success in the lab in establishing TCP connections through NATs and firewalls using a technique that we believe applies broadly. We also have broad success in the field in predicting NAT port assignments in order to establish UDP connectivity, and we believe that this success will apply equally well to TCP.

The purpose of this paper is to report these specific results, to suggest to the networking community that SIP be used generally as the standard means for negotiating and establishing network connectivity for P2P, including in support of transition to IPv6, and to discuss research issues associated with this model. Note we are not suggesting SIP for general public client-server communications. That model works more-or-less just fine as is—SIP would only add unnecessary overhead.

The remainder of this paper is structured as follows: The next section briefly describes this architectural approach, which we dub NUTSS. The following two sections describe the NUTSS components not already described by the ICE/STUN/TURN suite of protocols, namely how to do port prediction for NATs that assign different ports for every outgoing connection (so-called Symmetric NATs), and how to establish direct TCP connections through NATs. We close by discussing issues and open problems in the NUTSS architecture.

2. NUTSS ARCHITECTURE

If we are going to suggest an architecture to support a standard approach to connectivity establishment, it seems appropriate to give it a name (even if we are not responsible for the majority of its component parts). We like to call the architecture NUTSS, which stands for what we consider to be its main architectural components: NAT, URI, Tunnel, SIP, and STUNT[2].

NAT effectively extends the address space, though ideally someday IPv6 becomes ubiquitous and the architecture can be called UTSS. Indeed, when a host selects among the possible communications methods that ICE presents to it (i.e. IPv6, native IPv4, STUN, STUNT, TURN), the host selects IPv6 in preference to the others, if it is available. Therefore, as IPv6 usage spreads, it will naturally become the default method of communications. In this sense, NUTSS supports a natural *exit strategy* away from NAT.

The URI is the end-to-end naming scheme. The Tunnel component refers to the need to encapsulate some low-level protocols in UDP, specifically Mobile IP, IPsec, and even IPv6 itself. As stated already, SIP is the protocol used by hosts to negotiate network-level communications. NUTSS's usage of SIP will certainly include ICE, but will also include mechanisms to convey the TCP/UDP application to be run.

STUNT is a protocol that extends STUN to include TCP. STUN is the protocol that a host uses to determine what global address and UDP port has been assigned to it by its outermost NAT box, and to determine what kind of NAT the box is.

STUNT has not yet been specified in detail, but its TCP usage would be very similar to STUN's UDP usage. More details are given later, but the basic idea is this. The host establishes multiple TCP (or UDP) connections with a globally reachable server (the STUNT Server). The STUNT Server records the global addresses (GA) and ports (GP) assigned by the NAT, and conveys them back to the host. Using SIP, the host in turn conveys to the remote host the expected GA and GP (and any other addresses and ports it may be able to communicate with), and likewise receives the addresses and ports of the remote hosts. Both hosts then send an initial outgoing packet through the NAT addressed to the remote hosts GA and GP. The purpose of this initial packet is to establish the mapping at the NAT box, thus allowing incoming packets from the remote host. This is sometimes called *punching a hole* in the NAT box. Subsequently bidirectional packet exchange can take place.

While this approach is ugly (and some specific ugly details are discussed in this paper), much of the ugliness stems from the fact that NAT behavior is unstandardized and therefore ad hoc and unnecessarily complex. In any event, such ugliness cannot be avoided if we are to transition to IPv6 (and therefore have to deal with IPv4-IPv6 translating "NAT" boxes) so in our minds NUTSS is simply something that has to be done, and we ought to make it as painless and functional as possible. Over time, the ugly parts will disappear as NATs disappear and IPv6 takes hold. On the other hand, if IPv6 never takes hold, there will at least exist better functionality than we have today.

Finally, it is easy to imagine a socket-like NUTSS-based network API. Such an API could have the standard bind(), listen(), connect(), accept(), and close() calls, but could substitute URIs for IP addresses, and application names for port numbers. Indeed we have implemented such an API, though it has not been tested broadly.

3. PORT PREDICTION

The key to successful NAT traversal (for TCP or UDP) is of course that the remote host know which global port (GP) and global IP address (GA) has been assigned by the NAT for a given flow. The problem here is that STUNT knows what GP and GA has been assigned by the NAT *for the flow between the host and the STUNT Server*, but STUNT can only make guesses about what GP and GA will be assigned for a subsequent flow to the remote host.

Unfortunately, different NAT boxes take different approaches to port assignment (no doubt in part because the IETF for many years refused to accomodate NATs and standardize their behavior). Some NATs will repeatedly assign the same GP to a given host using a given local port (LP), even if that host has flows with multiple remote hosts. This kind of NAT is called a Cone NAT in [5], and its behavior is shown in Figure 1. Here we see that the host has

[2]Simple Traversal of UDP through NATs and TCP too

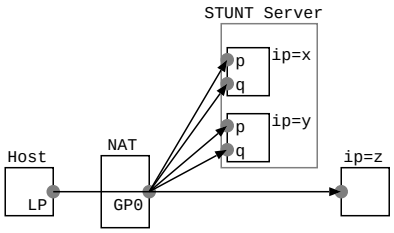

Figure 1: Cone NAT Type

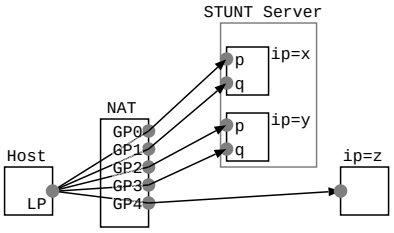

Figure 2: Symmetric NAT Type

established four flows with the STUNT server, each using the same LP, but to different IP addresses (x and y) and different ports (p and q) at the STUNT Server. The STUNT Server sees that the NAT box has assigned the same GP each time, and so it is a good bet that the NAT box will do so again when the host establishes a flow with the remote host at address z. Note that NAT boxes virtually always assign the same global address GA, so for simplicity this is not shown in Figure 1. Note also that many Cone NATs assign GP = LP if LP is not already assigned to another flow. These are called Port Preserving Cone NATs.

While it is easy to traverse Cone NATs, traversing so-called Symmetric NATs is more difficult. Symmetric NATs assign a different port for every new flow, as shown in Figure 2. This can be detected by the NAT box, because it sees a different port assignment for each flow.

Fortunately, most Symmetric NATs assign port numbers in uniform increments δp, typically of 1 or 2, and so it is possible to predict the next port number assignment [6]. For example, if GP0 through GP3 each increment by 1, GP4 is likely to be GP3+1. Some Symmetric NATs randomly choose global port values, making port prediction impossible, but these are quite rare.

3.1 Issues with Port Prediction

The correct operation of port prediction relies on a single host getting an uninterrupted series of port assignments from the NAT. Unfortunately, other hosts may be getting ports assigned at the same time. For instance, a host *H* may get assignments GP0 and GP1, but a different host may get port assignment GP2 before *H* does, with *H* subsequently getting assignments GP3 and GP4. Note that we get the same effect if a different host obtained GP2 long ago and hadn't relinquished it yet.

There are two scenarios where port assignment interruption can cause a problem. One is when the host is probing the STUNT Server to determine what kind of NAT it is behind (i.e. Cone or Symmetric, and if Symmetric, the value of δp). This case is not so bad for two reasons. First, the host can establish its TCP flows (or UDP packets) in immediate succession, thus minimizing the chance of interruption. The host doesn't need to wait for one connection to complete before starting the next connection. Second,

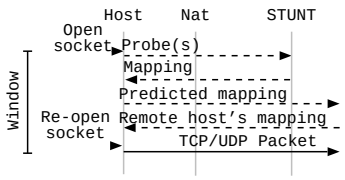

Figure 3: Window of vulnerability for port predictions

inconsistencies in the series of assignments can usually be detected (i.e. different δp for different probes).

The far more difficult scenario is where the host probes the STUNT box in order to predict the GP of the subsequent flow to to a remote host. The main problem here is that in some cases there is necessarily a gap in time between the probe flow to the STUNT box and the subsequent flow to the remote host. There are two cases to consider. In the first case, the host knows the GP and GA of the remote host before probes the STUNT Server. This is possible when the remote host's NAT is Cone, or when it is Symmetric but there are very few hosts using the NAT box (and so very unlikely that a port sequence interruption will take place). In this case, the gap is due to the time it takes for the host to receive an answer from the STUNT Server and establish the flow to the remote host. This gap is roughly equal to the RTT between the host and its STUNT Server, and so could easily be on the order of 100ms.

In the second case, the host cannot know the GP and GA of the remote host before it probes the STUNT Server. This would be the case where the remote host is also behind a heavily utilized Symmetric NAT. In this case, the host has to wait for both the answer from the STUNT Server conveying its own GP and GA, and a message from the remote host conveying the remote host's GP and GA. Essentially, both hosts must, at the same time, probe the STUNT Server, get an answer, inform the other host of the answer, and then establish the flow. This requires two round trips even where the two hosts are synchronized (i.e. start their probes at the same time), and longer to the extent that they are not synchronized as illustrated in Figure 3. Thus it is easy to imagine 300ms or greater gap between the two port assignments.

Given this, there are going to be cases (medium to large enterprise networks) where port prediction is simply not possible. Cornell, which has over 20,000 students, faculty, and staff, has 10's of outgoing TCP connections per second.

3.2 Likelihood of Failure

Given that there are failure modes, we would like to get a sense of how often failures may occur. While we don't have hard numbers, we can look at the percentage of commercial NAT boxes that are Cone, Symmetric, and so on. We looked at 9 different NAT boxes from 5 manufacturers (Netgear, Linksys, Dlink, Hawking, and Speedstream). Regarding UDP behavior, eight of the nine are Cone type, and of these, five are port preserving. For TCP, six of the nine are Cone type, with five of these being port preserving (the same five as with UDP). Of the Symmetric NATs, all of them have a $\delta p = 1$.

Given this, we can guess that port prediction may work for the large majority of home or small office NAT users (because none of these NATs have random port assignment, and because small networks don't have the port series interruption problem). Indeed in tests run by Panasonic Communications Corp., UDP port prediction through home/small office NATs worked virtually 100% of the time.

We can guess that many large enterprise users may be able to

establish connectivity through NAT because they use Cone type NATs. Nevertheless, there will be significant numbers of large enterprise users that still will not be able to establish connectivity.

We should keep in mind, however, that large institutions often block most applications at the firewall anyway. In other words, the connectivity problem is often not with NAT per se, but rather is a feature desired by the enterprise's IT organization. In addition, if an institution does want to allow P2P flows, it has the option of using a Cone NAT.

There are two points we are trying to make here to application developers who may consider using the NAT technologies described in this paper. First, while port prediction is not foolproof, it adds significantly to the population of users who could connect through NAT compared with no port prediction. Second, even if port prediction were foolproof, there would still be connectivity issues, because of firewall policy. In other words, if the P2P application developer expects fault-free P2P operation, he or she is out of luck in any event.

4. NAT TCP SOLUTION

The basic problem with establishing TCP through NATs is that, with TCP, normally one host listens while the other host initiates the connection (with a SYN packet). Unfortunately, NAT requires that there be an outgoing packet to assign the NAT port mapping (punch a hole) before any incoming packets can be received. A listening host never sends such a packet, and therefore can never receive the SYN from the initiating host.

Now it so happens that the TCP *protocol specification* allows both ends to behave as initiators by simultaneously sending SYN packets. This simultaneous operation will work with NATs as long as each SYN packet exits its respective NAT outgoing before the remote host's SYN packet arrives. This is actually not hard to achieve as long as the path between NATs is longer than the paths between each host and its NAT. The point is moot, however, because Microsoft does not allow simultaneous TCP in its implementation. Thus this isn't a meaningful option unless Microsoft changes its TCP.

Another broad approach is to allow the host to explicitly establish an address mapping or firewall rule in the NAT/firewall through some out-of-band protocol exchange. This is the approach taken, for instance, by Universal Plug-and-Play (UPnP) [1], and by the IETF midcom working group (which is also the group that produced STUN). The downside of this type of approach is that they require that these protocols exist and are enabled in the NAT/Firewall. An application developer cannot depend on either of these, and so for the time being they are not an attractive option. Something more incrementally deployable (i.e. does not require explicit cooperation from the NAT/Firewall) is preferable.

4.1 Our Approach

In this section we describe our approach of establishing TCP connections when each end-point is behind a NAT as shown in Figure 4. One design principle we followed in this approach is that, from the perspective of the NAT boxes on both ends, the TCP connection setup appears to be a normal three-way handshake (with both hosts appearing to be the initiator). This approach minimizes the chance that a NAT box (or firewall) will disallow the connection because it looks non-standard.

For simplicity, in Figure 4 we show only NATs N and M which are the border NATs that connect the private networks to the public internet. Additional NATs that may be present between the border NAT and the respective end host do not effect the protocol. Though only a single STUNT Server is shown, each host A and B may

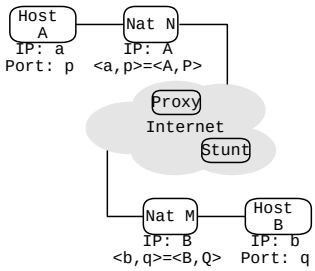

Figure 4: Network Configuration

have separate independent STUNT Servers. The STUNT Server(s) must be placed in the public internet and must be able to spoof IP source addresses. The Proxy shown in Figure 4 is required to pass messages between hosts A and B to coordinate the establishment of TCP. The Proxy could be a single box that both A and B have established communications with, or it could represent an infrastructure, for instance composed of SIP proxies, through which A and B can exchange messages.

The TCP connection setup protocol that we describe is symmetric for A and B. We describe it from A's perspective as depicted in Figure 5. Both A and B use the standard TCP stack and API provided by the OS. Note that the solid lines represent the actual TCP packets issued by the OS. The dashed lines represent messages sent between the hosts and their respective STUNT Servers, or between themselves via one or more Proxies.

At startup, Hosts A and B establish their intent to communicate with each other via a proxy. Each then predicts its global mappings GA and GP as outlined in Section 3. If either NAT is cone then the respective end point can predict its global mapping prior to step #1 and use it for multiple TCP connections later. Each host must also open a RAW socket so that it can see a copy of the first SYN the OS TCP sends. Host A sends its GA and GP to B via the proxy and likewise receives B's global mappings. Upon receiving B's global mapping, A initiates a TCP handshake with it. The SYN sent as part of the 3-way handshake must have a sufficiently low TTL that the packet is dropped between N and M. The reason for this is that NAT M might close the hole created by Host B if it sees an unexpected SYN arrive from outside. The TTL is set using the IPHDRINCL option. The low TTL may cause an ICMP error to be generated in the network, though the ICMP packet itself doesn't hurt (or help) the connection setup (see Section 4.2.1 and 4.2.2).

Host A sends the contents of the TCP SYN (packet 3) that it heard via the RAW socket in a message to Host B (via the Proxy, packet 5). This message could certainly be a SIP message. Likewise, it receives the contents of Host B's TCP SYN via a message. Host A then constructs a message containing both its and Host B's SYN packets, and sends the message to its STUNT Server. From these two SYNs, the STUNT Server is able to construct the SYNACK that Host A and NAT N expect to see (i.e. containing the sequence number that Host B generated, and the ACK number that Host B would have produced had it received Host A's SYN. This SYNACK is transmitted from the STUNT Server to NAT N, spoofing the source address so that it appears to have come from Host B via NAT M (this can be done with a RAW socket).

NAT N finds the SYNACK it is expecting to see and translates the destination address to A's private address and routes it accordingly. The packet is then received by A's OS which acknowledges the SYNACK with an ACK therefore completing the 3-way handshake. The final ACK has the default TTL and makes its way to B,

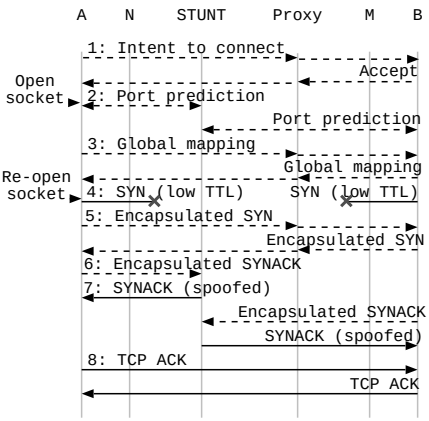

Figure 5: TCP Connection Setup

#	Source	Dest.	Contents
1	over Proxy		Intent to establish connection
2	STUNT port prediction		
3	over Proxy		A's global mapping A:P
4	a:p (A:P)[1]	B:Q	SYN, low TTL, Seq# SA
5	over Proxy		Encapsulated SYN, default TTL, source A:P
6	from A to STUNT		Encapsulated packet 7 (below)
7	B:Q (spoofed)	A:P (a:p)[1]	SYNACK, Seq#SB, ACK#SA+1, default TTL
8	a:p (A:P)[1]	B:Q (b:q)[2]	ACK, Seq#SA+1, Ack#SB+1, default TTL

Table 1: TCP Connection Setup

thus completing the three-way handshake.

4.2 Issues

In addition to the port prediction of the global mapping discussed earlier, the NAT TCP solution has its own fair share of issues. This section discusses some of these issues including NAT characteristics, host requirements and the spoofing requirement for the STUNT Server.

4.2.1 NAT Characteristics

The scheme would fail if the NAT releases the port mapping in response to an ICMP TTL Exceeded message that may be generated as a result of packet 4 in Figure 5. This, however, is unlikely since it would make the NAT box highly susceptible to a DoS attack where any host could silence outgoing connections by sending ICMP packets. In addition, ICMP errors are considered transient and the NAT box should let the host's stack determine whether or not to retry. Of the two NAT boxes we tested TCP establishment with, both ignored the ICMP unreachable.

A second issue relates to the separation between the two NATs N and M. If M is one hop away from N then it is not possible to determine a sufficiently low TTL, such that the packet will be discarded after it traverses N but before it reaches M. If M receives this SYN packet, it may respond by either silently dropping it, by returning an ICMP error or by returning a TCP RST packet. The first case would be ideal since neither A nor N sees any packets that cause them to deviate from the 3-way handshake. The second case is addressed in the previous paragraph. The third case, would result in N releasing the port mapping and A aborting the handshake causing the scheme to fail.

The scheme suggested above would also fail if the NAT unpredictably changes the Seq# of outbound TCP packets. Certain NATs, in particular BSD's pf module, can be configured to add a fixed offset to all outbound Seq# and subtract the same offset from inbound Ack#s. If this offset is picked at random for every connection then A will be unable to make the necessary changes to packet 5 in Figure 5. As a result, B's encapsulated SYNACK will have A's original Seq#, but packet 8 which traverses the NAT will have A's translated Seq# and will be considered by B's stack as a stray packet.

4.2.2 Other Requirements

The requirements for a STUNT end-point depends on the implementation. Our implementation is a user-space library that is source compatible with the BSD-style socket API. This imposes some limitations, for instance, the API depends on the kernel to construct network packets and send them on the physical medium. In order to reclaim a copy of packet 4 (Figure 5) created by the kernel, the API need to snoop for it using RAW sockets. While most OSs support RAW sockets at an application level, some OSs require super-user permissions to grant access to them. The second point of concern is that different OSs respond differently to ICMP errors received during connection setup, like the possible ICMP TTL exceeded message generated as a result of packet 4 having a low TTL. If the particular host OS considers this ICMP error non transient and aborts the connection setup, then the user-space API needs to block the ICMP message from propagating up the stack by temporarily inserting an appropriate rule in the OS's packet filter module. In our tests, we had to do this for the Linux hosts.

The requirement for a STUNT Server, which answers global mapping queries and generates the SYNACK (packet 7, Figure 5), is that it be able to *spoof* packets from arbitrary source IPs. This is because the encapsulate SYNACK sent by A to the STUNT Server which the server then inserts into the routing infrastructure has B's address as the source IP address. Sometimes ISP edge routers filter out packets with spoofed source addresses. Since STUNT Servers can be deployed in a controlled fashion, however, it should generally be possible to have the ISP turn off this filter at the STUNT Server's edge router (if indeed it is turned on in the first place).

A different approach to TCP connection setup through NATs is possible that does not require spoofing packets. In this scheme, hosts A and B discover their global mappings as before but only Host A sends the low TTL SYN packet (packet 4 in Figure 5). This establishes a mapping for A in NAT N. Host A then closes the socket it used to send the SYN and reopens another socket on the same local port listening for incoming connections. It signals B through the proxy when it is ready to receive connections. Upon receiving this signal, B, initiates a 3-way TCP handshake with the default TTL. The SYN sent by B is NAT'ed twice, first by M and then by N before it reaches A, who then replies with a SYNACK as expected. This scheme is less general than the one outlined in Section 4.1 since N may expect to see only an inbound SYNACK in response to packet 4. In addition, aborting the connection at A prior to opening a listen socket may result in a RST being sent out by A's stack which would close the mapping at N.

5. CONCLUSIONS AND THOUGHTS

This paper presents two *hacks* (important hacks, we believe, but hacks just the same) which expand the scope of connectivity establishment through NATs. This report is preliminary in that we don't

[1] After NAT by N

[2] After NAT by M

have a lot of field experience with these techniques. One goal of this paper is to encourage experimentation.

Because these techniques expand the scope of NAT connectivity to include TCP and Symmetric NATs, this paper additionally suggests that it is appropriate to ask whether the ICE/STUN/TURN style of NAT traversal should be expanded beyond its focus on media to include all data communications between *P2P* users, including easing the way towards transition to IPv6.

We certainly don't answer this question—indeed we haven't fully implemented the NUTSS architecture, much less experimented with it on a broad scale. We do believe, however, that NUTSS is an approach that deserves debate within the research community.

6. ACKNOWLEDGEMENTS

The authors would like to acknowledge Ian Yuan for contributing a key idea to the TCP traversal scheme. We would also like to thank Jeanna Matthews and Shashi Bhushan for their help.

7. REFERENCES

[1] Microsoft Corporation. UPnP – Universal Plug and Play Internet Gateway Device v1.01, Nov. 2001. Available online `http://www.upnp.org/standardizeddcps/documents/UPnP_IGD_1.0.zip`. 30 April 2004.

[2] J. Rosenberg. Internet draft: ICE – Interactive Connectivity Establishment, Feb. 2004. Available online `ftp://ftp.isi.edu/internet-drafts/draft-ietf-mmusic-ice-01.txt`. 30 April 2004.

[3] J. Rosenberg, R. Mahy, and C. Huitema. Internet draft: TURN – Traversal Using Relay NAT, Feb. 2004. Available online `ftp://ftp.isi.edu/internet-drafts/draft-rosenberg-midcom-turn-04.txt`. 30 April 2004.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. RFC 3261: SIP Session Initiation Protocol, June 2002. Available online `http://www.rfc-editor.org/rfc/rfc3261.txt`. 30 April 2004.

[5] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. RFC 3489: STUN – Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs), Mar. 2003. Available online `http://www.rfc-editor.org/rfc/rfc3489.txt`. 30 April 2004.

[6] Y. Takeda. Internet draft: Symmetric NAT Traversal using STUN, June 2003. Available online `http://community.roxen.com/developers/idocs/drafts/draft-takeda-symmetr%ic-nat-traversal-00.txt`. 10 May 2004.