NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs[*]

Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, Adrian Perrig
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{biggadike, ferullo, ggw, perrig}@cmu.edu

ABSTRACT

Firewalls and Network Address Translation (NAT) devices are becoming increasingly prevalent, and they pose a significant problem for connection establishment for peer-to-peer protocols. When properly configured, these *middle-boxes*[1] inhibit TCP connections solicited from outside the local network. This paper proposes novel mechanisms to create direct TCP connections between two hosts behind middle-boxes with minimal help from a third-party. We implement two of these solutions on common hardware within a common environment. We are able to create direct TCP connections between two hosts which are both located behind typical NATs designed for small networks. Once this connection is established, the applications can communicate with each other using a standard TCP implementation with no further external help.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management—*Network communication*; C.2.5 [**Computer Communications Networks**]: Local and Wide-Area Networks—*Internet*; C.2.2 [**Computer Communications Networks**]: Network Protocols—*Protocol Architecture*

General Terms

Algorithms, Design, Reliability

[*]This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Carnegie Mellon University, or the U.S. Government or any of its agencies.

[1]We refer to a NAT, firewall, or combination of the two as a *middle-box*.

Keywords

TCP Connectivity, Network Address Translation, Peer-to-peer over NAT, stateful firewall, consistent translation, unsolicited filtering, loose source routing, hole punching

1. INTRODUCTION

Network Address Translation was introduced as a means to continue the Internet's growth despite rapid depletion of IPv4's 32-bit address range. A secondary function of NAT is to hide the network topology from an external entity. Network Address Translation devices (NATs, which are also commonly called middle-boxes) separate an internal network from the broader Internet through the use of a separate address space in the internal network [9]. NATs dynamically translate between these address spaces for each network connection. In addition to the IP address translation, NATs must also allocate distinct external ports to distinct internal hosts. This allows multiple internal hosts to communicate with the same external host on the same source port. Another characteristic of NATs is that they only allow connections originating from within the internal network. NATs drop unsolicited connection attempts[2], since they have no way of knowing to which internal host the packet should be forwarded to.

Peer-to-Peer (P2P) networks have become increasingly popular. Despite the controversy generated by P2P file-sharing programs such as Napster and KaZaA, many useful and legitimate applications of P2P exist, for example, instant messaging, workspace sharing [3], and file sharing. The OpenHash Project [8] is another application of P2P networks. It provides a publicly available distributed hash table (DHT) upon which applications can be developed, including, many-to-many instant messaging, and a reliable CD label database.

Since each NAT only allows the establishment of outgoing connections, two peers that are both located behind a NAT cannot establish a direct TCP connection. Commercial NAT vendors have addressed this limitation by adding port forwarding characteristics to their devices. With port forwarding, administrators can specify the end hosts that should receive unsolicited connection attempts for each port. While this solution provides the needed support for many cases, it is limiting in cases where there are multiple machines providing the same service and when services require accepting connections on ports determined dynamically. Furthermore, if the end user doesn't have the required access or knowledge

[2]An unsolicited connection attempt is a TCP SYN packet or a UDP packet from the external network for which no mapping already exists at the NAT.

to configure the NAT, this solution is of no help.

P2P protocols have addressed this problem through a few common approaches. First, protocols have enabled techniques whereby peers that cannot be servers are sent messages that instruct them to initiate connections to peers that are requesting their data. This solution works in cases where only one peer is behind a NAT. The second common approach is to route traffic through proxies that each peer can connect to. While this solution enables a connection between two hosts behind NATs, it is inefficient since all traffic passes through the proxy. More related solutions are discussed in Section 3.

The aim of our work is to develop a solution that will enable a direct TCP connection between hosts that are located behind NATs. In particular, we have developed solutions for various environments depending on the port allocation characteristics of the NAT and the availability of loose source routing in the network. We have focused on TCP, rather than UDP connections because most P2P applications require reliable data transfer. Furthermore, UDP is a connectionless protocol, and it doesn't require a connection handshake or sequence numbers coordination. Solving P2P over UDP requires less complexity due to the simplicity of the protocol. Our solutions use a third party to provide the peers with the information needed to establish the direct connection. Depending on the environment we employ various techniques to enable connections to be established in a predictable and timely manner. Among these techniques are setting the packets' Time To Live (TTL) low, capturing and parsing outgoing packets to provide information to the third-party helper, and injecting manually-constructed packets into the network to determine ports selected by the NAT. Additionally, if port allocation is random, the birthday paradox is leveraged to reduce the necessary search space when determining the external port the NAT has allocated. This approach yields a search space that is approximately square root the size of a naive, port-scanning approach.

2. NAT CATEGORIZATION

Three specific features must be performed by a network device for it to be considered a NAT: transparent address assignment, transparent routing, and ICMP packet payload translation.

Address assignment refers to the creation of a mapping between non-routable internal and routable addresses at the start of a network session. For networks to continue to function correctly, NATs must perform this address assignment transparently with regard to both the source and destination. NATs can perform this assignment in either a static or dynamic manner. Static mappings must be predefined in a particular NAT such that an ⟨Internal IP address, Internal port⟩ tuple is mapped to a single ⟨External IP address, External port⟩ tuple for every session. Dynamic mappings, on the other hand, are defined on a per-session basis, and there is no guarantee that the same mapping will be created for future sessions.

A similar feature that NATs must implement is transparent routing. As mentioned, a NAT is a particular type of router that translates addresses in the packets it routes. This translation involves changing the IP addresses and ports in packets based on observed traffic flows. This must occur transparently with regard to the devices in the network to ensure compatibility with existing network stacks. A less obvious requirement of transparent routing is that NATs must ensure routing advertisements on the internal network do not reach the external network.

The final feature that must be implemented by NATs is performing the same translations that are done on regular packets to the payloads of ICMP error packets. When an error occurs in the network, such as when a packet's TTL expires, often an ICMP error packet is sent back to the sender. ICMP error packets embed the packet that generated the error inside their payload so the sender knows exactly which packet the error occurred on. If these error packets are generated from the external network, the addresses in the embedded packet will be the external addresses rather than the internal addresses. To make the embedded packet meaningful for the internal network, it is necessary for NATs to perform a reverse translation on the IP addresses within the ICMP error packet's payload.

While all NATs implement these three features, there are further categorizations of NATs based on their characteristics and the networking environments that they support. There are four categories of NATs: Two-way NATs, Twice NATs, Multi-homed NATs, and Traditional NATs. For further discussion on Two-way NATs, Twice NATs, and Multi-homed NATs, see [12]. By far the most common type of NAT is a Traditional NAT, and can be further divided into Basic NATs and Network Address Port Translation (NAPT) devices.

Basic NATs and NAPTs differ only in whether the number of external addresses the NAT can assign to internal addresses is larger or smaller than the number of internal addresses. A simple NAT is used in the situation where the number of external addresses is larger than or equal to the number of internal addresses. Since every internal IP address can be given a unique external IP address, these NATs do not perform port translation. NAPTs are used in the environment where there are less externally-allocatable addresses than internal addresses. The common situation is where multiple internal machines share one external IP address. In these situations the NAT needs to allocate ports in addition to IP addresses to eliminate the chance of network flow ambiguity. NATs and NAPTs are similar because neither accept incoming connections and both can assign addresses either statically or dynamically.

NAPTs are the most common type of Traditional NAT since they allow many internal machines to share a smaller set of addresses. Most commercial NAT devices designed for small networks are NAPTs. We have chosen to use NAPTs for our work due to their prevalence and since they conflict with common P2P protocols by not allowing incoming connections. Henceforth we will refer to NAPTs simply as NATs.

Our first step was to acquire commercial NATs to ensure that their characteristics are in accordance with how NAPTs are described in literature. We used the NatCheck program [1] to verify three common NATs: Netgear MR814, Linksys BEFSR41, and Linksys BEFW11S4. All three NATs have the same behavior: All three NATs provide *consistent translation* for both TCP and UDP traffic. Consistent translation means that the NATs directly map an ⟨Internal IP, Internal port⟩ pair to the same ⟨External IP, External port⟩ for the duration of the time that the ⟨Internal IP, Internal port⟩ pair is in use, regardless of the ⟨Destination IP, Destination port⟩ of the outgoing packets. Consistent translation is a distinct concept from static and dynamic address assignment because it refers to not only the the IP address but also the port used by the internal machine. RFC 3022 explicitly allows consistent translation [11]. None of the three NATs provide *loopback translation* for either TCP or UDP, which indicates if NATs correctly handle connections between two internal computers that only know each other's external addresses. This test is not relevant for our purposes since in our work we have assumed that the two peers are behind different NATs. Finally, all three NATs provide *unsolicited filtering* for TCP and UDP, which tests whether NATs prevent unsolicited messages to internal computers. Unsolicited filtering occurs in all NATs except Two-way NATs and is the major hindrance in enabling P2P communications between two devices behind NATs.

3. RELATED WORK

Independently three authors from Cornell worked on direct TCP connectivity through NAT and had similar results to ours. Their framework, termed NUTSS [4], provides for UDP and TCP connectivity between hosts behind NATs, but their TCP technique has a significant drawback. The protocol relies on spoofing packets in order to enable TCP connectivity, which limits its feasibility in real networks. Many Internet Service Providers perform ingress filtering to prevent spoofed packets from entering their networks, which would cause the authors' protocol to fail. Spoofing cannot be part of any solution to reliably connect hosts. To their credit, the authors do mention a technique that does not rely on spoofing. However, the technique relies upon TCP stack behavior, which is platform dependent. Our techniques described in this paper avoid spoofing while making realistic assumptions as well as provide for connectivity in environments beyond what is considered in NUTSS [4].

To address the difficulties that NATs create for many Internet protocols, a middle-box communication (MIDCOM) architecture is being developed [13]. MIDCOM is a protocol that would allow users behind a NAT or firewall to alter the middle-box's behavior to allow desired connections on demand. This system, while appropriate in some cases, is not always possible. In an environment where the user does not have control of the middle-box, P2P connections would still be disabled.

Many times users behind NATs or firewalls connect to a P2P network through a proxy server. A commercial proxy solution is provided by Hopster [6]. Hopster's proxy runs locally on the peer machine and tunnels application level traffic over https (port 443) to Hopster's own machines. However, since Hopster routes all traffic through their own machines, their approach is inefficient in comparison to ours, as is shown in the Section 5.

To enable direct P2P connections, UDP techniques have been developed. UDP Hole Punching [5] allows direct connections in a limited environment. The Simple Traversal of User Datagram Protocol through Network Address Translators (STUN) Protocol is an implementation of UDP Hole Punching that allows for NAT behavior to automatically be detected and UDP connections created, in limited cases [10].

In UDP hole-punching, the external port at each NAT is learned by a third party assisting the direct communication attempt. Both parties behind NATs send a UDP packet to the correct external port of the other peer. This creates the necessary NAT port mappings and establishes the connection. Once the mappings are created, direct UDP communication can occur. In situations where UDP hole-punching succeeds, one or more of our techniques presented in this paper will also work. Establishing a TCP connection between peers, rather than an UDP connection has advantages. First, UDP mappings at the NAT cannot be relied upon to exist for the duration of the connection and its setup. UDP is connectionless and there is no explicit end to UDP communication. NATs typically timeout UDP port mappings after period of inactivity. To maintain the UDP NAT mappings periodic traffic must be sent, even if it is null, to uphold the UDP communication. Second, many firewalls are configured to explicitly reject any incoming UDP packet. Finally, a pure TCP implementation of the connection is more intuitive and existing code can be more easily modified to leverage our techniques.

Recent work by Ford, et al. [2] has extended the hole-punching technique to enable TCP connections between hosts behind well-behaving NATs. The approach is similar to UDP hole-punching since a mapping is created at each host's NAT that enables a direct TCP connection to be created, either via asymmetric or simultaneous TCP opens. This work focuses on developing a technique that works on most NATs as well as defining the characteristics necessary for other NATs to become compatible with TCP hole-punching. Our work differs in that we have developed solutions for enabling direct TCP connections in the presence of various NAT behaviors, including those that are incompatible with TCP hole-punching.

Gnutella has a solution [14] for enabling TCP communication between two peers, but it only handles situations in which one peer is behind a NAT. This solution is referred to as Push Proxy and essentially creates multiple nodes that can push connection requests to the server that is behind that NAT. Servers behind NATs send messages to peers asking if they would be willing to be push proxies. When the server behind the NAT indicates that it has a file matching a query, it includes a list of those peers that have agreed to be push proxies. When the peer wants to download the file, it sends a Gnutella PUSH message to a push proxy, who then passes that message along to the server behind the NAT. The server behind the NAT then opens a connection to the peer who sent the PUSH message so the file can be transferred. While this approach does make it easier to establish a connection, it only handles the case where one peer is behind a NAT. Our solution addresses the more difficult problem of when both peers are behind a NAT.

Walfish et al. [15] suggest using an indirection service that will provide connectivity of two peers behind NATs by having both peers open a connection to the indirection server and having the server forward all traffic between them—in this paper we want to achieve such a connection without the need for such an indirection service.

4. PROBLEM STATEMENT AND ASSUMPTIONS

Consider the situation in which a peer and its buddy[3] know each other's IP addresses and are both behind NATs. If these hosts want to set up a direct TCP connection, a traditional TCP connection will not suffice. In a traditional TCP connection one party must be the initiator (creates the initial SYN packet) while the other listens for the initiation. In the situation where two peers are behind NATs, the listening peer will be prevented from seeing the SYN from the initiating peer because the SYNs will dropped at the listening peer's NAT. The SYNs are dropped because NATs and firewalls will typically not allow unsolicited packets from IP addresses on the Internet to enter their private network space. So, in order to establish a direct connection between two hosts behind NATs, each NAT must believe that the connection is solicited by the host within its internal network. We achieve this by making both parties be the initiators of the TCP connection–both peers create an initial SYN packet. Each NAT will believe the TCP connection attempt is solicited and will allow subsequent incoming data through to its private network. Note that although both peers send SYN packets, we are not using TCP simultaneous open.

In order to successfully create a TCP connection between the peers, each peer must know its buddy's externally-facing port prior to initiating the connection. This port is chosen by the NAT once a packet arrives from its internal network requesting to be routed to an IP address outside the internal network. As its bookkeeping, the NAT binds the internal IP address and port with the external port it chooses. We refer to this binding as the NAT's mapping. The NAT does not share this mapping with any host. Our techniques show how the NAT's mapping can be efficiently determined. Once both peers know the externally-facing port of their buddy, TCP con-

[3]We refer to a peer as "peer" and the other peer it is trying to connect to as its "buddy."

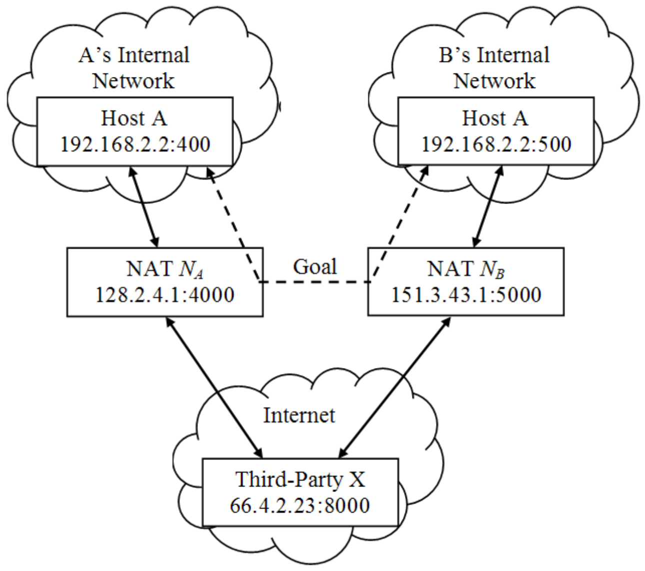

Figure 1: Environment for which we develop our techniques.

nections are initiated by both peers. The TCP sequence and acknowledgment numbers are integral components of synchronizing the TCP connection. Sequence numbers cannot be chosen, only observed. Our paper shows how the coordination of these parameters can be managed in order to successfully create a TCP connection regardless of the network environment.

Establishing direct TCP connections between hosts behind NATs is a difficult problem because external ports chosen by NATs are not directly accessible by hosts behind the NATs, and because successful TCP connections require coordination of sequence and acknowledgment numbers. There is no single solution that will work for all environments. The behavior of the NAT is dependent on its implementation, and the ability to predict ports is dependent on the amount of activity on the internal network.

We make two valid assumptions about the network which held for all the NATs we tested. The first assumption is we assume that hosts do not see ICMP TTL Exceeded packets from the external network. These packets, if received by either the peer or buddy, will terminate the TCP connection attempt. Many of our solutions rely on the ability to initiate a TCP connection by sending out an initial SYN packet with the TTL set too low. Once the SYN packet is dropped en route, ICMP TTL Exceeded packets are returned to the NAT. The NATs used for implementation did not forward ICMP TTL Exceeded packets to the private network when in response to TCP packets. Even if a NAT does forward ICMP TTL Exceeded packets to the private network, firewalls can be deployed at the host to block such packets. Our second assumption is that a NAT will not nullify the mapping created if it sees ICMP TTL Exceeded packets. Alternatively, we could leave the TTL value with the default value, and rely on the destination NAT not generating TCP RST packets. In practice this is a viable option since many NATs don't generate RST packets to help defend against port scanning.

5. TECHNIQUES

Using Figure 1 as the model environment, our goal is to enable a direct TCP connection between *A* and *B*, residing behind NATs N_A and N_B.

We have developed various techniques for enabling this TCP connection depending on the exact properties of the NATs and the network properties. If we consider this information as the ordered triplet
⟨N_A port allocation, N_B port allocation, source routing availability⟩, we consider the following cases:
Case 1: ⟨predictable, predictable, LSR⟩
Case 2: ⟨predictable, predictable, no LSR⟩
Case 3: ⟨random, predictable, LSR⟩
Case 4: ⟨random, predictable, no LSR⟩
Case 5: ⟨random, random, LSR⟩
Case 6: ⟨random, random, no LSR⟩

Note that ⟨random, predictable, X⟩ is equivalent to ⟨predictable, random, X⟩.

5.1 Pre-connection Diagnostics

In order for the helper, *X*, and the two peers, *A* and *B*, to determine which of the following cases their connection attempt falls into, *X* must first do some diagnosis of each peer.

In order to use Cases 1, 3, and 5, the parties must determine if loose source routing is available on the segments between *A* and *X* and *B* and *X*. Loose source route (LSR) is an IP option that allows the creator of an IP packet to specify a list of mandatory IP addresses to be used in the packet's route. The result of this option is that each IP address in the route list will receive the packet in the order specified in the route list. The loose source route option introduces a security risk, because an attacker can eavesdrop on a session by being in the route list. Due to this potential risk, many routers drop packets containing the loose source route option.

To determine if LSR is available from *A* to *B* through *X*, *A* can simply try to connect to *B*, loose source routing the packet through *X*. If *X* receives this packet, then LSR is available from *A* to *B* for the first half of the journey to *X*. If *X* does not receive any packets after a specified timeout then it can be assumed LSR is not available. Because *X* can only tell if the first half of the journey from *A* to *B* allows LSR from this method, it must check to see that it also gets LSR packets from *B*. If it does then *X* can conclude that LSR is available from *A* to *B* through *X*, in any other case it must assume LSR is not an option.

To determine if N_A will randomly or predictably allocate ports, *A* can open two TCP connections to *X* from sequential ports. If the ports for these connections observed by *X* are sequential then *X* can conclude that N_A allocates ports sequentially, and thus predictably. When connecting to *B*, *A* should use the next port in sequence, to ensure that N_A will continue to map ports in the manner *X* can predict.

If N_A does not assign ports sequentially, it is still possible to predict *A*'s ports, if N_A implements consistent translation. *A* must first open a legitimate connection to *X* from internal port p_A. N_A will assign this connection a random port. *X* very clearly can see the port chosen by N_A since the packet was sent to it. *A* can open a second connection to *X* sent to a different port on *X* from the same internal port , and *X* can see if these two connections contain the same external port. If they do, N_A implements consistent translation. *A* must now use the internal p_A to connect to *B*, so that *X* can tell *B* the external port chosen by N_A. It is important that *A* and *X* maintain this connection until *A* is connected to *B* so that N_A will not alter the port mapping.

If, after trying both methods of port prediction *X* is unable to reliably predict ports assigned by N_A, then *X* must assume N_A assigns ports randomly.

While *X* completes this diagnosis of *A* it can simultaneously do the same with *B*. Once *X* has all the required information, the connection protocol can begin. The specific case to implement is determined by the information gathered from this diagnosis.

5.2 Sequence and Acknowledgment Number Coordination

Every participant in a TCP connection maintains two variables, a sequence number and an acknowledgment number. At any given time, the sequence number at any host is the sequence number of the last packet sent. On the other hand, at any given time, the acknowledgment number at a host is the sequence number of the next expected packet. [7].

Stepping through the three-way handshake, the initial sequence and acknowledgment numbers are established as follows:

1. After client sends SYN packet,
   Client's seq#: P, ack#: N/A
   Server's seq#: N/A, ack#: N/A

2. After server receives SYN packet and sends SYN+ACK,
   Client's seq#: P, ack#: N/A
   Server's seq#: Q, ack#: P+1

3. After client receives SYN+ACK and sends ACK,
   Client's seq#: P, ack#: Q+1
   Server's seq#: Q, ack#: P+1

4. After server receives ACK,
   Client's seq#: P, ack#: Q+1
   Server's seq#: Q, ack#: P+1

The state at the end of the three-way handshake must be replicated by our solutions even though both peers assume client roles. At the end of each solution, each peer's acknowledgment number must be one greater than their buddy's sequence number. Our solutions achieve this coordination.

5.3 Low TTL Value Determination

Some of our solutions depend on setting a TCP packet's time to live (TTL) value such that the packet will leave the peer's internal network, but not reach the buddy's NAT. For different networks this value will be different, and as such it must be able to be dynamically determined.

To determine how far away the buddy is, a peer can follow the typical traceroute method. That is, send SYN packets with increasing TTL values, starting at 1. Each of these packets will cause ICMP TTL Exceeded messages to be sent back to the peers when the TTL expires. By analyzing when ICMP TTL Exceeded messages are returned the peer can determine a safe value to use for the low TTL value in the connection.

Most NATs will not forward ICMP TTL Exceeded messages back to an internal host, so a peer can conclude that a TTL value caused a packet to leave the internal network as soon as an ICMP TTL Exceeded message is not returned.

Likewise, in situations where the NAT does forward ICMP TTL Exceeded messages the peer must base the discovered safe TTL value by analyzing the buddy NAT's messages. If the buddy's NAT generates a RST packet then the peer can use a TTL value one less than the value that cause the RST packet. If the peer never gets a RST packet but begins to stop receiving ICMP TTL Exceeded messages then it can conclude the buddy's NAT drops unsolicited messages without reply, which is safe behavior. In fact, this case is the same as when the peer's NAT does not forward ICMP TTL Exceeded messages.

This safe TTL value determination does not require any participation by any party other than the peer. Thus, it can be done at any point before the safe low TTL value must be used in the connection.

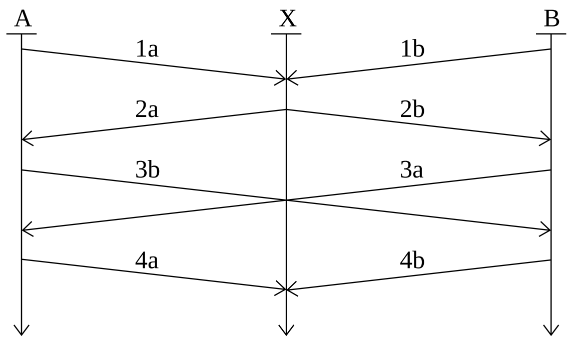

Figure 2: Case 1

5.4 Case 1: ⟨predictable, predictable, LSR⟩

We use the notation N_A:4000 → N_B:5000, *options/payload* to denote the contents of the packet while it is in transit on the Internet from NAT N_A to NAT N_B. This notation signifies that the packet has a source address of N_A's IP address, source port of 4000, destination address of N_B's IP address, and destination port of 5000. Additionally, any important options or payload values appear after the destination port. The options include *LSR:X*, *SYN:P*, *ACK:Q*, and *SYN+ACK:R,S*. *LSR:X* denotes that the packet will be loose-source-routed through *X*. *SYN:P*, *ACK:Q*, denote the type of TCP packet followed by the sequence or acknowledgment number. *SYN+ACK:P, Q+1* denotes that the packet is a TCP SYN+ACK packet with sequence number *P* and acknowledgment number $Q+1$. Initially we develop Case 1, ⟨predictable, predictable, LSR⟩, using the sequence of events found in Figure 2.

1. *A* and *B* send a SYN to each other loose source routed through Helper *X*

   (a) N_A:4000 → N_B:5000, LSR:*X*, SYN:P

   (b) N_B:5000 → N_A:4000 , LSR:*X*, SYN:Q

   These SYN packets are generated by TCP `connect()` calls. These SYNs create the desired mappings at NAT N_A and N_B. The mapping at N_A will allow subsequent communication from N_B:5000 to be relayed to A and vise versa.

2. *X* buffers both packets and sends *A* and *B* the ISNs each other used

   (a) *X*:1234 → N_A:3999 , B just used ISN Q

   (b) *X*:1235 → N_B:4999 , A just used ISN P

   Each peer needs their buddy's ISN, so they can fabricate a legitimate SYN+ACK packet.

3. *A* and *B* send SYN+ACKs to each other

   (a) N_B:5000 → N_A:4000 , LSR:*X*,
   SYN+ACK:Q, P+1

   (b) N_A:4000 → N_B:5000 , LSR:*X*,
   SYN+ACK:P, Q+1

   These SYN+ACKs are generated from a separate thread running on each peer. By reusing their original sequence numbers, *P* and *Q*, as the sequence numbers in the SYN+ACKs, *A* and *B* will ensure the final state of the sequence and acknowledgment numbers replicates that of a real TCP connection as discussed in Section 5.2.

4. *A* and *B* send ACKs to each other

   (a) N_A:4000 → N_B:5000, LSR:*X*, ACK:Q+1

   (b) N_B:5000 → N_A:4000, LSR:*X*, ACK:P+1

   The TCP stack will do this step for us automatically once the fake SYN+ACKs are received.

5. *X* drops the two ACKs as they arrive, because no one is expecting to receive an ACK.

Figure 2 assumes that *A* and *B* are aware of which port their buddy will be working on; this assumption is reasonable since the peer and buddy must have known about each other ahead of time. Prior to step 1, *X* must perform port prediction on both *A* and *B* so that *X* can predict the ports that will be chosen by the NAT devices. *A* must know N_B is working on port 5000, while *B* must know that N_A is working on port 4000. For simplicity we assume *X* itself is not behind a NAT, but the only condition is that *X* must have prior direct connections with both *A* and *B*.

An alternative solution to Case 1 exists. *X* could spoof the needed SYN+ACK packets in steps 2 and 3 rather than send information to *A* and *B* so they can fabricate the SYN+ACKs themselves. We choose the presented method because if *X* spoofs the SYN+ACKs, they may be dropped by a router rather than forwarded. Additionally, moving SYN+ACK forging from *X* to *A* and *B* removes the need for *X* to run with superuser privileges. *A* and *B* must already run with superuser privileges for other purposes.

Since steps 2 through 5 are so repeatedly used in our techniques, we will denote *Function Case1(integer extPortA, integer extPortB)* as the execution of steps 2 through 5, substituting the parameters extPortA and extPortB for the external ports 4000 and 5000 respectively.

5.5 Case 2: ⟨predictable, predictable, no LSR⟩

Case 1 relied on the availability of loose source routing. Most routers currently are configured to prevent loose source routing, and will typically drop packets requesting the service. As such, there is a high probability that techniques relying on loose source routing will not be successful in practice. If loose source routing is not available, the sequence number of the SYNs can be communicated to *X* using an out-of-band channel (their pre-established TCP connection with *X*) instead of having *X* physically see the packets. Note that in step 2 of Figure 2, *X* knows the TCP sequence numbers P and Q because *X* actually received the two SYN packets. Without loose source routing this is not the case.

To initiate the connection, each end host sends an initial SYN packet to their buddy that they know will not reach its destination. They then sniff the packet off the network, note the sequence number, and report this information to *X*. *X* needs the TCP sequence number from these packets so that it will be able to relay the information back to *A* and *B* so that they can generate SYN+ACKs. Two ways of sending packets that will not reach their destinations are addressed.

The simplest solution is for each peer to send a SYN to their buddy without regard. Properly configured NATs and firewalls at the receiving end will not forward this packet to the internal host because no mapping exists. Some NATs and some firewalls will send TCP Reset packets (RSTs) to the source of an unsolicited SYN packet. If a NAT does generate RST packets, *A* and *B* cannot simply send a SYN to each other like step 1 in Figure 2 would suggest, because upon receipt of this RST, N_A and N_B would terminate the hole created. If the NATs do not generate RST packets, the open TCP connections will not be abruptly terminated.

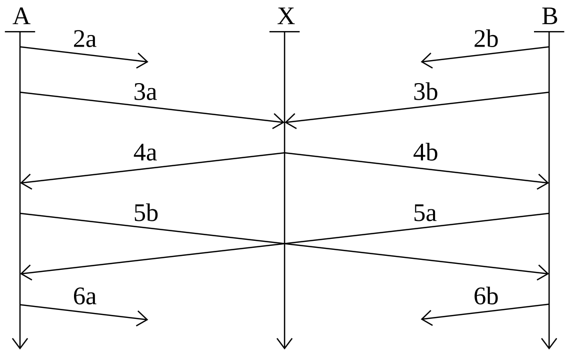

Figure 3: Case 2

Another way to ensure the SYN packet will not reach its destination network is to send SYN packets with TTL values less than the path length to the buddy's NAT. The packets will definitely be dropped on the way to the destination, and a TCP RST packet will not be seen by either sender. Rather, an ICMP Time Exceeded packet will be seen and is a problem because ICMP Time Exceeded packets terminate a TCP connection abruptly. However, if the user can configure their local firewall to drop ICMP packets or if the NAT doesn't forward these ICMP messages to its internal network, the TCP connection attempt will not abruptly close.

A solution cannot involve simply spoofing the source address of the SYN packet so that the sender does not receive either the ICMP packet or the RST packet. Doing this would create an invalid mapping at the middle-box. Upon seeing a SYN packet, the middle-box will create a mapping from internal IP address and port to external IP address and port. However, since a spoofed SYN packet has an incorrect source IP address, the mapping will not correspond to the correct host in the internal network. Additionally, a solution cannot involve setting the TTL so low that even the middle-box does not see the SYN packet, because doing this would not create the mapping that we need to allow subsequent communication into the network from the outside.

Assuming a ⟨predictable, predictable, no LSR⟩ environment, the connection as we now have described is presented in Figure 3.

1. *X* does port prediction as described in Section 5.1. *X* predicts N_A's next port to be 4000 and N_B's next port to be 5000. *X* informs *A* and *B* of this via their existing connections.

2. *A* and *B* send a SYN to each other that they know will be either dropped by the NAT at the other side or dropped due to a TTL expiration

   (a) N_A:4000 → N_B:5000, SYN:P

   (b) N_B:5000 → N_A:4000, SYN:Q

   This is the point at which the actual TCP `connect()` call is made at each peer. The SYN packets are generated by the TCP stacks. This creates the mappings at the NATs that will allow subsequent communication from the buddy's IP address and port to reach the peer.

3. *A* and *B* send *X* the ISNs (P and Q) they observed

   (a) N_A:3999 → *X*:1234, I just used ISN P

   (b) N_B:4999 → *X*:1235, I just used ISN Q

Each peer will need its buddy's ISN so they can fabricate legitimate SYN+ACKs to their buddy.

4. X sends A and B the ISNs each other observed
   (a) X:1234 $\rightarrow$ N$_A$:3999, B just used ISN Q
   (b) X:1235 $\rightarrow$ N$_B$:4999, A just used ISN P

5. A and B send SYN+ACKs to each other
   (a) N$_B$:5000 $\rightarrow$ N$_A$:4000, SYN+ACK:Q, P+1
   (b) N$_A$:4000 $\rightarrow$ N$_B$:5000, SYN+ACK:P, Q+1

   This is the second part of the three-way handshake. Again, by reusing their original sequence numbers, P and Q, as the sequence numbers in the SYN+ACKs, A and B will ensure the final state of the sequence and acknowledgment numbers replicates that of a real TCP connection as discussed in Section 5.2.

6. A and B send ACKs to each other that they know will be either dropped by the NAT at the other side or dropped due to a TTL expiration
   (a) N$_A$:4000 $\rightarrow$ N$_B$:5000, ACK:Q+1
   (b) N$_B$:5000 $\rightarrow$ N$_A$:4000, ACK:P+1

   The TCP stack will send these ACKs automatically for us, finishing the three-way handshake. We do not want the ACKs to reach their destinations because no one is waiting for an ACK.

Much like in Case 1, as an alternate to steps 4 and 5, X could spoof the needed SYN+ACK messages to A and B. However, we have chosen the presented method for the same reasons as in Case 1.

Since steps 2 through 6 are so repeatedly used in our techniques, we will denote *Function Case2(integer extPortA, integer extPortB)* as the execution of steps 2 through 6, substituting the parameters extPortA and extPortB for the external ports 4000 and 5000 respectively.

5.6 Case 3: ⟨random, predictable, LSR⟩

Case 3 ⟨random, predictable, LSR⟩ is similar to Case 1 as described in Figure 2. However, X will not be able to predict one of the two NAT's ports, say N$_A$. A will have to send its SYN packet first to allow X to view which port N$_A$ chose. X will then have to report this information to B so that B can send its SYN out to the proper destination IP address and port. This modification of Case 1 is depicted in Figure 4 and is explained below.

1. X does port prediction as described in Section 5.1. X cannot predict N$_A$'s next port, but can predict N$_B$'s next port to be 5000 and informs A and B of this via their existing connections.

2. A and B synchronize via X
   (a) N$_A$:m $\rightarrow$ N$_B$:5000, LSR:X, SYN:P
   (b) X lets B know that N$_A$ is working on port m
   (c) N$_B$:5000 $\rightarrow$ N$_A$:m, LSR:X, SYN:Q

   These SYN packets are generated by TCP `connect()` calls. These SYNs create the desired mappings at NAT N$_A$ and N$_B$.

3. *Call Case1(m,* 5000*)*

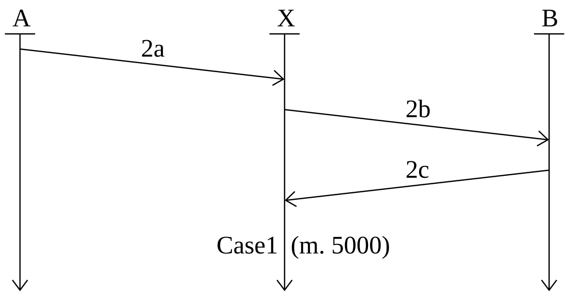

Figure 4: Case 3

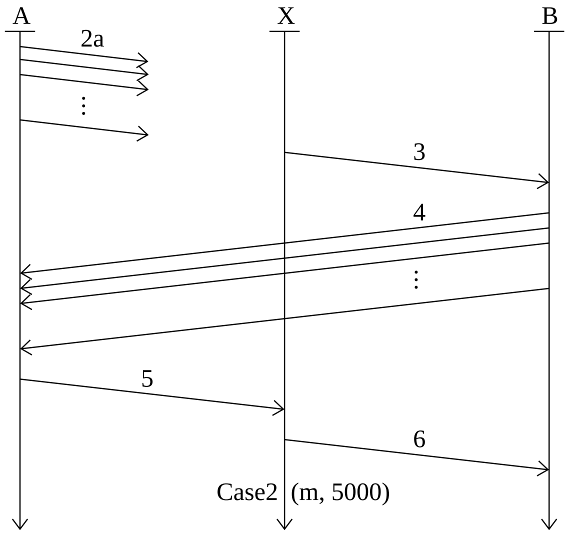

Figure 5: Case 4

5.7 Case 4: ⟨random, predictable, no LSR⟩

The environment in Case 4 is ⟨random, predictable, no LSR⟩. We have developed a solution for this environment that depends on the random NAT not rejecting a TCP packet with an invalid ACK or checksum field corresponding to a connection previously initiated by the host behind the NAT. The solution is presented in Figure 5 and explained below.

1. X does port prediction as described in Section 5.1. X cannot predict N$_A$'s next port, but can predict N$_B$'s next port to be 5000 and informs A and B of this via their existing connections.

2. A sends T SYNs to B that will either be dropped by the NAT at the other side or dropped due to a TTL expiration

```
$i = 0$
While $i < T$
        N$_A$:$rand$ $\rightarrow$ N$_B$:5000, SYN:$anything$
        $i = i + 1$
End While
```

   This creates T mappings at NAT N$_A$, one of which B will eventually guess with a SYN+ACK.

3. X instructs B to begin sending SYN+ACKs to N$_A$

4. B sends many SYN+ACKs to N_A until one reaches A

```
$i = 1024$
While $A$ has not reported success
        $N_B$:5000 $\rightarrow$ $N_A$:$i$,
        SYN+ACK:,anything,anything, Payload:$i$
        $i = i + 1$
End While
```

5. A reports the payload of the packet that made it through the NAT.
   N_A:3999 $\rightarrow$ X:1234, port m worked
   A will see this invalid SYN+ACK packet by listening on the wire for any SYN+ACK packet from N_B.

6. X tells B to connect with A on port m
   B now knows where to send its SYN.

7. *Call Case2(m,* 5000*)*

The T SYNs sent by A in step 2 are independent of any TCP `connect()` call. They are merely packets generated using the libnet libraries, creating T mappings at NAT_A. On the other hand, the SYNs generated in step 2 of the *Case2* call are due to TCP `connect()` calls by A and B. This solution to a Case 4 environment depends on the behavior of the NAT that allocates ports randomly. The solution relies on the middle-box not denying TCP packets with incorrect fields such as the sequence number or checksum.

The value T can be chosen such that B has a 95% chance of guessing a correct external port after generating T SYN+ACKs with random destination port numbers. In essence, NAT_A randomly chooses T numbers (its external port numbers), then B must keep guessing numbers until one chosen by B is in the set chosen by NAT_A. We can use a probabilistic analysis to construct an efficient scenario in which a minimal amount of work is imposed on both A and B. Let Pr_G be the probability that B guesses at least one correct port in T trials, and let $Pr_{\neg G}$ be the probability that B does not choose a correct port in T trials. Given that NAT_A has already chosen T distinct port numbers within the range $[1025, 65535]$, if B chooses T distinct ports, the probability of B not choosing a number from the set chosen by NAT_A is

$$Pr_{\neg G} = \frac{n-T}{n} \cdot \frac{n-1-T}{n-1} \cdot \frac{n-2-T}{n-2} \cdot \ldots \cdot \frac{n-(T-1)-T}{n-(T-1)}$$

where n is the number of possible port choices ($n = 65535 - 1024 = 64511$).

$$Pr_{\neg G} = \prod_{i=0}^{T-1} \frac{n-i-T}{n-i}$$

Conversely, the probability of guessing at least one port correctly in T trials is

$$Pr_G = 1 - Pr_{\neg G}$$

As stated before, T should be chosen such that

$$Pr_G > 95\%$$

$$1 - \prod_{i=0}^{T-1} \frac{n-i-T}{n-i} > 95\%$$

Solving this product for T yields $T = 439$.

$$1 - \prod_{i=0}^{439-1} \frac{64511-i-439}{64511-i} = 0.9506 > 95\%$$

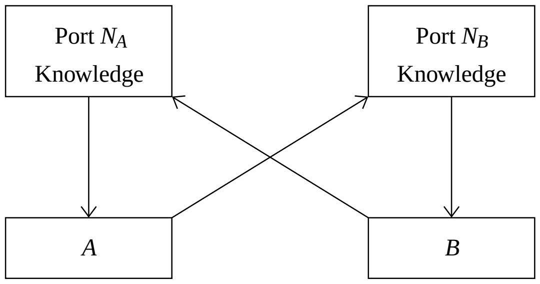

Figure 6: Resource Diagram Deadlock

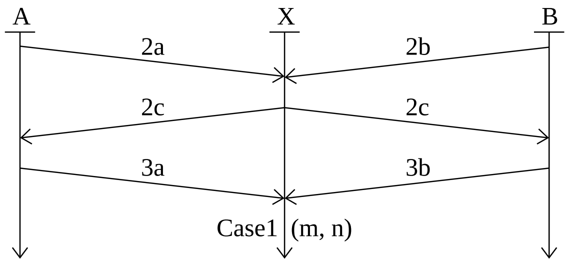

Figure 7: Case 5

This result says that if A sends out 439 SYN packets, which are mapped to distinct, random, external ports at NAT_A, and B sends many SYN+ACK packets with distinct, random, destination ports, B has greater than a 95% chance of correctly guessing one of the 439 mapped external ports before it sends the 440th SYN+ACK.

The reason for only sending T SYN packets is to minimize two resources, the first being network bandwidth use, and the second being the number of mappings created at the NAT.

5.8 Case 5:⟨random, random, LSR⟩

In Case 5 the environment is ⟨random, random, LSR⟩. In order to allow X to synchronize both A and B, B must know the port chosen by N_A prior to sending out its SYN. In order to determine what port N_A will choose, X will have to see A's SYN packet. A's SYN packet cannot be sent until X determines which port N_B chooses. This deadlock is illustrated in Figure 6. A is holding the “Port N_A Knowledge” resource by not sending out a SYN, effectively preventing X from learning the port chosen by N_A. Likewise B is holding the “Port N_B Knowledge” resource. Each needs the other's port before they can release the resource held. Our solution prevents this deadlock by having A and B send two SYN packets loose source routed through X, not connected to a TCP `connect()` call. These two SYN packets create the mappings needed at each NAT and allows X to gain the two resources, and coordinate the connection in a similar manner to Case 1 or 2. Our solution for Case 5 is shown in Figure 7 and is explained below.

1. X does port prediction as described in Section 5.1. X cannot predict N_A or N_B's next ports and informs A and B of this via their existing connections.

2. A and B each send a SYN loose source routed through X

   (a) N_A:m $\rightarrow$ N_B:*anything* SYN:*anything*, LSR:X
   (b) N_B:n $\rightarrow$ N_A:*anything* SYN:*anything*, LSR:X
   (c) X reports m to B and n to A.

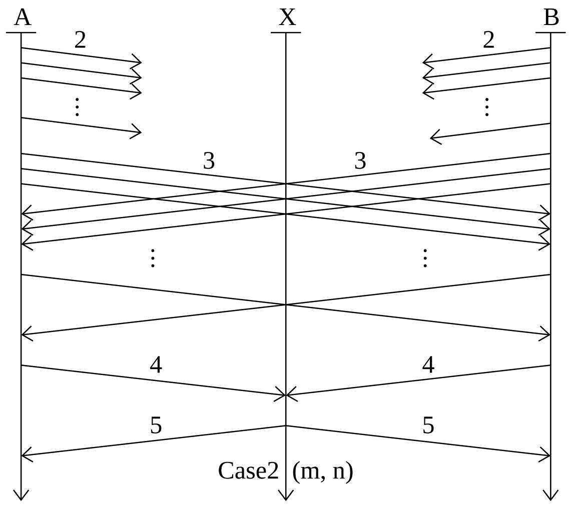

Figure 8: Case 6

These SYNs will create the necessary mappings at each NAT.

3. *A* and *B* send a SYN to each other loose source routed through *X*

   (a) N_A:*m* → N_B:*n*, LSR:*X*, SYN:P

   (b) N_B:*n* → N_A:*m*, LSR:*X*, SYN:Q

   Because of *Consistent Translation*, even though the destination ports are different from the previous step, the NAT will still utilize use the same mapping (and thus the same external port) for these packets.

4. *Call Case1(m, n)*

Note that the SYNs sent in step 2 are not connected to any TCP `connect()` call, rather the SYNs sent out in step 3 are due to a TCP `connect()` call. Also the SYN+ACKs sent in step 3 of the *Case1* call are not tied to a TCP `accept()` subroutine.

5.9 Case 6:⟨random, random, no LSR⟩

In Case 6 the environment is ⟨random, random, no LSR⟩. Looking back at the resource diagram deadlock in figure 6, neither *A* nor *B* holds these port knowledge resources since packets cannot be loose source routed. The solution to this case is pictured in Figure 8 and explained below.

1. *X* does port prediction as described in Section 5.1. *X* cannot predict N_A or N_B's next ports and informs *A* and *B* of this via their existing connections.

2. *A* sends *T* SYNs to *B* and *B* sends *T* SYNs to *A* that will either be dropped by the NAT at the other side or dropped due to a TTL expiration

```
$i = 0$
While $i < T$
        $N_A$:rand → $N_B$:rand, SYN:anything
        $N_B$:rand → $N_A$:rand, SYN:anything
        $i = i + 1$
End While
```

   These SYNs create *T* mappings at both NATs.

3. *B* and *A* send many SYN+ACKs to their buddy's NAT until one reaches their buddy.

```
$i = 1024$
While A has not reported success
        $N_B$:rand → $N_A$:i,
        SYN+ACK:,anything,anything, Payload:i
        $N_A$:rand → $N_B$:i,
        SYN+ACK:,anything,anything, Payload:i
        $i = i + 1$
End While
```

4. *A* and *B* report the payload of the packet that made it through the NAT.
   N_A:3999 → *X*:1234, port *m* worked
   N_B:4999 → *X*:1235, port *n* worked

5. *X* tells *B* to connect with *A* on port *m* and tells *A* to connect with *B* on port *n*.
   A and *B* now know the external port of their buddy.

6. *Call Case2(m, n)*

Case 6 is significantly more difficult than Case 4, because each peer must correctly guess one entire mapping ⟨source port, destination port⟩ at the opposite NAT. In Case 4, the peer behind the non-random NAT only had to guess the destination port correctly. The source port was fixed since one of the NATs was predictable. The search space for Case 6 is the square of the search space for Case 4 - instead of 64,511 possibilities, there are 4,161,669,121 combinations to be guessed from.

6. IMPLEMENTATION

We have implemented Cases 2 and 4 in C on Linux workstations and have made use of the libnet and libpcap libraries. Cases 1, 3, 5, and 6 were not implemented.

Both the helper and peer connection libraries consist of a single function. The helper routine, `natblaster_server()`, only needs to be provided the port number the helper should listen on. The peer connection routine, `natblaster_connect()`, must be provided seven parameters: (1) the helper's IP address and (2) port number, (3) the local peer's external IP address, (4) internal IP address, and (5) port, (6) the buddy's external IP address, and (7) port. The local peer and buddy ports are only needed by the helper to help create a unique identifier for the connection attempt. The ⟨Local External IP, Buddy Internal IP, Buddy Internal Port⟩ triple is used as the unique identifier at the helper. The libraries will try to provide a socket on the specified ports, however, the returned socket is not guaranteed to be over the ports specified. Assuming the `natblaster_connect()` works, the library returns a valid socket handle.

To test our implementation we ran two peers, each located behind different commercial NATs on separate networks. The third-party program was run on a third computer not located behind a NAT. We tested our code on the Internet rather than a local network to make our tests more realistic.

In order to create packets that will never reach the buddy and return no error message, we set the TTL value too low to reach the buddy. Setting the TTL too low was accomplished by calling the `setsockopt()` system call with the `IP_TTL` option. The option also requires a TTL value. This value must be less than the number of hops to the buddy, but greater than the number of hops to the outermost NAT. The socket option must not be persistent for the entire life of the socket. For instance, after the three-way handshake has succeeded, `setsockopt()` should be called again to raise the TTL so that subsequent data will make it to the peer. Relying on a low TTL only works if an ICMP TTL Exceeded packet is not seen by the peer's TCP stack, because it could cause the socket to fail at the peer. The NATs we tested do not forward ICMP TTL Exceeded packets to the internal network. The alternative would have been to send normal packets and hope the buddy's NAT will silently drop them, however, some NATs may send RST packets in response to unsolicited data. This behavior is implementation specific. We did not make use of the TTL determination technique presented in Section 5.3; instead we chose low and normal TTL values that we knew were appropriate.

For the pre-connection diagnostics we implemented the sequential port allocation determination method, but did not implement the consistent translation determination. Our implementation does not make use of consistent translation.

Both our Case 2 and Case 4 implementations are successful and able to open direct TCP connections. Case 2 reliably opens connections, and Case 4 is successful with a high probability (the probability of success is determined by the number of SYNs and SYN+ACKs sent, as discussed earlier).

We did not implement Case 6 due to the reasons given at the end of Section 5.9. We did not implement Cases 1, 3, and 5 because LSR is not typically available on the Internet and we believe it would have a low probability of succeeding in practice.

As previously mentioned, we used the standard Berkeley network implementation, augmenting it with additional system calls when necessary. For instance, when we send a SYN packet but need to know its sequence number, the packet is sent using a standard `connect()` call, after first having started a thread to watch the wire for the sent SYN packet. This thread can then report the sequence number used.

In both Case 2 and Case 4 it is necessary to run the peers with root privileges, as they are required by the libpcap and libnet libraries. The helper can run with normal user privileges since no spoofing nor sniffing is necessary.

7. CONCLUSION

We have shown how to create direct TCP connections between hosts behind NATs in typical environments with typical hardware. These solutions do not involve changing the TCP/IP stack in any way, but rather leverage the cooperation of these parties to establish a connection. Our solutions can be applied to many applications from Peer-to-Peer networks to Instant Messaging. Existing solutions for this connectivity problem include proxies, which are not an efficient use of network resources and do not scale.

Our techniques outlined in this paper are meaningful regardless of whether or not NATs are integral network components. The cases which contain predictable NATs can also be applied to hosts behind stateful firewalls. Similar to NATs, stateful firewalls are capable of only allowing TCP connection instantiation from within the network they protect. Our solutions enable both parties to instantiate the TCP connection, which these stateful firewalls will allow. Some of our solutions would not be advisable in situations where IDSes are deployed due to the port scanning equivalent technique used in Cases 4 and 6, which will most likely set off such network monitoring devices. However, our solutions are general enough to work in most environments, and even those environments which may not yet exist: NATs that do random port allocation.

8. REFERENCES

[1] Bryan Ford. NatCheck: Hosted by the MIDCOM-P2P project on SourceForge. `http://midcom-p2p.sourceforge.net`.

[2] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[3] Groove Networks. `http://groove.net`.

[4] Saikat Guha, Yutaka Takeday, and Paul Francis. NUTSS: A SIP-based approach to UDP and TCP Network Connectivity. In *SIGCOMM 2004 Workshops*, Aug 2004.

[5] M. Holdrege and P. Srisuresh. Protocol Complications with the IP Network Address Translator. RFC 3027, Internet Engineering Task Force, January 2001.

[6] Hopster: Bypass Firewall Bypass Proxy Software. `http://www.hopster.com`.

[7] Information Sciences Institute. Transmission Control Protocol (TCP). RFC 793, Internet Engineering Task Force, September 1981.

[8] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT Service. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, Feb 2004.

[9] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918, Internet Engineering Task Force, February 1996.

[10] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP). RFC 3489, Internet Engineering Task Force, September 2003.

[11] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, Internet Engineering Task Force, January 2001.

[12] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, Internet Engineering Task Force, August 1999.

[13] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework. RFC 3303, Internet Engineering Task Force, August 2002.

[14] Jason Thomas, Andrew Mickish, and Susheel Daswani. Push Proxy: Protocol Document 0.6, June 2003.

[15] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes No Longer Considered Harmful. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, December 2004.