Characterization and Measurement of TCP Traversal through NATs and Firewalls *

Saikat Guha Paul Francis
Cornell University
Ithaca, NY 14853
{*saikat, francis*}*@cs.cornell.edu*

Abstract

In recent years, the standards community has developed techniques for traversing NAT/firewall boxes with UDP (that is, establishing UDP flows between hosts behind NATs). Because of the asymmetric nature of TCP connection establishment, however, NAT traversal of TCP is more difficult. Researchers have recently proposed a variety of promising approaches for TCP NAT traversal. The success of these approaches, however, depend on how NAT boxes respond to various sequences of TCP (and ICMP) packets. This paper presents the first broad study of NAT behavior for a comprehensive set of TCP NAT traversal techniques over a wide range of commercial NAT products. We developed a publicly available software test suite that measures the NAT's responses both to a variety of isolated probes and to complete TCP connection establishments. We test sixteen NAT products in the lab, and 93 home NATs in the wild. Using these results, as well as market data for NAT products, we estimate the likelihood of successful NAT traversal for home networks. The insights gained from this paper can be used to guide both design of TCP NAT traversal protocols and the standardization of NAT/firewall behavior, including the IPv4-IPv6 translating NATs critical for IPv6 transition.

1 Introduction

Network address and port translators (NATs) and firewalls break the IP connectivity model by preventing hosts outside the *protected network*[1] from initiating a connection with a host inside the protected network. If both endpoints are protected by their respective NAT or firewall, ordinary TCP cannot be established since the end initiating the TCP is outside the other end's NAT[2]. This is true even if the connection would be allowed according to each end's firewall security policy. For instance, if the firewall policy is that internal hosts may initiate TCP connections, and both hosts wish to initiate. Recent work has proposed work-arounds that establish a TCP connection without the use of proxies or tunnels [9, 5, 3, 6]. This is accomplished by setting up the necessary connection-state on the NAT through a carefully crafted exchange of TCP packets. However, not all NATs in the wild react the same way, causing these approaches to fail in various cases. Understanding such behavior in NATs and measuring how much they detract from the original goal of universal connectivity in the Internet is crucial to integrating them cleanly into the architecture.

The Internet architecture today is vastly different from that envisioned when TCP/IP was designed. Firewalls and NATs often make it impossible to establish a connection even if it does not violate policy. For instance, Alice and Bob may disallow unsolicited connections by hiding behind a NAT or configuring their firewalls to drop inbound SYN packets. Yet when both Alice and Bob agree to establish a connection there is no way to do so without reconfiguring their NAT since Alice's SYN is dropped by Bob's NAT and vice versa. Even so, NATs and firewalls have become a permanent part of the network infrastructure and will continue to remain so for a long time. Even if IPv6 is deployed globally, IPv4-IPv6 NATs will be needed during the lengthy transition, and IPv6 firewalls will be needed for security. As a result, mechanisms that enable two consenting hosts behind NATs to communicate with each other are needed.

This problem has been solved for UDP by STUN [15]. In STUN, Alice sends a UDP packet to Bob. Although this packet is dropped by Bob's NAT, it causes Alice's NAT to create local state that allows Bob's response to be directed to Alice. Bob then sends a UDP packet to Alice. Alice's NAT considers it part of the first packet's flow and routes it through, while Bob's NAT considers it a connection initiation and creates local state to route Alice's responses. This approach is used by Skype, a popular VoIP application [2]. Unfortunately, establishing TCP is more complicated. Once Alice sends her SYN packet, her OS stack

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

as well as her NAT expect to receive a SYNACK packet from Bob in response. However, since the SYN packet was dropped, Bob's stack doesn't generate the SYNACK. Proposed workarounds to the problem are complicated, their interactions with NATs in the wild are poorly understood, and the extent to which they solve the problem is not known. Consequently applications such as the file-transfer module in Skype use contraindicated protocols like UDP. While such approaches may work, we believe it is important that wherever possible, applications use the native OS TCP stack. This is in part to avoid increasingly complex protocol stacks, but more importantly because TCP stacks have, over the years, been carefully optimized for high performance and congestion friendliness.

Overall this work makes four contributions. First, we identify and describe the complete set of NAT characteristics important to TCP NAT traversal. Second, we measure the prevalence both of these individual characteristics and of the success rate of peer-to-peer TCP connections for the various proposed approaches. Third, based on these measurements, we suggest modifications to the proposed approaches. Fourth, we provide public-domain software toolkit that can be used to measure NATs as they evolve, and as the basis of TCP NAT traversal in P2P applications. Altogether we provide insights for application developers into the inherent tradeoffs between implementation complexity and NAT-traversal success rate. Finally, our results can be used to guide the standardization process of NATs and firewalls, making them more traversal friendly without circumventing security policies.

The rest of the paper is organized as follows. Section 2 discusses the proposed TCP NAT-traversal approaches. Section 3 and Section 4 explain our setup for testing NATs and the observed NAT behavior. Section 5 analyzes port-prediction and Section 6 analyzes peer-to-peer TCP establishment. Section 7 discusses related work. Section 8 concludes the paper.

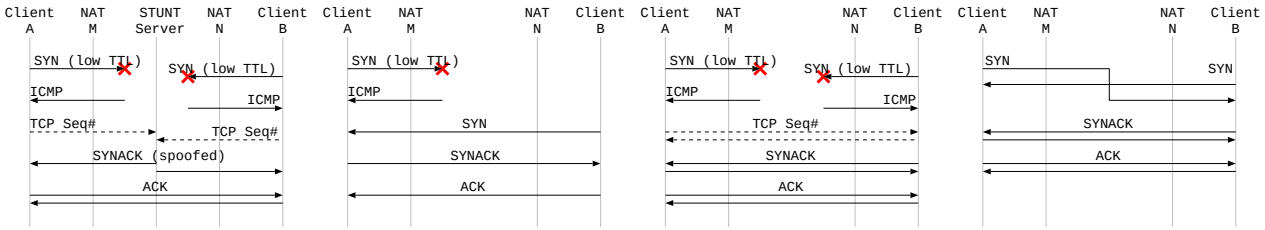
2 TCP NAT-Traversal

In this section we discuss the TCP NAT-traversal approaches that have been proposed in recent literature. In all the approaches, both ends initiate a TCP connection. The outbound SYN packet from each host creates the necessary NAT state for its own NATs. Each approach then reconciles the two TCP attempts into a single connection through different mechanisms as described in this section. The address and port to which these original SYNs are sent is determined through *port prediction*. Port prediction allows a host to guess the NAT mapping for a connection before sending the outbound SYN and is discussed in detail later. The approaches also require some coordination between the two hosts. This is accomplished over an out-of-band channel such as a connection proxied by a third party or a UDP/STUN session. Once the direct TCP connection is established, the out-of-band channel can be closed. The reconciliation mechanism used triggers different behavior in different NATs causing the proposed approaches to fail in many instances. In addition, it is possible for either endpoint to be behind *multiple NATs* in serial[3]. In such cases the behavior observed is a composite of the behavior of all the NATs and firewalls in the path. For brevity we shall overload the term 'NAT' to mean the composite NAT/firewall.

2.1 STUNT

In [9], the authors propose two approaches for traversing NATs. In the first approach, illustrated in Figure 1(a), both endpoints send an initial SYN with a TTL[4] high enough to cross their own NATs, but small enough that the packets are dropped in the network (once the TTL expires). The endpoints learn the initial TCP sequence number used by their OS's stack by listening for the outbound SYN over PCAP or a RAW socket. Both endpoints inform a globally reachable STUNT server of their respective sequence numbers, following which the STUNT server spoofs a SYNACK to each host with the sequence numbers appropriately set. The ACK completing the TCP handshake goes through the network as usual. This approach has four potential problems. First, it requires the endhost to determine a TTL large enough to cross its own NATs and low enough to not reach the other end's NAT. Such a TTL does not exist when the two outermost NATs share a common interface. Second, the ICMP TTL-exceeded error may be generated in response to the SYN packet and be interpreted by the NAT as a fatal error. Third, the NAT may change the TCP sequence number of the initial SYN such that the spoofed SYNACK based on the original sequence number appears as an out-of-window packet when it arrives at the NAT. Fourth, it requires a third-party to spoof a packet for an arbitrary address, which may be dropped by various ingress and egress filters in the network. These network and NAT issues are summarized in Table 1.

In the second approach proposed in [9], similar to the one proposed in [5], only one endhost sends out a low-TTL SYN packet. This sender then aborts the connection attempt and creates a passive TCP socket on the same address and port. The other endpoint then initiates a regular TCP connection, as illustrated in Figure 1(b). As with the first case, the endhost needs to pick an appropriate TTL value and the NAT must not consider the ICMP error a fatal error. It also requires that the NAT accept an inbound SYN following an outbound SYN – a sequence of packets not normally seen.



(a) STUNT #1 (b) STUNT #2 (c) NATBlaster (d) P2PNAT

Figure 1: Packets generated by various TCP NAT-traversal approaches. Solid lines are TCP/IP and ICMP packets pertaining to the connection attempt while dotted lines are control messages sent over an out-of-band channel.

Approach	NAT/Network Issues	Linux Issues	Windows Issues
STUNT #1	• Determining TTL • ICMP error • TCP Seq# changes • IP Address Spoofi ng	• Superuser priv.	• Superuser priv. • Setting TTL
STUNT #2	• Determining TTL • ICMP error • SYN-out SYN-in		• Setting TTL
NATBlaster	• Determining TTL • ICMP error • TCP Seq# changes • SYN-out SYNACK-out	• Superuser priv.	• Superuser priv. • Setting TTL • RAW sockets (post WinXP SP2)
P2PNAT	• TCP simultaneous open • Packet fbod		• TCP simultaneous open (pre WinXP SP2)
STUNT #1 no-TTL	• RST error • TCP Seq# changes • Spoofi ng	• Superuser priv.	• Superuser priv. • TCP simultaneous open (pre WinXP SP2)
STUNT #2 no-TTL	• RST error • SYN-out SYN-in		
NATBlaster no-TTL	• RST error • TCP Seq# changes • SYN-out SYNACK-out	• Superuser priv.	• Superuser priv. • RAW sockets (post WinXP SP2) • TCP simultaneous open (pre WinXP SP2)

Table 1: NAT and network issues encountered by various TCP NAT-traversal approaches as well as the implementation issues we encountered.

2.2 NATBlaster

In [3], the authors propose an approach similar to the first STUNT approach but do away with the IP spoofing requirement (Figure 1(c)). Each endpoint sends out a low-TTL SYN and notes the TCP sequence number used by the stack. As before, the SYN packet is dropped in the middle of the network. The two hosts exchange the sequence numbers and each crafts a SYNACK packet the other expects to receive. The crafted packet is injected into the network through a RAW socket; however, this does not constitute spoofing since the source address in the packet matches the address of the endpoint injecting the packet. Once the SYNACKs are received, ACKs are exchanged completing the connection setup. As with the first STUNT approach, this approach requires the endpoint to properly select the TTL value, requires the NAT to ignore the ICMP error and fails if the NAT changes the sequence number of the SYN packet. In addition, it requires that the NAT allow an outbound SYNACK immediately after an outbound SYN – another sequence of packets not normally seen.

2.3 Peer-to-Peer NAT

In [6], the authors take advantage of the simultaneous open scenario defined in the TCP specifications [13]. As illustrated in Figure 1(d), both endpoints initiate a connection by sending SYN packets. If the SYN packets cross in the network, both the endpoint stacks respond with SYNACK packets establishing the connection. If one end's SYN arrives at the other end's NAT and is dropped before that end's SYN leaves that NAT, the first endpoint's stack ends up following TCP simultaneous open while the other stack follows a regular open. In the latter case, the packets on the wire look like the second STUNT approach without the low-TTL and associated ICMP. While [6] does not use port-prediction, the approach can benefit from it when available. As with the second STUNT approach, it requires that the NAT accept an inbound SYN after an outbound SYN. In addition, the approach requires the endhost to retry failed

connection attempts in a tight loop until a timeout occurs. If instead of dropping the SYN packet, a NAT responds to it with a TCP RST then this approach devolves into a packet flood until the timeout expires.

2.4 Implementation

We implemented all the above approaches on both Linux and Windows. We also developed a Windows device driver that implements the functionality required by the approaches that are not natively supported by Windows. The first STUNT approach requires superuser privileges under both Windows and Linux to overhear the TCP SYN packet on the wire and learn its sequence number. In order to set the TTL on the first SYN packet, we use the `IP_TTL` socket option under Linux and our driver under Windows. We also implemented the STUNT server and host it behind an ISP that does not perform egress filtering in order to spoof arbitrary addresses. While the server was able to spoof most SYNACKs, it was not successful in spoofing SYNACKs where both the source and destination were in the same administrative domain and the domain used ingress filtering. When possible, we install an additional STUNT server inside such domains. The second STUNT approach requires the driver to set the TTL under windows. The NATBlaster approach requires superuser privileges to learn the sequence number of the SYN and inject the crafted SYNACK through a RAW socket. Due to a restriction introduced in Windows XP SP2, the approach requires the driver to inject this packet. The P2PNAT approach requires the OS to support TCP simultaneous open; this is supported under Linux and Windows XP SP2 but not by Windows XP prior to SP2. On Windows XP SP1 and earlier our driver adds support for this. These implementation issues are summarized in Table 1.

We found that setting the TTL is problematic under Windows; therefore, we consider the consequences of not using it. If the TTL is not reduced, the first SYN sent by one of the hosts reaches the other end's NAT before that end's SYN exits the same NAT. The NAT can either silently drop the inbound packet, or respond with an ICMP unreachable error or a TCP RST/ACK. The response, if any, may trigger transitions in the sender's NAT and OS stack unaccounted for by the approach. If the TTL for the other end's SYN packet is not reduced either, the SYN may reach the intended destination triggering unforeseen transitions. The behavior may be favorable to the ultimate goal if, for instance, it triggers a TCP simultaneous-open, or it may be detrimental if it confuses the stack or NAT. We implement modified versions of the above approaches that do not use low TTLs. The network and implementation issues corresponding to the modified approaches are listed in Table 1.

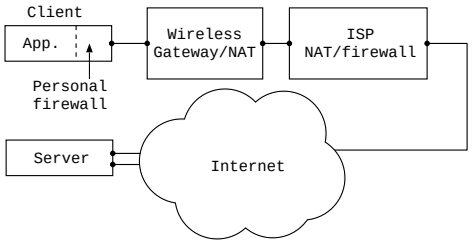


Figure 2: A possible experiment setup for STUNT. Client component is behind multiple NATs and the server component is outside all of them. The behavior determined by STUNT is the composite of the individual NAT behaviors.

Brand	Model	Firmware
3Com	3C857	2.02
Allied Telesyn	AT-AR220E	R1.13
Belkin	F5D5231-4	1.0.0
Buffalo	WYR-G54	1.0 r31
Checkpoint	VPN-1/FireWall-1	(NGAI) release 55
DLink	DI-604	3.30
Linksys	BEFSR41	1.40.2
Linux	iptables	2.4.20
Netgear	RP614	5.13
Netopia	3386	8.0.10
Open BSD	pf	3.5
SMC	SMC7004VBR	R1.00
Trendnet	TW100-S4W1CA	1.02.000.267
USR	8003	1.04 08
VMWare	Workstation	4.5.2
Windows XP	Internet Connection Sharing	SP2

Table 2: NATs tested in the lab, chosen to represent a variety of brands and implementations.

3 Experiment Setup

We have defined the STUNT client-server protocol that both tests NAT/firewall behavior and assists in establishing TCP connections between NAT'ed peers. A complete protocol description is available in [7]. The protocol is implemented by our test applications comprising a client component and server component. As shown in Figure 2, the client is run on a host behind one or more NATs while the server is external to all of them. The STUNT test client detects the composite behavior of all the NATs and firewalls between the client and the server. While both the test client and server require superuser privileges to analyze raw packets on the wire, the requirement can be dropped for the client in exchange for a small loss in functionality. The server, in addition, requires at least two network interfaces to properly differentiate between various NAT port allocation algorithms in use. The tests performed by the client and the NAT characteristics inferred from them are described later in Section 4.

We used the client to test a diverse set of sixteen NATs in

Brand	Sample	Market Survey
Linksys	22.6%	28.8%
D-Link	9.7%	20.2%
Netgear	6.5%	14.7%
Buffalo Technologies	2.2%	10.9%
Belkin	8.6%	4.6%
Other	50.5%	20.9%

Table 3: Observed market share of NAT brands in our sample set and worldwide SOHO/Home WLAN market share of each brand in Q1 2005 according to Synergy Research Group

the lab (Table 2). These include one of each brand of NAT that we could find in online stores in the United States. The NATs tested also include software NAT implementations in popular operating systems. In each lab test the client host was the only host internal to the NAT and the server was on the same ethernet segment as the external interface of that NAT. The client host was set to not generate any network traffic other than that caused by the test client.

In addition to these lab tests, we also tested NAT boxes in the wild. The main reason for this was to expose our test software to a wider range of scenarios than we could reproduce in the lab, thus improving its robustness and increasing our confidence in its operation. In addition, it provided a sample, albeit small, of what types of NAT we can expect to see in practice. For this test, we requested home users to run the test client. This tested 93 home NATs (16 unique brands) being used by CS faculty and students at Cornell and other universities. Test traffic was in addition to typical network activity on the client host and other hosts behind the NAT and included web browsing, instant messaging, peer-to-peer file-sharing, email etc. The resulting data draws from a mix of NAT brands with both new and old models and firmware; however, it admits a bias in the selection of NATs given the relatively small user base with most of them living in the north-eastern United States. This discrepancy is evident in Table 3, where the observed popularity of brands in our sample is listed under 'Sample' and the worldwide SOHO/Home WLAN market share of the brands in the first quarter of 2005 as per the Synergy Research Group [18] is listed under 'Market Survey'. In particular, Buffalo Technologies and Netgear were under-represented in our sample and the percentage of other brands was significantly higher. The full list of home NATs tested is available in [8].

4 NAT TCP Characteristics

In this section, we identify how different NATs affect TCP NAT-traversal approaches. We identify five classifications for NAT behavior; namely, NAT mapping, endpoint packet filtering, filtering response, TCP sequence number preserving, and TCP timers. The classifications and the possible values that a NAT can receive in each class are listed in Table 4. We use the STUNT testing client and server described earlier in Section 3 to classify a collection of sixteen NATs in the lab and ninety-three NATs in the wild. The full set of test results with a wider set of classifications is available in [8].

Classification	Values
Nat Mapping	Independent Address$_\delta$ Port$_\delta$ Address and Port$_\delta$ Connection$_\delta$
Endpoint Filtering	Independent Address Port Address and Port
Response	Drop TCP RST ICMP
TCP Seq#	Preserved Not preserved
Timers	Conservative Aggressive

Table 4: Important categories distinguishing various NATs.

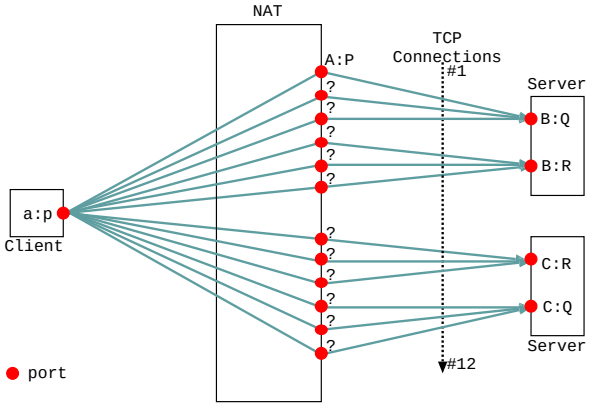


Figure 3: TCP connections established to find NAT Mapping classification. Client uses the same local IP address and port `a:p` to connect three times to two ports `Q` and `R` on two servers at IP addresses `B` and `C`. The pattern of mapped address and port for each connection determines the NAT Mapping classification.

4.1 NAT Mapping

A NAT chooses an external mapping for each TCP connection based on the source and destination IP and port. Some NATs reuse existing mappings under some conditions while others allocate new mappings every time. The *NAT Mapping* classification captures these differences in mapping behavior. This knowledge is useful to endhosts attempting to traverse NATs since it allows them to to predict the mapped address and port of a connection based on previous connections. For UDP, it is known that some NATs

#	From	To	Nat1	Nat2	Nat3	Nat4	Nat5	Nat6
1	a:p	B:Q	A:P	A:P	A:P	A:P_1	A:P	A:P
2	a:p	B:Q	A:P	A:P	A:P+1	A:P_2	A:P	A:P
3	a:p	B:Q	A:P	A:P	A:P+2	A:P_3	A:P	A:P
4	a:p	B:R	A:P	A:P+1	A:P+3	A:P_4	A:P+1	A:P
5	a:p	B:R	A:P	A:P+1	A:P+4	A:P_5	A:P+1	A:P
6	a:p	B:R	A:P	A:P+1	A:P+5	A:P_6	A:P+1	A:P
7	a:p	C:R	A:P	A:P+2	A:P+6	A:P_7	A:P+1	A:P+1
8	a:p	C:R	A:P	A:P+2	A:P+7	A:P_8	A:P+1	A:P+1
9	a:p	C:R	A:P	A:P+2	A:P+8	A:P_9	A:P+1	A:P+1
10	a:p	C:Q	A:P	A:P+3	A:P+9	A:P_{10}	A:P	A:P+1
11	a:p	C:Q	A:P	A:P+3	A:P+10	A:P_{11}	A:P	A:P+1
12	a:p	C:Q	A:P	A:P+3	A:P+11	A:P_{12}	A:P	A:P+1
13	a:s	B:Q	A:S	A:S	A:S	A:S_1	A:S	A:S
⋮								
Classification			NB: Independent	NB:Address and $Port_1$	NB: $Connection_1$	NB: $Connection_{\Re}$	NB: $Port_1$	NB: $Address_1$

Table 5: NAT Mapping test behavior observed. Nat1–5 show the 5 different mapping patterns that are observed in practice. Nat6 is a possible mapping pattern that has not been observed in our sample set.

assign a fixed address and port for all connections originating from a fixed source address and port [15]. We test for similar behavior for TCP in the STUNT client. The client establishes 12 connections in close succession from a fixed local address and port `a:p` to various server addresses and ports as shown in Figure 3 and tabulated in Table 5. Each connection is closed before the next is initiated. The server echoes back the mapped address and port it perceives to the client. The test is repeated multiple times for different choices of the local port.

We notice several distinct patterns among the mapped ports shown as Nat1–Nat6 in Table 5. Let the mapping allocated for the first connection be called `A:P` for each NAT. Nat1 reuses this mapping as long as the client source address and port of a new connection matches that of the first connection. We classify this behavior as *NB:Independent* since the mapping is determined only by the source address and port and is independent of the destination address and port. This is equivalent to *cone behavior* in [15] extended to include TCP. Nat2 reuses the mapping only if both the source and destination address and port for the new connection match the first connection. Such NATs are classified *NB:Address and* $Port_1$ since both the destination address and port affect the mapping. The subscript '$_1$' signifies that the difference between new mappings, denoted by δ, is 1. [17] shows that for UDP, δ is fixed for many NATs and is usually 1 or 2. We find that the same holds for TCP as well, however, all of the NATs we encountered have $\delta = 1$. Nat3 allocates a new mapping for each new connection, however, each new mapping has port $\delta = 1$ higher than the previous port. We classify Nat3 as *NB:*$Connection_1$. Nat4, like Nat3, allocates a new mapping for each TCP connection but there is no discernable pattern between subsequent mappings. We classify such NATs *NB:*$Connection_{\Re}$ where the subscript '$_{\Re}$' indicates a random δ. Nat5 is a variation of Nat2 where the mapping is reused if the destination port matches in addition to the source address and port. Nat6 is similar except the destination address needs to match instead of the port. Together Nat5 and Nat6 are classified *NB:*$Port_1$ and *NB:*$Address_1$ respectively. NATs 2–6 display *symmetric behavior* as per [15].

NAT Mapping	Lab	Wild
NB:Independent	9	70.1%
NB:Address and $Port_1$	3	23.5%
NB:$Connection_1$	3	3.9%
NB:$Port_1$	0	2.1%
NB:$Address_\delta$	0	0.0%
NB:$Connection_{\Re}$	1	0.5%

Table 6: NAT mapping types observed in a set of 16 NATs in the lab, and that estimated for NATs in the wild based on our sampling of 87 home NATs and worldwide market shares.

Table 6 shows the relative proportion of each type of NAT. Column 2 shows the number of NATs from our testbed of sixteen NATs that were classified as a particular type. A majority of them are NB:Independent. The only one that is NB:$Connection_{\Re}$ is the NAT implementation in OpenBSD's `pf` utility. We also noticed that our Netgear RP614 NAT with firmware 5.13 is NB:$Connection_1$, however, more recent Netgear NATs such as MR814V2 with firmware 5.3_05 are NB:Independent. Column 3 estimates the behavior of NATs in the wild. The estimates are computed by taking the proportion of each type and brand of NAT from eighty-seven home NATs sampled and scaling them with a correction factor chosen to overcome the bias in our sample. The correction factor for each brand is the ratio between the surveyed and observed market shares presented in Table 3. The factor serves to increase the contribution of under-represented brands in the estimated result and decrease the contribution of over-represented brands.

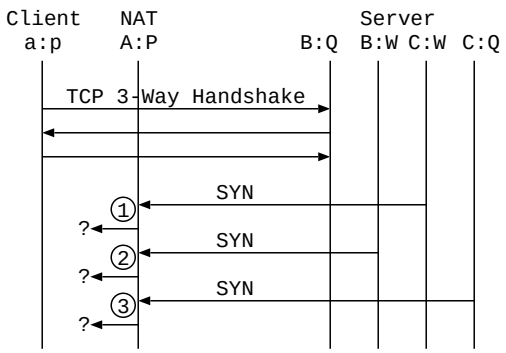


Figure 4: TCP packets exchanged for Endpoint Filtering test. Client establishes a connection to `B:Q`. Packet (1) is an inbound SYN from a different address and port (`C:W`), (2) from the same address but different port (`B:W`) and (3) from the same port but different address (`C:Q`). The response to each of these determines the Endpoint Filtering classification.

While our estimates are indicative of the general trend to the best of our knowledge, we note that in an industry changing at the rate of 47% per year [18] the accuracy of any results is short lived at best. Nevertheless, we estimate a majority of the NATs (70.1%) to be NB:Independent and almost none to be NB:Connection$_{\Re}$. A significant percent (29.9%) of NATs have symmetric behavior. Consequently in a large fraction of cases, multiple connections from the same port will not be assigned the same mapping and applications must employ the more sophisticated port-prediction techniques described later.

4.2 Endpoint Filtering

Both NATs and firewalls may filter inbound packets addressed to a port unless certain conditions are met. If no NAT mapping exists at that port, a NAT is forced to filter the packet since it cannot forward it. If a mapping exists, however, or if it the device is a firewall then it may require that the source address and/or port of the inbound packet match the destination of a preceeding outbound packet. These differences in conditions that trigger filtering is captured by the *Endpoint Filtering* classification. The STUNT test client determines this by first establishing NAT state by connecting to the server. It then requests the server to initiate connections to the mapped address and port from different addresses and ports as shown in Figure 4.

The different filtering behavior observed for the test are tabulated in Table 7. Nat1′ accepts all three SYN packets. Such NATs allow inbound TCP connections independent of the source address and port as long as necessary state exists for routing the request. We classify such NATs as having the endpoint filtering behavior *EF:Independent*. Nat2′ filters all the packets thus requiring the source of the inbound TCP packet match both the address and port of the destination of the connection that created the mapping. The endpoint filtering of such NATs is classified *EF:Address and Port*. Nat3′ and Nat4′ allow inbound packets from the same address or port as the destination address of the connection but filter packets from a different address or port respectively. We classify the endpoint filtering behavior of such NATs as *EF:Address* and *EF:Port* respectively. In general we find that endpoint filtering behavior of a NAT is independent of NAT mapping behavior. The subclassifications of cone NATs defined in [15] translate as follows: *full cone* is equivalent to NB:Independent and EF:Independent, *restricted cone* is NB:Independent and EF:Address and Port and *port restricted cone* is NB:Independent and EF:Port.

Endpoint Filtering	Lab	Wild
Address and Port	12	81.9%
Address	1	12.3%
Independent	3	5.8%

Table 8: NAT endpoint filtering types observed in a set of 16 NATs in the lab, and that estimated for NATs in the wild based on our sampling of 93 home NATs and worldwide market shares.

Sequence	Filtered
SYN-out SYNACK-in	0%
SYN-out SYN-in	13.6%
SYN-out ICMP-in SYNACK-in	6.9%
SYN-out ICMP-in SYN-in	22.4%
SYN-out RST-in SYNACK-in	25.6%
SYN-out RST-in SYN-in	28.1%

Table 9: Percentage of NATs not accepting various packet sequences. Inbound packets (-in) are in response to preceeding outbound packets (-out). ICMP code used is TTL-exceeded (non-fatal error).

Table 8 shows the endpoint filtering classification for sixteen NATs in the lab and the estimated percentage of NATs in the wild. The estimates are computed based on the market survey as described earlier. 81.9% of the NATs are estimated to be EF:Address and Port while only 5.8% are EF:Independent. This implies that in most cases, to establish a connection between two NAT'ed hosts an outbound SYN must be sent from each end before inbound packets are accepted.

4.2.1 TCP State Tracking

NATs implement a state machine to track the TCP stages at the endpoints and determine when connection-state can be garbage-collected. While all NATs handle the TCP 3-way handshake correctly, not all of them implement the corner cases of the TCP state machine correctly thereby prematurely expiring connection-state. The STUNT client and server test how NAT/firewall implementations affect TCP NAT-traversal approaches by replaying the packet sequences observed for these approaches.

#	From	To	Nat1′	Nat2′	Nat3′	Nat4′
1	C:W	A:P	accepted	filtered	filtered	filtered
2	B:W	A:P	accepted	filtered	accepted	filtered
3	C:Q	A:P	accepted	filtered	filtered	accepted
Classification			EF:Independent	EF:Address and Port	EF:Address	EF:Port

Table 7: NAT endpoint filtering behavior observed. Nat1′–Nat4′ show 4 different filtering behaviors that are observed for inbound SYN packets after an internal host establishes a connection from `a:p` to `B:Q` with allocated mapping `A:P`.

Table 9 lists some of the packet sequences tested. We estimate that 13.6% of NATs do not support TCP simultaneous open where an outbound SYN is followed by an inbound SYN. This affects the P2PNAT approach, which requires at least one end support simultaneous open as well as the second STUNT approach. 6.9% filter inbound SYNACK packets after a transient ICMP TTL-exceeded error. A similar number of NATs drop the inbound SYN packet after the ICMP but accept it in the absence of the error. This behavior affects all the approaches that set low-TTLs on SYN packets. A fair number of NATs (28.1%) accept inbound SYN packets even after the SYN packet that created the connection-state is met with a fatal TCP RST. This mitigates the issue of spurious RSTs that some approaches contend with. Not mentioned in the table is the sequence SYN-out SYNACK-out that is required for the NATBlaster approach. We did not test this case widely due to restrictions introduced by Windows XP SP2. In the lab, however, we found that the D-Link NAT (DI-604) does not support it.

4.2.2 Filtering Response

When an inbound packet is filtered by a NAT it can choose to either drop the packet silently or notify the sender. An estimated 91.8% of the NATs simply drop the packet without any notification. The remaining NATs signal an error by sending back a TCP RST acknowledgment for the offending packet.

4.3 Packet Mangling

NATs change the source address and port of outbound packets and the destination address and port of inbound packets. In addition, they need to translate the address and port of encapsulated packets inside ICMP payloads so endhosts can match ICMPs to their respective transport sockets. All the NATs in our sample set either perform the ICMP translation correctly or filter the ICMP packets, which are not always generated in in the first place. Some NATs change the TCP sequence numbers by adding a constant per-flow offset to the sequence number of outbound packets and subtracting the same from the acknowledgment number of inbound packets. We estimate that 8.4% of NATs change the TCP Sequence Number. Consequently in some cases, TCP NAT-traversal approaches that require the initial sequence number of the packet leaving the NAT cannot use the sequence number of the SYN at the end host in its stead.

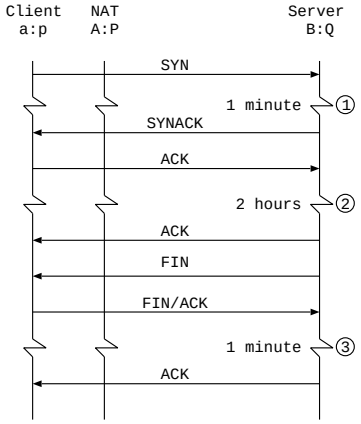


Figure 5: NAT timers in effect during a TCP connection. (1) Timer in SYN_SENT state, (2) Timer in established state, (3) Timer in TIME_WAIT.

4.4 TCP Timers

NATs and firewalls cannot indefinitely hold state since it makes them vulnerable to DoS attacks. Instead they expire idle connections and delete connection-state for crashed or misbehaving endpoints. In addition, they monitor TCP flags and recover state from connections explicitly closed with a FIN/FINACK exchange or RST packets. They might allow some grace time to let in-flight packets and retransmissions be delivered. Once connection-state has been unallocated, any late-arriving packets for that connection are filtered. NATs typically use different timers for these cases as illustrated in Figure 5. At location 1 and 3 in the figure, they use a short timer to expire connections not yet established or connections that have been closed respectively. RFC 1122 [4] requires that endhosts wait for 4 minutes ($2\times$`MSL`[5]) for in-flight packets to be delivered; however, most operating systems wait for about 1 minute instead. At location 2 in the figure, NATs use a longer timer for idle connections in the established state. The corresponding requirement in RFC 1122 is that TCP stacks should wait for at least 2 hours between sending `TCP-Keepalive` packets over idle connections.

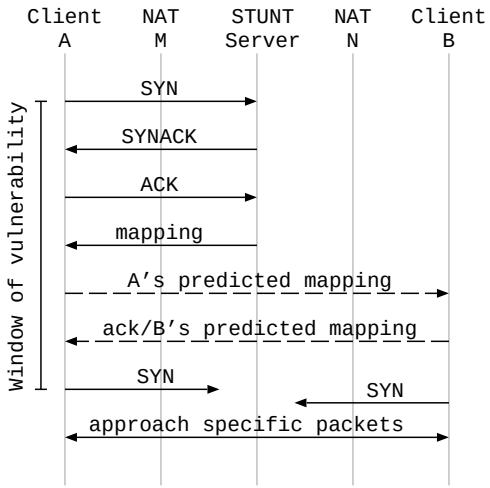


Figure 6: Port-prediction in TCP NAT-Traversal approaches.

The STUNT test client checks the NAT timers for compliance with current practice and RFCs. It does so by performing three timed tests to check each case separately. In the first test, a TCP connections is initiated by the client, but the SYNACK from the server is delayed by a little under a minute. In the second test, the connection is established and left idle for a little under 2 hours, at which point a couple of bytes are sent across from the server. In the third test, the connection is established and then closed but the last ACK from the server is delayed by about a minute. In each case if the packet sent from the server after the introduced delay is delivered to the client then the corresponding timer is termed *conservative*, otherwise it is termed *aggressive*. We estimate only 27.3% of the NATs have conservative timers for all three cases while 35.8% have a conservative timer for the second case. 21.8% of the NATs have an extremely aggressive timer for the second case where they expire an established connection after less than 15 minutes of inactivity. This implies that applications should not rely on idle connections being held open for more than a few minutes.

5 Port Prediction

Port prediction allows a host to predict the mapped address and port for a connection before initiating it. It therefore allows two hosts to initiate a connection with each other's mapped address and port even though the mapping is allocated by the NAT *after* the connection is initiated. Figure 6 shows a typical TCP NAT-traversal attempt using port prediction information. In the figure, we assume that A has already determined the type of NAT it is using. When client A wishes to establish a connection with client B, A first establishes a TCP connection to the STUNT server and learns the mapping. Based on the NB:type of NAT M, A predicts the mapping for the next connection. B does the same and both A and B exchange their predictions over an out-of-bound channel. Each end then initiates a connection to the other's mapped address and port by sending a SYN packet. The remainder of the packets exchanged are designed to reconcile the two TCP attempts into one connection and vary from one NAT-Traversal approach to another as described in Section 2. The period between the first SYN and the SYN for the target connection may be a *window of vulnerability* depending on the type of NAT M. For some types of NAT, if another internal host A′ behind NAT M initiates an outbound connection in this period, M will allocate the mapping predicted by A to the connection from A′ instead.

Port-prediction depends on the NAT Mapping type explored earlier in Section 4.1. If the NAT is of type NB:Independent then the mapping for the connection to the STUNT server will be reused for any connection initiated soon afterward from the same source address and port. Since the reuse of the mapping is completely under the client's control, the window of vulnerability does not exist in this case. However, this approach introduces a latency of $2\times$RTT[6] to the STUNT server before the mapping can be predicted. For a possible optimization, we noticed that a number of NATs usually allocate a mapped port equal to the source port used by the client. We term these NATs *port preserving*. Clients behind such a NAT can, with high probability, predict the mapped port without first establishing a connection to the STUNT server. If the NAT is not NB:Independent but has a fixed δ then a connection initiated immediately after the server connection will have a mapped port δ higher than the mapped port observed by the server. Since the mapping changes from connection to connection, a "rogue" connection attempt in the window of vulnerability can steal the mapping. In addition, this approach fails if the predicted mapping is already in use, causing the NAT's allocation routine to jump over it onto the next available one.

We implemented port prediction in the STUNT test client and predicted mappings for eighty-three home users for an hour. Every minute, the test client initiates a connection to the STUNT server from a source address and port and learns the mapping allocated. Next it uses the same source address and port to initiate a connection to a remote host setup for the purpose of this experiment. The test client checks the mapping actually observed for the second connection against the one predicted based on the mapping for the first and type of NAT. Port-prediction is successful if and only if they match. The predictions are performed while users use their host and network normally. This includes web browsers, email readers, instant messaging and file-sharing applications running on the client host and other hosts behind the same NAT. 88.9% of the NB:Independent NATs are port preserving. This represents a big win for interactive applications that implement the optimization above. We find that in 81.9% of the cases the

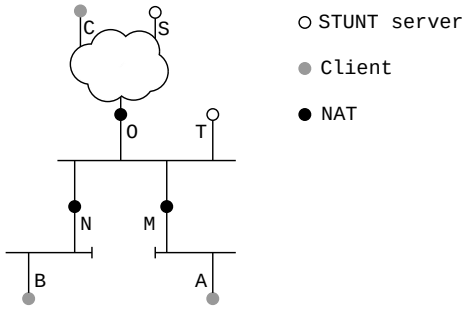


Figure 7: Problematic scenarios for port prediction.

port was predicted correctly every time. This includes all but one of the NB:Independent NATs and 37.5% of the non-NB:Independent NATs. For the remaining 62.5% of the latter variety, at least one time out of the sixty another host or application stole the mapping the test client predicted for itself. In one particular case, the client host behind a NB:Connection$_1$ NAT was infected with a virus that generated several randomly-addressed SYN packets per second causing all predictions to fail! In another case, the user initiated a VPN connection midway through the test causing all subsequent requests to be sent over the VPN and thus through a different type of NAT. This suggests that long-running applications may cache the NAT mapping type for some time but must revalidate it from time to time. Overall, in 94.0% of the cases, more than three-fourths of the port predictions were correct. Hence after a failed attempt if an application simply retries the connection, it is likely to succeed.

5.1 Problems

Port prediction has several corner cases where it can fail. In Figure 7 if A uses STUNT server T to predict an address and port when trying to establish a connection to C it would end up learning NAT M's external address instead of NAT O's external address. Port prediction requires that the line of NATs between the client and STUNT server be the same as that between the client and the most external NAT of the endpoint it wishes to connect with. Therefore A somehow needs to discover S in order to connect to C. If, however, A wishes to communicate with B and both use STUNT server S then their port prediction attempts can interfere with each other, preventing either from correctly predicting the port. In addition, even if the port is predicted correctly, both A and B will end up using O's external address. This scenario is called a *hairpin translation* since A's SYN addressed to B's predicted address and port (O's external address in this case) will be delivered to O, which needs to send it back out on an *internal* interface. Not all NATs handle hairpin translations correctly and we estimate this erroneous behavior to be as high as 72.8% based on tests performed by the STUNT test client.

The port-prediction technique described earlier does not handle NB:Connection$_{\Re}$ NATs since sequential connections are randomly assigned. [3] proposes an interesting technique for handling such cases that uses the birthday paradox to cut down on the number of guesses before a collision is found. The technique initiates 439 connections such that the guessed port will match one of them with 95% probability. Unfortunately, we find that some NATs, like Netgear, limit the total number of pending connection attempts to 1000 causing this approach to quickly stifle the NAT. Fortunately, very few NATs demonstrate NB:Connection$_{\Re}$ behavior mitigating the problem.

6 TCP Establishment

In this section we estimate the success of the various NAT traversal approaches as well as report our experience with peer-to-peer TCP establishment for a small wide-area testbed. The success of TCP NAT-traversal approaches depends on the behavior of all NATs between the two endhosts as well as the activity of other hosts behind the NATs. Section 4 analyzes a variety of NATs in isolation while Section 5 analyzes competing network activity and its effect on port prediction. Combining the results from these sections we can quantitatively estimate the success of each NAT traversal approach.

We make the following assumptions about the deployment of the TCP-traversal approaches. We assume that STUNT servers are deployed widely enough to ensure that for each pair of hosts, there is a STUNT server that meets the port-prediction requirements and can spoof packets appearing to come from the mapped address and port of each host. We assume that endhost stacks can be fixed so all software issues at the ends are resolved. Lastly, since we lack data to model the scenarios presented in Section 5.1 we assume the contribution from such scenarios to be negligible. As a result of these assumptions, our estimates may be optimistic.

Peer-to-peer TCP establishment depends on the NATs at both ends. An endpoint with an unpredictable NAT may still be able to establish a connection if the other endpoint's NAT is predictable but not if it is unpredictable. We estimate TCP connectivity in the wild by considering all pairs of NAT behavior observed in practice. Figure 8 plots the estimated success rate of various TCP NAT traversal approaches. We plot the two STUNT approaches (#1 and #2), NATBlaster and P2PNAT as proposed in [9, 3, 6]. In addition, we plot modified versions of STUNT #1 and #2 and NATBlaster approaches that do not use low-TTLs. We also plot a modified version of the P2PNAT approach that uses port-prediction. There is a race condition between the SYN packets in some of these approaches that leads to spurious packets for certain NAT-pairs. The light-gray bars repre-

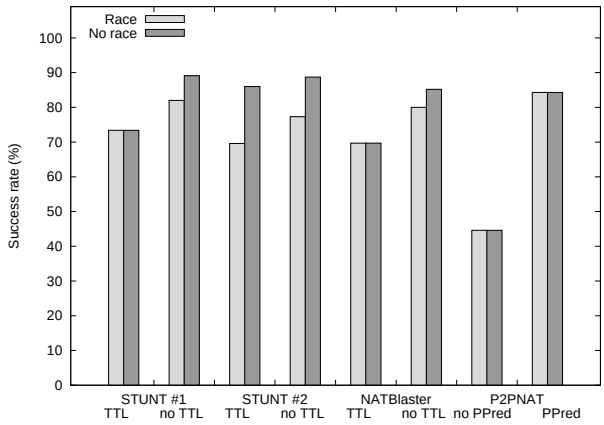


Figure 8: Estimated peer-to-peer TCP NAT traversal success rates of various approaches. With prevailing race conditions (Race) the success rate is lower than when races are resolved for the best outcome (No race).

sent the success rate when each end has an equal chance of winning the race; this corresponds to simultaneous invocation of the approach on both ends. The dark-gray bar represents the success rate when the race is broken in favor of a successful connection; this corresponds to two separate invocations of the approach where for the first invocation, one end is initiated slightly before the other while for the second invocation, the order is reversed. The attempt is declared successful if either invocation succeeds in establishing a TCP connection.

As shown in the graph, the original approaches proposed succeed 44.6% and 73.4% of the time for P2PNAT and STUNT #1 respectively. Breaking the race condition in the original STUNT #2 approach by trying it once from each end boosts its success to 86.0%. Similarly, adding port prediction to the P2PNAT approach allows it to handle symmetric NATs increasing its success rate to 84.3%. Surprisingly, modifying the original approaches to not use low-TTLs benefits all of them by $\sim$5%! Breaking the race conditions thus introduced yields the best success rates of 89.1% and 88.7% for the two modified STUNT approaches and 85.2% for the modified NATBlaster approach.

The unexpected benefits to *not* using low-TTL SYNs are explained as follows. A large fraction of NATs silently drop the first SYN packet (Section 4.2.2) and only a small fraction of NATs filter inbound SYN packets after the outbound SYN packet (Table 9). Consequently in a large number of cases, the modified approaches end up triggering TCP simultaneous open even though they do not intend to. The small penalty they pay for NATs that generate a TCP RST response is more than compensated for by the successful TCP simultaneous opens. This advantage is eroded away if more NATs respond with TCP RST packets or if the endhost's operating system does not support TCP simultaneous

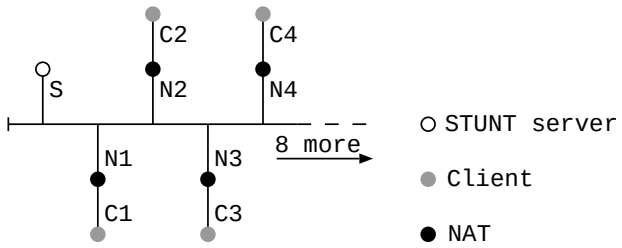


Figure 9: Network of 12 clients for peer-to-peer TCP NAT-traversal tests.

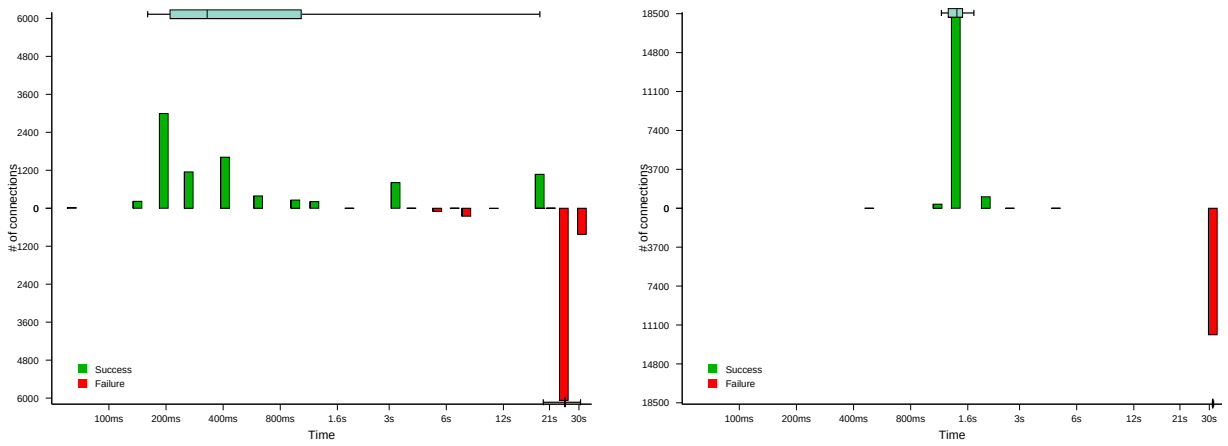
open.

6.1 Implementation

We implemented the above approaches in a peer-to-peer program written in C. The program was run on 12 windows clients connected in a LAN as shown in Figure 9 as well as a slightly larger set of 20 clients connected over a WAN. Each client randomly picks another client and attempts to establish TCP with it. While all the approaches work as advertised, we limit this discussion to STUNT #2 without low-TTL and P2PNAT with port prediction. This is because we find that a large global deployment of STUNT #1 and NATBlaster approaches is impractical; the STUNT #1 approach requires a broad deployment of servers that can spoof arbitrary packets while the NATBlaster approach requires RAW socket functionality that has been removed following security concerns in Windows XP.

Figure 10 shows a semi-log plot of the time taken by each of the approaches to establish a connection or report a failure in a low-latency network. The time-distribution of successful connections (plotted above the x-axis) varies largely for the P2PNAT approach while that of the second STUNT approach is very consistent. This is because the P2PNAT approach repeatedly initiates a connection until one of them succeeds or a timeout occurs while the second STUNT approach only initiates one connection. From the graph in Figure 10(a), a fair number of connections do not succeed until 21 seconds into the connection attempt, thus requiring a large timeout to achieve the estimated success rate determined earlier. In several cases a dangerous side-effect of such a large timeout is observed when port-prediction fails and a peer's NAT responds with TCP RST packets. This causes the P2PNAT approach to generate a SYN-flood as the endhost repeatedly retries the connection until the timeout expires. Admittedly, this problem does not exist in the original P2PNAT approach since it does not advocate port-prediction.

Overall, we find that TCP based protocols can traverse NATs most of the time. Applications must perform port prediction to deal with "delta" type NATs and employ out-of-band signalling to setup connections while satisfying se-



(a) P2PNAT

(b) STUNT #2

Figure 10: Semi-log plot of time taken to successfully establish a connection or report a failure.

curity contraints at the NATs. Non-latency-sensitive applications must also attempt a couple of connections from either side before giving up. Furthermore, applications must not rely on NATs preserving idle connections for more than a few minutes and should not expect TCP sequence numbers to remain unchanged end-to-end. While all the TCP NAT traversal approaches falter under certain scenarios, the second STUNT approach is the most robust in establishing peer-to-peer TCP connections in the Internet today. It succeeds 88% of the time and does not require spoofing, RAW sockets, or superuser privileges. It works with legacy TCP stacks that do not support TCP simultaneous open, while taking advantage of the same in modern operating systems to provide 100% success in many common scenarios. Finally, we find that it is the simplest to implement out of all the approaches and works under both Linux and Windows. We encourage application developers to adapt this approach to provide TCP NAT traversal in their peer-to-peer applications that currently cannot establish TCP between NAT'ed peers.

7 Related Work

NAT traversal is an idea that has long existed since it was first proposed for UDP by Dan Kegel, and used for P2P gaming, in the late 90's. His basic approach was standardized and published in [15]. The UDP approach has resulted in several working documents and Internet drafts that classify UDP behavior of NATs [11] and propose standardization of the same [1]. The area of TCP NAT-traversal without explicit control of the NAT, however, is fairly new. Several approaches have been analyzed in this paper [9, 5, 3, 6] and have been demonstrated to traverse NATs in the current internet. [6] includes a similar study where the authors test a small subset of UDP and TCP NAT characteristics for a number of NATs. We present the first broad study of NAT behavior and peer-to-peer TCP establishment for a comprehensive set of TCP NAT traversal techniques over a wide range of commercial NAT products.

Protocols such as UPnP [12] and MIDCOM [16] allow endhost applications to explicitly control the NAT in order to facilitate peer-to-peer connections. The downside of this type of approach is that they require that these protocols exist and are enabled in the NAT. An application developer cannot depend on either of these, and so for the time being they are not an attractive option.

Another endhost approach to TCP connectivity includes TURN [14], where TCP data is proxied by a third party. This third party represents a potential network bottleneck. Teredo [10] allows IPv6 packets to traverse IPv4 NATs by tunneling IPv6 over UDP. Here, TCP runs natively in the sense that it is layered directly above IPv6. Teredo has been implemented in the Windows OS.

8 Conclusion and Future Work

This paper presents the first measurement study of a comprehensive set of NAT characteristics as they pertain to TCP. While this study shows that there are a significant number of problems with TCP traversal of current NATs, it nevertheless gives us much to be optimistic about. Even with existing NATs, which have not been designed with TCP NAT traversal in mind, we are able to show a 88% average success rate for TCP connection establishment with NATs in the wild, and a 100% success rate for pairs of certain common types of NAT boxes. These numbers are especially encouraging given that only a couple years ago, it was widely assumed that TCP NAT traversal was simply

not possible. While a failure rate of 11% is not acceptable for many applications, the users of those applications at least have the option of buying NAT boxes that allow TCP traversal, and NAT vendors have the option of designing their NAT boxes to be more TCP friendly.

This study is limited in several respects. First, we did not test all existing NAT boxes. For the most part our study was limited to NAT boxes available in North America. Second, our field test is too small and biased to be able to accurately predict success rates broadly. Using market data helps, but even this data was limited in scope and in detail. Third, we tested only for home NATs. Port prediction in enterprise networks for "delta" type NATs will succeed less often, and it would be useful to measure this. Fourth, although TCP NAT traversal techniques apply broadly to firewalls, we did not test firewalls outside the context of NAT. Nor did we test IPv4-IPv6 translation gateways. Finally, like most measurement studies, this is a snapshot in time. Indeed, during the course of this project, new versions of the same NAT product exhibited different behavior from previous versions. Moving forwards, we hope that our TCP NAT traversal test suite can continue to be used to broaden our knowledge of NAT and firewall characteristics as well as to track trends in NAT products and their deployment.

Acknowledgments

The authors wish to thank Bryan Ford, Andrew Biggadike, and the anonymous IMC reviewers for valuable feedback on early drafts of this paper. We also wish to thank Synergy Research Group, Inc., for providing up-to-date market survey data. Finally, we wish to thank the many volunteers who ran the STUNT test client and the peer-to-peer TCP tests on their systems and submitted the results.

References

[1] AUDET, F., AND JENNINGS, C. Internet draft: Nat behavioral requirements for unicast udp, July 2005. Work in progress.

[2] BASET, S. A., AND SCHULZRINNE, H. An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol. Tech. Rep. CUCS-039-04, Columbia University, New York, NY, Sept. 2004.

[3] BIGGADIKE, A., FERULLO, D., WILSON, G., AND PERRIG, A. NATBLASTER: Establishing TCP connections between hosts behind NATs. In *Proceedings of ACM SIGCOMM ASIA Workshop* (Beijing, China, Apr. 2005).

[4] BRADEN, R. RFC 1122: Requirements for Internet Hosts – Communication Layers, Oct. 1989.

[5] EPPINGER, J. L. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. Tech. Rep. CMU-ISRI-05-104, Carnegie Mellon University, Pittsburgh, PA, Jan. 2005.

[6] FORD, B., SRISURESH, P., AND KEGEL, D. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Annual Technical Conference* (Anaheim, CA, Apr. 2005).

[7] GUHA, S. STUNT - Simple Traversal of UDP Through NATs and TCP too. Work in progress. [online] `http://nutss.gforge.cis.cornell.edu/pub/draft-guha-STUNT-00.txt`.

[8] GUHA, S. STUNT Test Results. [online] `http://nutss.gforge.cis.cornell.edu/stunt-results.php`.

[9] GUHA, S., TAKEDA, Y., AND FRANCIS, P. NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity. In *Proceedings of SIGCOMM'04 Workshops* (Portland, OR, Aug. 2004), pp. 43–48.

[10] HUITEMA, C. Internet draft: Teredo: Tunneling ipv6 over udp through nats, Apr. 2005. Work in progress.

[11] JENNINGS, C. Internet draft: Nat classifi cation test results, Feb. 2005. Work in progress.

[12] MICROSOFT CORPORATION. UPnP – Universal Plug and Play Internet Gateway Device v1.01, Nov. 2001.

[13] POSTEL, J. RFC 761: DoD standard Transmission Control Protocol, Jan. 1980.

[14] ROSENBERG, J., MAHY, R., AND HUITEMA, C. Internet draft: TURN – Traversal Using Relay NAT, Feb. 2004. Work in progress.

[15] ROSENBERG, J., WEINBERGER, J., HUITEMA, C., AND MAHY, R. RFC 3489: STUN – Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs), Mar. 2003.

[16] STIEMERLING, M., QUITTEK, J., AND TAYLOR, T. MIDCOM Protocol Semantics, June 2004. Work in progress.

[17] TAKEDA, Y. Internet draft: Symmetric NAT Traversal using STUN, June 2003. Work in progress.

[18] VANCE, A., Ed. *Q1 2005 Worldwide WLAN Market Shares*. Synergy Research Group, Inc., Scottsdale, AZ, May 2005, p. 40.

Notes

[1] Network behind a NAT or fi rewall

[2] Throughout this paper, the term NAT is understood to include fi rewalls.

[3] sometimes referred to as *dual* or *double NAT*

[4] IP time-to-live fi eld

[5] Maximum Segment Length

[6] Round-trip time