TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem

Jeffrey L. Eppinger
*Carnegie Mellon University*

Follow this and additional works at: [http://repository.cmu.edu/isr](http://repository.cmu.edu/isr)

This Article is brought to you for free and open access by the School of Computer Science at Research Showcase @ CMU. It has been accepted for inclusion in Institute for Software Research by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.
TCP Connections for P2P Apps:
A Software Approach to Solving the NAT Problem

Jeffrey L. Eppinger
January 2005
CMU-ISRI-05-104

Institute for Software Research International
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract
Many P2P applications need to connect to each other via TCP, but are increasingly stymied by NAT boxes. Some popular P2P applications do not address NAT traversal or do so poorly. A few newer ones route communications between NATed peers through relay servers or through non-NATed peers, or they ask users to reconfigure their NAT boxes. Some emerging solutions suggest using SIP to set up tunneling over UDP, using UPnP, or even deploying IPv6. This paper argues that the above approaches suffer from scalability problems, do not address mobility issues, require deploying new network infrastructure, or require using non-standard communications interfaces, non-standard communication stacks, and non-standard security protocols. We advocate direct TCP connections between peers. We present NatTrav, our NAT Traversal Java package that makes TCP connections between NATed peers and addresses all of the above concerns. We then compare NatTrav with some of the other current and emerging solutions.
Keywords: P2P, peer-to-peer, NAT Traversal, TCP, network address translation.
1. The Problem

We are building some really cool peer-to-peer (P2P) applications. If you’re reading this paper, we’ll bet you are, too. Among the first problems we encountered was how to connect our P2P applications together. We want TCP connections between instances of our application running on mobile (or even desktop) machines that (i) connect to the Internet from various places so they don’t have a fixed network address, and (ii) often connect to the Internet via a Network Address Translation (NAT) box. We must solve problem (ii) in light of problem (i). Here’s a scenario: our applications run on notebook computers and PDAs that are sporadically connected to the Internet from a home network, the office, “free high-speed Internet” in a hotel room, or WiFi in a coffee shop. In most if not all of these scenarios, our application requires TCP connections between two machines that are both behind NAT boxes. The problem is that computers attached to the Internet via NAT boxes can make outbound connections to non-NATed computers, but typically cannot receive inbound connections.

NAT boxes allow several computers to share one public IP address [1]. The computers get private IP addresses and must send their communication to the public Internet through the NAT box. The NAT box uses port translation in order to determine which computer should receive the responses [2]. For example, when the NAT box receives communication from a private IP address and port, say A:X, it sends the communication out via its public IP address on a proxy port, say NA:Y. The NAT maintains a table of mappings, the NAT uses NA:Y for all future communications from A:X until the mapping is removed from its internal tables. Mappings are only removed after a period of inactivity on NA:Y.

2. Current Approaches

Popular P2P applications have addressed the NAT Traversal Problem in different ways. Below we enumerate our concerns about how these applications address NAT Traversal:

1) If only one peer is NATed, have the NATed peer set up the connections (e.g., Kazaa [4], LimeWire [5]). This is only a partial solution: in a majority of the scenarios in which our applications run both peers will be NATed.

2) Recommend users configure their NAT boxes to forward incoming requests on specific ports to one computer behind the NAT (e.g., BitTorrent [6]). This is unacceptable because (a) several computers behind the NAT may need to run this application and (b) users may not have the technical capability or administrative access to configure the NAT.

3) Route P2P communication through central relay servers (e.g., Groove [7]). This approach is too expensive. As our application scales, we cannot afford to provide the CPU and network bandwidth.

4) Route TCP connections through instances of the application that happen to be running on open, non-NATed computers (e.g., Skype [8]). We believe users who happen to be running our applications on non-NATed machines will not want to bare the CPU and network costs either. Moreover most machines running our applications will be NATed so there may be shortage of non-NATed machines to help out.

5) Use a proprietary session initiation protocol to exchange UDP ports between the peers and then tun-
nel TCP over UDP (e.g., Newrong [9]). We want to use a standard TCP stack with standard TCP stacks and interfaces so that we can capitalize on the benefits of the years of work optimizing TCP and building application packages for it. Also many networks do not allow UDP.

Finally, we have a security concern: our application requires the peers authenticate each other. We want to use SSL. Approaches 3, 4, & 5 require some sort of custom-made authentication scheme.

In Section 7, we discuss emerging solutions including UPnP, IPv6, and SIP.

3. Solution Requirements

Summarizing our concerns described above, our NAT Traversal solution should permit applications to talk to each other with:

a) Scalability – We want to incur minimal (if not zero) cost as the number of instances of our applications grows.

b) Standard Interfaces – For ease of development, we want to use standard interfaces for network communication, e.g., Sockets. This is more familiar to developers and allows many existing software packages/libraries to be used without modification.

c) Standard Stacks – Many years and tears have gone into optimizing TCP. We don’t want to reinvent and/or debug this. Furthermore, many networks do not allow the use of UDP.

d) Security – We want to use SSL with standard interfaces and X.509 certificates. We don’t want to have to re-invent this complex wheel, either.

e) Mobility – We want peers to communicate with each other even though they move from network to network. Often these networks are not controlled by the user who cannot reconfigure the NAT to add port mappings, allow UDP, or turn on UPnP.

4. Our Approach

Our approach is called NatTrav. In NatTrav, “recipient” peers willing to receive connections from “initiator” peers register with an intermediate connection broker by providing a network address and a URI – a Universal Resource Indicator to identify uniquely the recipient peer.

Connection brokers facilitate connections to recipients by (i) providing the current network address for the recipient and (ii) facilitating NAT Traversal if the recipient is NATed. Connection brokers are replicated for availability and scalability. See Figure 1.
Look up – In Step 3, when an initiator peer wants to make a TCP connection to a recipient peer, it sends a Lookup Request to a connection broker. The connection broker eventually replies (Step 10) with a public IP address and port that the initiator can use (in Step 11) for a direct TCP connection from initiator to recipient (via their NAT boxes) as shown in Figure 1.

The connection broker checks to see if there is an active registration for the recipient. The connection broker may need to contact other connection brokers to determine this. If so, it sends a Connection Request to the recipient on the UDP address and port the recipient (or its proxy) used to register (Step 4). The Connection Request contains a correlator (corr) to track which Lookup Request subsequent steps are working on and the IP address and port of the connection broker that the recipient should use for the subsequent steps.

Punching the Hole – In Steps 5 & 6 the recipient uses TCP to exchange network address information with the connection broker specified in the Connection Request. The connection broker compares the recipient’s private IP address (R) and its public IP address (NR). If they match, the recipient is not NATed. If they do not match, the recipient is NATed and the recipient is told to punch a hole in its NAT for messages coming from the initiator’s public IP address and port (NI : Y).

Note: The IP addresses and ports used for communication are shown in square brackets. For example: [I: X > NI: Y > B: Z] denotes communication from the initiator’s IP address I and port X to the connection broker’s IP address B on port Z via NAT_I’s public address NI and public port Y.
The trick here is that we use the same IP addresses and ports for TCP communication with the connection broker and for the direct TCP connection between initiator and recipient. Because we are punching holes in cone-type NATs, we know that the NATs will use the same proxy ports for communication with the connection broker and the peers. In the example shown in Figure 2, the initiator uses port X when communicating with the connection broker (Step 3), but the connection broker sees it coming from NATI’s port Y. Similarly, the recipient uses port J (Step 5) and the connection broker sees NATR’s port K. To punch the hole, the recipient closes its TCP connection with the connection broker used in Steps 5 & 6 and in Step 7 uses port J to attempt to open a TCP connection to NI:Y. Depending on the type and implementation of NATI, Step 8 may be a timeout, a TCP reset response, or (if NATI is a full-cone NAT) a port-in-use error (because NATI forwarded the TCP SYN on to I:X). In any case, the TCP connection will not succeed but the attempt enables NATR to forward communication from NI:Y sent to NR:K on to R:J.

Between Steps 8 & 9, we set up a ServerSocket to listen on R:J. Then we send a Connection Response message to the connection broker (Step 9). In Step 10, the connection broker tells the initiator that it can now set up a direct TCP connection to the recipient (via the NAT boxes) by using NR:K. In Step 11, the TCP connection is set up.

Our P2P applications use a Java package called nattrav. The nattrav package contains classes that parallel the style of the standard Java class libraries for setting up TCP Sockets between applications. Not only this, but the objects returned by the nattrav package are instances of the standard Java Socket class directly connecting the initiator and recipient peers. Figures 3 & 4 are code excerpts showing the interfaces for the most basic methods.

5. Security Extensions

In many cases, we want our P2P applications to use SSL connections between the peers. This is for privacy of the communication and to authenticate the peers. We use our own certification authority (CA) to sign X.509 certificates for connection brokers and the peers. The CA’s signing certificate is pre-installed into our applications’ key stores. We can then use the signed X.509 certificates to make SSL connections in Figure 2, Step 11.

The interesting security problem for our NAT Traversal Protocol as shown in Figure 2 is that nothing prevents a rogue peer from registering as a recipient thereby redirecting Connection Requests to itself. To guard against this, we also have developed a secure registration protocol that uses the X.509 certificates (and the corresponding private keys) to imple-
ment a simple challenge/response to authenticate
connection brokers and recipients during registration.

6. Evaluation
To date, we have built two P2P applications using
NatTrav: (1) a Remote Desktop Proxy that allows
Remote Desktop Connections to Windows machines
located by URI, and (2) a replicated, incremental
backup service. At the time of this writing, these
applications are in limited deployment.

In these deployments, the observed elapsed time
to setup a TCP connection from an initiator to a re-
cipient using NatTrav ranges from 1.1 to 21.2 sec-
onds depending on many factors, including the speed
of the network between the computers and the behav-
or of NAT in Step 8 of the protocol. The initial con-
nexion time was no problem for the more often used
backup application as the replication occurs in the
background.

Upon further investigation, we added a configur-
able timeout when setting up the TCP connection in
Step 7. We currently run with this timeout set to 2
seconds and have now seen connection times under 5
seconds in all cases observed.

We note that in both applications, the ability to
have direct TCP connections between the peers is
very important. We are using standard Java Socket
(and SSL) classes for easy development and efficient
network performance.

A test of a connection broker running on an 800
MHz laptop driven by simulated client loads proc-
essed over 5,000 UDP Registrations (Steps 1 & 2)
per second. UDP recipients re-register every minute,
so a connection broker should support about 300,000
recipients. This does not measure the cost of replica-
tion or occasional Lookup Requests and Connection
Requests. We believe these costs to be minimal.
Since connection brokers provide an application spe-
cific service, running connection brokers to support
instances of our applications is not a significant cost.

7. Other Emerging Solutions
UPnP is a new standard to support the ability to plug
devices into a home network [11]. An application
could use UPnP to configure a NAT box to forward
to it incoming requests received on specific ports
(thereby achieving NAT Traversal). Unfortunately,
early UPnP implementations had highly publicized
security issues and many NAT boxes that support
UPnP ship with it turned off by default. Furthermore,
UPnP enables any application on the LAN to create
port mappings. Many savvy consumers continue to
have security concerns. Anecdotally, we have found
that WiFi hotspots tend not to enable UPnP. Users
should not have to reconfigure their NAT boxes to
turn on UPnP as they may not be technically capable,
might not have administrative access, or their NAT
might not support UPnP.

Nevertheless, UPnP support can be easily added
to NatTrav. When available, recipients would use
UPnP to configure port mappings on the NAT. The
recipient would register the NAT’s public IP address
and a mapped port with a connection broker. Lookup
Requests would still be used to find these recipients.

IPv6 is a new standard that specifies 128 bit IP
addresses [12]. It is possible that the new, longer
IPv6 addresses will reduce the need for NAT boxes;
however we are skeptical. NAT boxes abound as
IPv6 adoption has been slow and there is an expecta-
tion that protocol translating NAT boxes (NAT-PT)
will be used to bridge between IPv4 and IPv6 net-
works. Other projects such as [13] are working on
general solutions for NAT Traversal without any ap-
lication changes, but they require deployment of
significant additional network infrastructure.

SIP is a new IETF Session Initiation Protocol
that can be used to initiate UDP connections for tele-
phony and multimedia applications [14]. Except for
security and NAT Traversal, it standardizes the
communication formats for functions similar to our
connection brokers. We chose not to use SIP be-
cause: (i) it has a tremendous amount of mechanism
we do not need for our application, (ii) it does not
provide NAT Traversal, and (iii) beyond standardiz-
ing how user ids and passwords are sent, it does not
address our security concerns. Even if we were to
include all the above capabilities in a SIP implemen-
tation, whether or not we use SIP internally in our
applications is transparent to our users.

NUTSS is a new research project at Cornell [15].
As its acronym suggests, it proposes to combine NAT
Traversal, URIs, Tunnelling over UDP, SIP, and
STUN. STUN [3] is a UDP-based Protocol that an
application can use to talk with intermediate STUN
servers to determine if it is NATed and if so, via what
type of NAT. Again, we have several concerns about
tunneling over UDP. Recently, the NUTSS research-
ers have proposed a TCP based approach [16] con-
sisting of (i) STUNT: TCP extensions to STUN, and
(ii) a TCP hole punching protocol. Their hole punch-
ing protocol is much more complex than ours requir-
ing use of port prediction, STUNT, spoofing, and
sending TCP SYNs with low TTls. However, they are attempting to traverse symmetric NATs.

8. Conclusions

Our P2P applications require direct TCP connections between peers. By using a NAT Traversal solution, such as NatTrav, we are able to directly connect our peers across cone-type NATs: (i) using standard TCP Socket and SSL packages, (ii) with minimal cost per new peer, (iii) without routing through non-NATed peers, (iv) without requiring users to reconfigure their NATs, and (v) locating specific peers wherever they are connected to the Internet.

While we could not afford to wait for prevalent UPnP utilization and IPv6 deployment (which may never happen) or the future standardization activities initiated by NUTSS, we look forward to their future successes. Should support for symmetric NATs become important to our applications, NUTSS may be very important to us. As NAT box makers start supporting UPnP and enabling it by default, it may be useful to add UPnP support to our arsenal.

A final note: our NAT Traversal solution is application specific. So is Skype’s approach of routing calls through non-NATed peers. Their approach works well because it’s a low bandwidth application (less than 16 kbps during a call with custom encryption) [17]. But they likely will need to use direct TCP connections when they add video.

9. Acknowledgements

Special thanks to Mukund Gopalan and Kamal Sau-rabh who, with the author, created algorithms and their implementation for NAT Traversal described herein.

References