NAT UDP HOLE Punching

Group 6: Weiting Wu, Clement Fang, David Zhu
Introduction:

Objective: To implement and verify a UDP-based hole puncher for use in applications such as P2P and others that require direct connections across NATs.

Due to the large growth of the internet and increasing security challenges, the de facto Internet architecture has evolved away from its original uniform address architecture; where every node has a globally unique IP address and has the ability to communicate directly with any other node. This newfangled architecture consists of the original global address realm existing in conjunction with private address realms; both interconnected by Network Address Translators (NATs). The root of the problem is caused by this architecture because only nodes sitting in the global address realm can readily contacted by anyone else in the network, since they all have globally unique IP addresses. So what about these private network nodes, what are their capabilities? These nodes generally can establish direct (outgoing/incoming) connections to other nodes on the same private network, and in like manner can communicate can open TCP or UDP connections to nodes on the global realm. NATs along the routed path will allocate a temporary public endpoint for outgoing connections and in most cases block incoming traffic unless otherwise specified. This is fine in a client-server model where the client (sitting behind a NAT) communicates with a global entity. The issue arises in applications that require direct communication between two private entities sitting behind NATs; to simply name a few, teleconferencing, online gaming among various others.

NATs operate at the network layer of OSI protocol (or the Internet Layer for TCP/IP), a majority of this computer networks class according the syllabus, and is used extensively in the internet. NAT traversal refers the the group or set of techniques used to establish and maintain Internet protocol connections. NAT traversal has many methods, among the simplest is to use a ‘relay’ server which uses two client/server connections between each node/peer and the server; however, this method adds bandwidth costs and latency which is detrimental to any type of real-time or high bandwidth application. Hole punching, the objective of our project, on the other hand only requires an intermediate server to establish the initial direct connection. Thus the disadvantages of the ‘relay’ server technique are no longer present when using hole punching instead.

Theoretical bases and literature review:

The basic problem is to overcome NAT’s characteristic of only allowing outbound connections and establish p2p communication between nodes that are distributed behind different NAT networks.

NAT, known as the Network Address Translation, is a network protocol that provides translations between private hosts and public IPs. Majority of the NATs used in today’s technology configure a subnet of private IP addresses that is only known within the local network. These private hosts are grouped together with a public IP address assigned to its router by the Internet Service Provider. An active connection between two hosts across different NATs is
made possible by transmitting and using IP address and port number as the tracking data. As a request comes into NAT, it translates requested IP address and looks up the known (IP address, host) pairs to determine the private address on the internal network to which to forward the reply.

The NAT design has a very important limitation to its network communication. Before a connection is established, a packet does not know the destination of the private host behind a NAT, so it can only be sent from the source host but not received by the destination host. NAT has the rule of only allowing outbound traffic but disallows inbound traffic. All inbound packets are dropped. When it comes down to establishing a Point to Point connection between two hosts behind NATs, the challenge is to initiate a communication with the constraint of only allowing outbound traffic from the private hosts behind NAT.

Two better-known solutions attempted in the past are Relaying and Connection Reversal. Relaying is the most reliable but least efficient method of P2P connection across NAT. It involves the concept of a shared server S that acts a middle man between two client hosts residing behind NATs. Client hosts A and B do not directly communicate with each other, instead, it goes through the third party server S and have it relay each other’s packets. Client A and B do not establish a direct connection during the duration of the communication; all messages are forwarded between the two clients by the existing client/server connection established by server S.

Another option is the Connection Reversal. This is used under the condition that only one host is behind a NAT and the other one is not. Given that both client A and B are listening on a rendezvous server S, which contains all publicly well-known IP addresses. If client A of a private network attempts to talk to client B which is on its own, a direct connection is automatically created because B is not behind a NAT. Vise versa, when B attempts to communicate with A, A’s NAT will block that incoming request. In this situation, B can use Rendezvous server S to relay a connection request, prompting A to initiate a connection out to B, therefore starting communication.

The Relaying method can always work as long as two clients are connected to the server; however, it’s very inefficient due to the latency through server S, extra server processing power and network bandwidth. Connection Reversal is a more limited option in practical use because end nodes almost always reside behind NATs - effectively creating a scenario of two private nodes, which renders connection reversal useless.

The central idea of using a well-known rendezvous server from Connection Reversal technique as the connection helper is fundamental to the solution described next - Hole Punching. Hole punching is a technique that involves the exchange of address info between source and destination hosts, as well as a step of tricking both ends to wait for responses given the initial packet relay between two hosts. Rendezvous server S contains a table of globally known public-private IP addresses pairs that identifies hosts spanning across all hosts.

The entire hole punching process can be broken down into 3 steps:
1. Client A wants to reach to Client B, A does not know B’s address and asks server S to help with connection establishment. The key here is that the server S will maintain an IP address directory.
2. S exchanges address info between A and B. It sends A B’s private-public (IP, port) pair, and sends B A’s private-public (IP, port) pair. Now both ends know each other’s targets.
3. Now A and B send each other packets, take into account that time is not important as long as the activities are asynchronous.
   a. if A and B are behind the same NAT - direct communication is established with private (IP, port) pairs.
   b. if A and B are behind different NAT - first round of delivered packets are dropped since no inbound traffic is allowed. Now that both A and B have sent their initial “requests”, they are also waiting for replies back. Therefore, any subsequent delivery of packets are now accepted by the other ends.

Unlike Connection Reversal, Hole Punching utilizes Rendezvous server as a two-way exchange medium info instead of just one-side information relay medium. Also, it utilizes the fact that each private host can switch between its role as a sender and a receiver, tricking both hosts to follow the send-then-receive convention. The highlight of Hole Punching is that it does not require additional server for message relay and also sharing of (IP, Host) address pair makes the communication to be exact.

**Hypothesis/Goals:**

Our goal as stated above is to establish a direct connection between two different peers behind private NATs with the help of an intermediate server only during the connection setup process. There are two different types of connections TCP/UDP and our goal is to investigate UDP hole punching. Despite the advantages of hole punching, there may exist some NAT configurations that will become problematic for this technique and cause it to fail. If time and simulation environment permits this, it may be investigated. Therefore the more inefficient method of relaying may be a good fallback method if all else fails.

**Methodology:**

The technique to solve the NAT traversal problem will be using UDP hole punching.

This section covers the steps a client and server must follow to use the rendezvous. Effectively, the intermediary server used as a middleman for IP address 4-tuple transferal can be viewed as a proxy for the two end nodes residing on private networks. This proxy server, or rendezvous point, needs to maintain an IP address directory. To implement a P2P communication, the end client nodes will need to have a pre-define API interface with the proxy server that provides a service to publish IP addresses and ports to which a client can attempt to connect. If such connections are blocked by firewalls, the proxy server in turn will be used as a relay (which is inefficient as discussed above). Onto the specificities of how to create hole-punching:
The server code will need to have a struct defined to hold the client’s information; IP address and port. For the purpose of this simulation, we will let the client and server be allowed to have an array of up to 2 clients. In real life, the server should be able to handle many clients. For the server, we will first create a UDP socket. Once we make a connection, we will store all the connection information. Note that the server cannot be behind a NAT. After that, the server will continuously listen for incoming requests from potential clients. When an incoming datagram arrives, the server will record the client’s IP and port into its IP address directory. Once we’ve added the client’s UDP endpoints, the server will broadcast the client list to each of its clients.

The client code will contain a similar struct defined to hold other client’s information; IP address and port. After performing the same UDP port setup as the server, the client will send a datagram to the server to declare its client information. Other clients will use this address broadcasted by server to send subsequent packets. After the initial communication packet is sent to server, server responds back with all available registered clients address information. Then, the client will attempt to send 3 packets to each of the clients. At this point, the two client’s NATs have already established a credible tunnel. This means the hole has been punched, and subsequent communication can be continued directly between the clients - without the help of the server to act as relay.

Language used is C for both server and client simulation code. Tools used are VirtualBox, Open WRT, and Ubuntu 12.04.

Possible Errors:
// It is possible to get data from an unregistered peer. These are some reasons
// I quickly came up with, in which this can happen:
// 1. The server's datagram notifying us with the peer's address got lost,
// or it hasn't arrived yet (likely)
// 2. A malicious (or clueless) user is sending you data on this endpoint (maybe
// unlikely)
// 3. The peer's public endpoint changed either because the session timed out,
// or because its NAT did not assign the same public endpoint when sending
// datagrams to different remote endpoints. If this happens, and we're able
// to detect this situation, we could change our peer's endpoint data to
// the correct one. If we manage to pull this off correctly, even if at most
// one client has a NAT that doesn't support hole punching, we can
communicate
// directly between both clients.

Implementation:

For source code please refer to our submission file client.c and server.c; along with our Makefile.
This figure shows our test setup. We used 6 total virtual machines in total--3 running ubuntu and 3 running openWRT, an embedded router OS. The two VMs below the dark green line, Client 1 and 2, are behind a NAT. The global realm is indicated by the inet0 VirtualBox internal network with the top openWRT router acting as a DHCP handing out IP addresses in the global realm. In this case that is the Ubuntu VM running the server and the two routers that act as different NATs on top of each client. Client1 is connected to its router on inet1 and Client2 is connected to its router inet2, as shown by the legend they are on two different IP address spaces and completely private.

Results and Discussion:

As our hypothesis is a proof of concept, our output consists of status outputs from our server and our client. We verify our hypothesis by analyzing this output. One example output from our server after our two clients attempts to initiate a connection via hole punching is the following:

Sending to 192.168.0.113:41288
Sending to 192.168.0.185:36304

This output indicates the two public client IP/Port pairs that are stored in the server and this info is sent to each client. To verify that we indeed have a port restricted NAT behavior we tear down this connection and attempt our client connections again.

Sending to 192.168.0.113:44801
Sending to 192.168.0.185:55227

We can clearly see the port mappings change i.e. a private IP/Port pair behind the NAT is not always mapped to the same external IP/port pair. This is in essence what causes the hardships for peer two peer connections when there are NATs between them.

On our client side we clearly see as demonstrated in our final project defense that we are directly receive packets from the external IP/port of the other client. This proves that we successfully hole punched since two clients behind a port restricted NAT are able to send
directly to each other without using the assistance of the server after the hole is punched. In our while loop as mentioned earlier we print the IP address and port pair that we receive from; so we can clearly see the difference in IP.

The results gained from this proof of concept implementation clearly shows the difficulty communicating with entities behind NATs, from the fact the the external IP/port pair mappings keep changing and our successful send and receive of UDP datagrams on each client shows the viability of hole punching as an effective method for establishing client-client connections. In some cases we noted that after a period of time the connection will no longer work, this is due to a timeout somewhere that automatically removes the filter rule for the previously established UDP hole punch connection. In which case the external IP/port pair is no longer valid.

**Conclusions and Future Work:**

We were able to prove that the hole punching concept is an effective method for establishing client to client connections even when they are behind two different port-restrictive NATs. We also showed that our test setup was indeed valid and thus so are our results. For future work a good extension is to replace the openWRT virtual box routers with actual Linux VMs configured, using IPTABLES, to act like various NATs. Since we were only able to test one type of NAT behavior this is not a conclusive proof of concept that UDP hole punching can work in any situation; but simply a proof that it will work in a relatively p2p unfriendly NAT. Also we can enhance our existing client program by sending keep alive UDP datagram packets so we will not see the timeout behavior anymore. Finally a natural extension of this work is to investigate and implement TCP hole punching.

**Bibliography:**


Ford B. ; Srisuresh P. ; Kegel, D , ATEC ’05 Proceedings of the annual conference on USENIX Annual Technical Conference : Peer-to-peer communication across network address translators: Publication Year: 2005

Seah D. ; Leong W.K. ; Yang Q. ; Leong B. ; Razeen A. ; Peer NAT Proxies for Peer-to-Peer Games

Venkatachalam G.; Linux Jounral: Developing P2P protocols across NAT Publication Year: 2006 Page 8