Developing P2P Protocols across NAT

Girish Venkatachalam

Abstract

Hole punching is a possible solution to solving the NAT problem for P2P protocols.

Network address translators (NATs) are something every software engineer has heard of, not to mention networking professionals. NAT has become as ubiquitous as the Cisco router in networking terms.

Fundamentally, a NAT device allows multiple machines to communicate with the Internet using a single globally unique IP address, effectively solving the scarce IPv4 address space problem. Though not a long-term solution, as originally envisaged in 1994, for better or worse, NAT technology is here to stay, even when IPv6 addresses become common. This is partly because IPv6 has to coexist with IPv4, and one of the ways to achieve that is by using NAT technology.

This article is not so much a description of how a NAT works. There already is an excellent article on this subject by Geoff Huston (see the on-line Resources). It is quite comprehensive, though plenty of other resources are available on the Internet as well.

This article discusses a possible solution to solving the NAT problem for P2P protocols.

What Is Wrong with NAT?

NAT breaks the Internet more than it makes it. I may sound harsh here, but ask any peer-to-peer application developer, especially the VoIP folks, and they will tell you why.

For instance, you never can do Web hosting behind a NAT device. At least, not without sufficient tweaking. Not only that, you cannot run any service such as FTP or rsync or any public service through a NAT device. This can be solved by obtaining a globally unique IP address and configuring the NAT device to bypass traffic originating from that particular IP.

But, the particularly hairy issue with NATed IP addresses is that you can't access machines behind a NAT, simply because you won't even know that a NAT exists in between. By and large, NAT is designed to be transparent, and it remains so. Even if you know there is a NAT device, NAT will let traffic reach the appropriate private IP only if there is mapping between the private IP/TCP or UDP port number with the NAT's public IP/TCP or UDP port number. And, this mapping is created only when traffic originates from the private IP to the Internet—not vice versa.

To make things more complicated, NAT simply drops all unsolicited traffic coming from the Internet to the private hosts. Though this feature arguably adds a certain degree of security through obscurity, it creates more problems than it solves, at least from the perspective of the future of the Internet.

At least 50% of the most commonly used networking applications use peer-to-peer technology. Common examples include instant messaging protocols, VoIP applications, such as Skype, and the BitTorrent download accelerator. In fact, peer-to-peer traffic is only going to increase as time progresses, because the Internet has a lot more to offer beyond the traditional client/server paradigm.

Peer-to-peer technology, by definition, is a mesh network as opposed to a star network in a client/server model. In a peer-to-peer network, all nodes act simultaneously as client and server. This already leads to programming complexity, and peer-to-peer nodes also have to deal somehow with the problematic NAT devices in between.

To make things even more difficult for P2P application developers, there is no standardized NAT behavior. Different NAT devices behave differently. But, the silver lining is that a large portion of the NAT devices in existence today still behave sensibly enough at least to let peer-to-peer UDP traffic pass through.

Sending TCP traffic across a NAT device also has met with success, though you may not be as lucky as with UDP. In this article, we focus purely on UDP, because TCP NAT traversal still remains rather tricky. UDP NAT traversal also is not completely reliable across all NAT devices, but things are very encouraging now and will continue to get better as NAT vendors wake up to the need for supporting P2P protocols.

Incidentally, voice traffic is better handled by UDP, so that suits us fine. Now that we have a fairly good idea of the problem we are trying to solve, let's get down to the solution.

Anatomy of the Solution

The key to the NAT puzzle lies in the fact that in order for machines behind a NAT gateway to interact with the public Internet, NAT devices necessarily have to allow inbound traffic—that is, replies to requests originating from behind the NAT device. In other words, NAT devices let traffic through to a particular host behind a NAT device, provided the traffic is indeed a reply to a request sent by the NAT device. Now, as mentioned above, NAT devices vary widely in operation, and they let through replies coming from other hosts and port numbers, depending on their own notion of what a reply means.

Our job is simple if we understand this much—that instead of connecting directly to the host behind NAT, we somehow need to mimic a scenario in which the target host originates a connection to us and then we connect to it as though we are responding to the request. In other words, our connection request to the target host should seem like a reply to the NAT device.

It turns out that this technique is easy to achieve using a method now widely known as UDP hole punching. Contrary to what the name suggests, this does not leave a gaping security hole or anything of the sort; it is simply a perfectly sensible and effective way to solve the NAT problem for peer-to-peer protocols.

In a nutshell, what UDP hole punching does already has been explained. Now if it were only that, life would be too simple, and you would not be reading this article. As it turns out, there are plenty of obstacles on the way, but none of them are too complicated.

First is the issue of how to get the private host to originate traffic so we can send our connection request to it masquerading as a reply. To make things worse, NAT devices also have an idle timer, typically of around 60 seconds, such that they stop waiting for replies once a request originates and no reply comes within 60 seconds. So, it is not enough that the private host originate traffic, but also we have to act fast—we have to send the "reply" before the NAT device removes the "association" with the private host, which will frustrate our connection attempt.

Now, a reply obviously has to come from the original machine to which the request was sent. This suits us fine if we are not behind another NAT device. So, if we want to talk to a private IP, we make the private IP send a packet to us, and we send our connection request as a reply to it. But, how do we inform the private IP to send a packet to us when we want to talk to it?

If both the peer-to-peer hosts are behind different NAT devices, is it possible at all to communicate with each other? Fortunately, it is possible.

It turns out that NAT devices are somewhat forgiving, and they differ in their levels of leniency when it comes to interpreting what they consider as reply to a request. There are different varieties of NAT behavior:

1. Full cone NAT
2. Restricted cone NAT
3. Restricted port NAT
4. Symmetric NAT

I won't go into the details and definitions of these here, as there are numerous resources explaining them elsewhere. Symmetric NATs are the most formidable enemy for P2P applications. However, with a degree of cleverness, we can reasonably "guess" the symmetric NAT behavior and deal with it—well, not all symmetric NATs, but many of them can be tamed to allow P2P protocols.

First, how do we tell the private IP that we are interested in connecting to it at a particular instance?

Implementation Details of the UDP Hole Punching Technique

This problem can be solved by joining the problem, rather than fighting it head on. In order to achieve peer-to-peer traffic across NATs, we have to modify our P2P mesh model slightly to make it a hybrid of a traditional star model and modern mesh model.

So, we introduce the concept of a rendezvous server, or mediator server, which listens on a globally routable IP address. Almost all peer-to-peer protocols have traditionally relied on certain supernodes, or in other words, in P2P, all nodes are equal but some are more equal. Some nodes always have acted as key players in any P2P protocol. If you have heard of a BitTorrent tracker, you know what I mean.

A rendezvous concept is nothing new in the P2P world, nor is the star model totally done away with in P2P.

Coming back to our original NAT problem, private IPs obviously can browse the Internet through NAT devices, and thus they can talk HTTP through port 80 or through a proxy HTTP port over TCP. So private IPs can almost always open TCP connections to global IP addresses. We use this fact to make the private IP connect to a mediator or rendezvous server through TCP.

Our solution relies on the fact that all the P2P nodes are constantly in touch with a rendezvous server, listening on a global IP address through a persistent TCP connection. Remember that P2P nodes are both client and server at the same time, so they can originate connections as well as serve connection requests simultaneously.

It is through this TCP connection that we inform a particular P2P node that another node wants to talk to it. Then, the target node sends a request following which the peer sends the connection request as a response to the request.

Because the private machines behind a NAT device do not have a routable IP address, the only way for us to access them from outside the NAT device is through the mapping that the NAT device maintains for the machine to talk to the external world. For each connection originated from the private IP, a unique port is assigned at the NAT device. For us to talk to the private IP, we have to send our packets to that particular port assigned for the private IP's connection to the external world. Now, we know that there is no notion of connection in the UDP world, so NAT assumes that if a reply doesn't come for a UDP request in about 60 seconds, the connection is deemed non-existent and closed.

So now we have another problem—that of determining the port assigned at the NAT's public interface for the private IP connection. This can be inferred by inspecting the source address of the UDP datagram that reaches any global IP.

So far so good. If we are not behind NAT, we can use the previously mentioned technique to initiate communication with a private IP using the rendezvous server.

However, reality tells us that P2P peers are more likely to be behind a NAT than otherwise. So, this solution is not enough. We want to initiate a P2P connection from behind a NAT device ourselves. So, now we have two NAT devices in the picture, one behind each P2P node.

Now the real fun begins. First, let's redefine our goal in the light of this new twist to the problem and attack it step by step. What we want to do now is use the rendezvous server and inform the target P2P node to send us a request, but we are behind a NAT.

So, for any external party to talk to us, we should have a global IP/port combo that exists at the NAT public interface. First we have to create one for ourselves. Only then we can receive communication requests coming from outside the NAT network.

We can create a mapping for us by sending a packet to a global IP. The global IP can then figure out our mapping by inspecting the from address. But how do we inform our P2P node of this address? For that we can use the TCP connection with the rendezvous machine. But, only the global IP to which we send the packet knows our association, so how do we figure that out? It's simple. The global IP can send that information to us as a reply in the packet payload to us.

Assuming that we somehow obtain a public IP, port pair and figure that out, we tell the mediator that we are listening at that public IP/port pair and request the P2P target node to initiate a request to us. Subsequently, we can connect to it as a reply to that message.

But, then we cannot receive packets from the P2P target node, because NAT is not expecting a reply from that global IP. In fact, some NATs that show full cone behavior allow packets to come from any IP, but most NATs do not—back to square one.

Consider this: if both P2P nodes behind the NAT send packets to each other's public IP/port, the first packet from each party is discarded because it was unsolicited. But subsequent packets are let through because NAT thinks the packets are replies to our original request. And voilà the hole is punched, and UDP traffic can pass through directly between the P2P nodes.

Unfortunately, NATs also differ in their behavior of assigning public ports for different destination IPs. Most NAT devices fortunately do not change public ports between requests to different destination IPs, so we can safely assume that.

So first we send certain probe or discovery packets to two different IPs and figure out the behavior of the NAT. If it is found to be consistent, our approach will work. In the unlikely case that we bump into symmetric NAT behavior that varies the port between requests, we can figure out the delta by which the port number varies. And, using this we can guess the port assigned for a particular request.

The reason we are so particular about this is because the first packet to our P2P destination behind NAT is dropped by NAT. So, all we can do is guess. In practice, however, it works fairly well. This is why it is important that the P2P nodes keep the source and the destination ports the same for communication.

Once this hole punching procedure is performed, the two P2P nodes can communicate with each other without the help of the rendezvous machine. So the rendezvous machine is useful only for informing a P2P node about an incoming connection and informing each of the communicating peers about each other's public addresses. Subsequently, the communication happens directly without the intervention of the rendezvous server.

Now we have to apply some ingenuity and introduce appropriate headers in the packets to inform the peer whether it is sending a reply meant for the P2P client or whether it is sending a request meant for the P2P server. Once we are able to differentiate between the two, we are set. We also need to differentiate between hole punching traffic and regular traffic, because hole punching traffic needs to be bounced, and regular traffic needs to be processed.

Of course, if we stop sending and receiving, the association at the NAT device at both ends will expire. So we either can send keepalive traffic or rerun the hole punching technique. You can choose whichever technique is suitable depending upon your needs.

This technique will not work if both the P2P nodes are behind the same NAT device. So, we also have to figure out whether we can communicate directly using the private IP address itself. Thus, our hole punching has to try the private interface along with the peer's public interface. And, it can happen that our private network has the same private IP as the peer's private IP. So we have to guard against getting spurious responses.

It also can happen that another P2P node in the same private network as ours has the same private IP as the P2P node we want to talk to in another private network. Then we have to do additional validation against the peer's identity to make sure we really are talking to the interested node.

In the unlikely case that you run into brain-damaged NAT devices at both ends, this technique obviously will fail, because we should be able to predict the public address assigned to us. In that situation, the only way is to make the rendezvous server act as a relay for the traffic. So peer-to-peer traffic goes through, but it is no longer peer to peer with the rendezvous machine acting as server. If you run into such situations, you need to think of implementing that as well.

Now, for the Real Dope, the C Code for Achieving the above

Due to their long length, the listings for this article are located on the *Linux Journal* FTP site at ftp://ftp.ssc.com/pub/lj/listings/issue148/9004.tgz. I leave out unnecessary detail and glue code and focus purely on the nontrivial aspects of UDP hole punching.

If you need more information on implementing your own hole punching library, you always can refer to the above design constraints and design a solution appropriately.

Please note that I have consciously left out the rfcs and NAT discovery techniques, such as STUN and frameworks like ICE. UDP hole punching is already complicated, and we don't gain anything by making it even more bloated without adding any real value. So, the technique as it stands works as good or even better than other NAT traversal mechanisms.

First, take a look at the rendezvous code (Listing 1). Note that we use select() to serve multiple sockets. We could as well use kqueue() on *BSD, or better, use the libevent abstraction (see Resources). But, I stuck to select() because performance doesn't matter so much to us. We talk to the mediator server only for establishing peer-to-peer connections, not otherwise.

The hole punching implementation is given in Listing 2 and the P2P client in Listing 3.

Using this method, you should be able to develop your own peer-to-peer protocol. You easily can develop your own instant messaging protocol along with some GUI code. You can transfer files either using nc or using code for that directly. You can develop certain applications, such as transferring voice via a microphone and speaker. In other words, you can develop a hobby VoIP application with this.

Several possibilities exist. You can add some reliability on top of UDP in case you are paranoid about your data reaching you safely.

One very useful tool that helped me immensely in this endeavor is the Network Swiss-Army knife, netcat.

You can see hole punching in action by using this simple command. At each end, type:

```
$ nc -u -p 17000 <peer public IP> 17000
```

With only the peer public IP different, you can start communicating if you are lucky, because most NAT devices try to assign the same private port as the public port.

If you want to test TCP hole punching, try this:

```
$nc -l -p 17000
```

at one end and this:

```
$nc -p 17000 <peer public IP> 17000
```

at the other end.

Future Work

Rather than having one rendezvous server, you can have a few of them for failover and geographical distribution. However, if you are behind two levels of NAT, sometimes this may not work. You also could listen on multiple virtual and real interfaces and attempt hole punching on all of them. You can add TCP hole punching on similar lines and try that first, and then attempt UDP hole punching.

Resources for this article: http://www.linuxjournal.com/article/9072.