ICE and libnice

Connectivity despite NAT?

Philip Withnall

2014-10-16

What is the problem?

- IPv4 address exhaustion
- Network Address Tunnelling (NAT)
- Client-to-client connections are not end-to-end connected
- NAT can't demultiplex incoming packets to listeners

The end of the IPv4 world is coming (and has been forever, and probably will continue to do forever). There are not enough IPv4 addresses for every computer, phone, fridge, toaster, watch and pair of glasses to have one. That would be fine if those devices didn't need to communicate, but people quite like their glasses to do video calls with their fridge to see how much milk they have left; it's cool.

How have we coped with running out of IPv4 addresses? Predominantly by using the standard CS trick of multiplexing things. Network Address Tunnelling (NAT) is a way of multiplexing multiple IP addresses through a single one, typically implemented in a router, so all traffic from the devices behind the router (on the LAN side) appears to come from the router's single IPv4 address (on the WAN side).

This works pretty well for client–server connections, but falls down somewhat when two clients want to communicate directly, as in peer-to-peer systems like file sharing or VoIP. Neither client can address the other directly, as both are hidden behind NATs, so the addresses they know for each other actually map to multiple devices behind each NAT.

The main problem is that NAT cannot demultiplex incoming packets to *listeners* when those packets are the first in a connection. It can only do that when the first packet in a connection originates inside the NAT, as that way it has access to the (source IP, source port, destination IP, destination port) tuple which identifies the connection.

Port translation

There are many ways to implement port translation, the key operation of a NAT translator. The most common is some form of symmetric translation, where the NAT translator's source port is used to multiplex between the private IP addresses and ports; different NAT mappings are used for each (source IP,

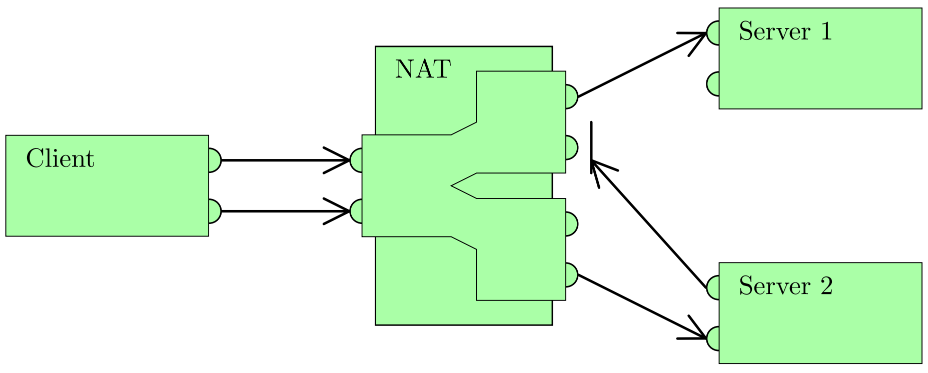

Figure 1: Symmetric NAT. CC-BY-SA 3.0, `http://en.wikipedia.org/wiki/File:Symmetric_NAT.svg`

source port, destination IP, destination port) tuple, even for packets to different destinations from the same (source IP, source port) pair. Only external hosts which have received a packet from a private host can send a packet back.

Other forms of port translation are less interesting and won't be covered here.

The last point is an important one: only external hosts which have received a packet from a private host can send one back. This clients behind a symmetric NAT from acting as servers and passively listening for incoming packets. Instead, they have to somehow actively send out packets in order to create a mapping in the NAT translator to handle a new incoming connection.

Even worse, if two clients which want to communicate peer-to-peer are both behind different symmetric NATs, there is no way they can communicate without relaying all their packets through a well-known server. We'll cover this later, but it's a bit of a pain, and not uncommon.

Network topologies

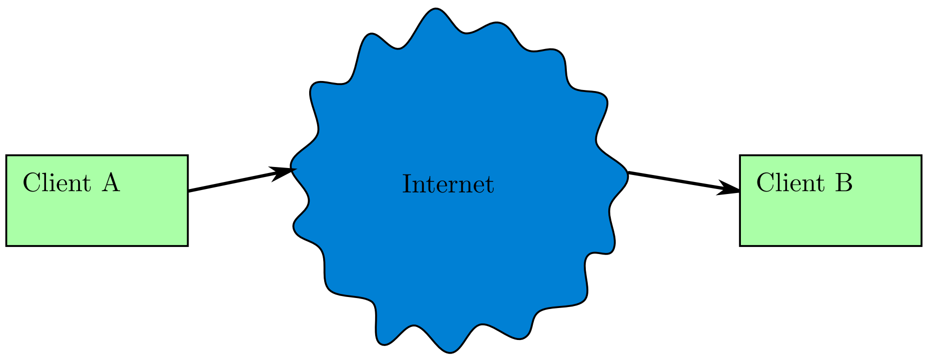

Figure 2: Two clients communicating without NATs.

The simplest network topology for two communicating clients is where no

NAT is involved at all. In this case, no NAT tunnelling is needed: the clients can communicate directly peer-to-peer. Great!

Network topologies

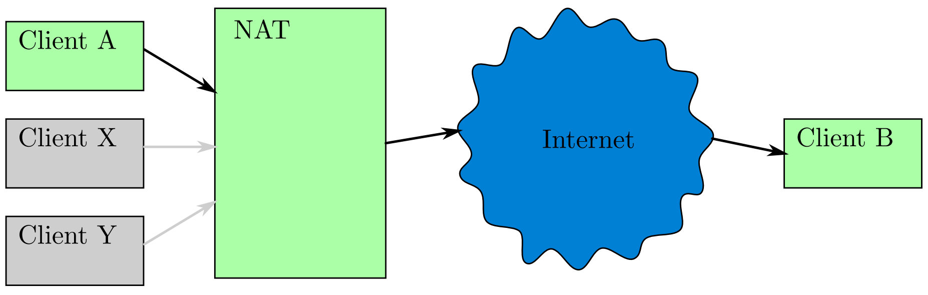

Figure 3: One client behind a NAT, the other not.

If client A moves behind a NAT, client B can no longer connect to it, as it can no longer address client A directly — it can only address the NAT. Client A can still connect directly to client B though.

If client A were to send a packet to client B, its NAT would create a new mapping, and client B could reply using the same ports. The reply would reach client A, and the two would then be able to continue the peer-to-peer conversation.

Network topologies

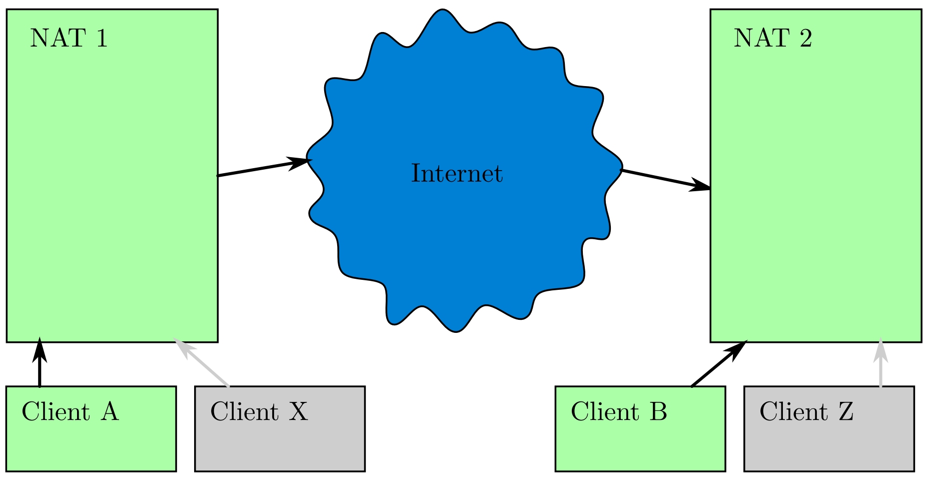

Figure 4: Both clients behind NATs.

With both clients behind NATs, neither client can connect to the other. Neither client can send an initial packet to the other either, so no communication is possible without relaying, covered later.

Network topologies

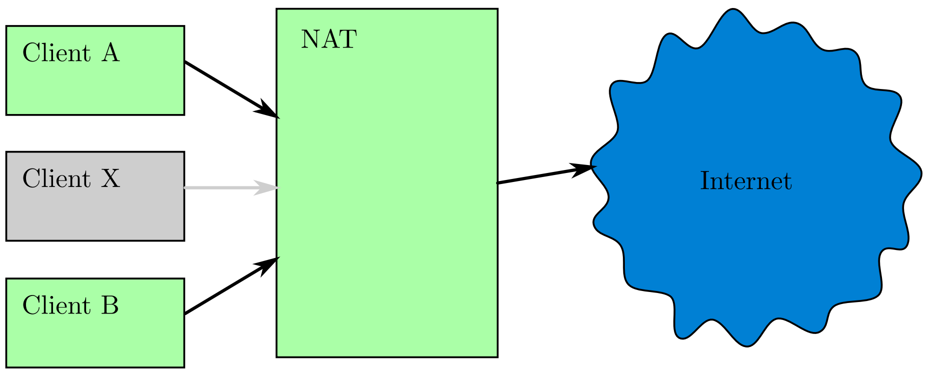

Figure 5: Both clients behind the same NAT.

This is a degenerate case of the previous one: both clients are behind a NAT, but they're behind the *same* NAT, so can actually address each other and communicate directly. However, this isn't possible if their retrieve each other's addresses from some central server (outside the NAT), since it will have the same address for both of them — the NAT's WAN address. The clients need to somehow establish their private addresses with each other.

What is the solution?

- Port forwarding
- Session Traversal Utilities for NAT (STUN)
- Traversal Using Relays around NAT (TURN)
- Interactive Connectivity Establishment (ICE)
- Hole punching

There are loads of solutions, but most of them only deal with one or two of the possible network topologies. The ones we care about most are STUN, TURN and ICE; together these deal with every possibility, and are the de-facto automated solution.

Port forwarding is where the NAT is configured to statically forward packets arriving at a specific WAN port to a specific private IP address and port. This works well in some situations (e.g. when hosting a server inside a NAT), but terribly for more dynamic situations such as torrenting or VoIP, where clients allocate arbitrary ports for communication.

Hole punching is where clients communicate out-of-band with the NAT translator to dynamically forward ports as the clients start listening on them. This works for the arbitrary port scenarios where static port forwarding doesn't, but only if the NAT translator supports it and allows it, and it doesn't work for nested NATs (not shown in the diagrams above, but you can imagine they're a nightmare).

So, onwards to STUN, TURN and ICE.

STUN

- RFC 5389 (for those who like that kind of thing)
- Client–server protocol
- Allows clients to find their WAN IP address
- Typically UDP based
- Has bells and whistles: authentication, encryption, message integrity, reliability, etc.

STUN is a component in the ICE protocol suite, and at its core is simply a way for a client behind a NAT to find out the WAN IP address of its NAT translator. It achieves this by sending a query to a well-known STUN server outside the NAT, and receiving a response containing the IP address and port the STUN server saw the query originate from.

Of course, like all these protocols, STUN provides a lot more. It's based on UDP, so it has to add its own reliability, message integrity, authentication, encryption, packet format, and other mundanities. The packet format it defines is re-used in ICE, covered later.

How does STUN help? It doesn't help in all situations; in fact, it can't help for symmetric NATs where the NAT mapping varies depending on the outgoing destination IP address — the IP address of the STUN server and of the peer-to-peer client differ, so create different mappings in the NAT translator, so the WAN port returned in the STUN response is worthless.

However, for less restrictive NAT translators (such as cone translators, where the outgoing destination IP address doesn't affect the NAT mapping), STUN can provide complete NAT traversal.

TURN

- RFC 5766
- Relays connections through a well-known server outside the NAT
- Expensive, slow, high latency
- Both clients are guaranteed to be able to connect to the TURN server

When STUN fails, TURN works, but at a high cost. In TURN, all packets between two clients are relayed through a well-known server outside the NAT, addressible by both clients. This adds latency, imposes a huge bandwidth cost on the TURN server operator and is generally expensive. But it's reliable.

In more detail, client A sends an allocation request to the TURN server, which allocates and returns a public IP address and port on the server. The client sends these out-of-band to client B, and then sends a 'create permissions' request to the server with client B's address (possibly retrieved via STUN), allowing it to connect to the allocated port. Client B can then connect and send and receive data. The relay server will forward packets to and from client A.

Forwarding adds a large header (36 bytes per packet), or requires a specific channel to be set up (4 bytes per packet).

TURN is good for cases where both clients are behind separate NATs, or when symmetric NATs are used. But should be a last resort.

ICE

Basically,

$$\text{ICE} = \text{STUN} + \text{TURN}$$

- RFC 5245 (117 pages of gripping reading!)
- Clients are ignorant of network topologies and NAT configurations
- Clients have an out-of-band signalling channel
- Candidates:
  - Host
  - Server reflexive
  - Peer reflexive
  - Relayed
- Candidates are paired and checked
- Pairs are prioritised and selected

In order to use STUN or TURN correctly, the clients have to have some idea of network topology and NAT configuration, which is entirely unreasonable to assume. Enter ICE, which allows for connection establishment without any prior knowledge. The only requirement it has is that the clients have an out-of-band signalling channel over which they can exchange control messages. This is typically implemented using a well-known rendezvous server outside the NATs. Since the server only handles control messages (and not, e.g., VoIP data) it doesn't get loaded anywhere near as much as a TURN server.

How does ICE work? Clients start by gathering all the candidate addresses they can for themselves: the addresses on all host interfaces (host), the results of STUN queries (server reflexive), and allocated addresses on TURN servers (relayed). They send these candidates to the other client over the signalling channel.

Both clients then pair up all their local candidates with the remote candidates from the other host, taking the cross product of the two sets. Both clients then prioritise pairs (so that pairs more likely to succeed are tested first), and proceed to check each pair using a STUN request from the pair's local address to its remote address.

If a pair works, a STUN response will be received from the other client. As it's a STUN response, it will contain the IP address and port the other client saw the request originate from. This may differ from other local candidates the client already knew about — if so, it's discovered a new peer reflexive candidate, which is added to the pool and paired up with the remote candidates.

Once a good working pair is found, that pair is selected and a connection has been established. Note that it's possible for multiple pairs to succeed, and they may succeed out of priority order. It's up to the application to choose a tradeoff between connection establishment latency and quality of established connection.

So that's ICE. This has glossed over a lot of the detail — there are a lot of timeouts, retries, authentication, prioritisation and race condition considerations implicit in the protocol, all covered in glorious detail in the RFC. But that's the big picture.

Show me the code!

```c
context = g_main_context_new ();
loop = g_main_loop_new (context, FALSE);
agent = nice_agent_new (context,
                        NICE_COMPATIBILITY_RFC5245);
stream_id = nice_agent_add_stream (agent, 1);

/* Do _not_ use this in production code:
 * use nice_agent_recv_messages() instead. */
nice_agent_attach_recv (agent, stream_id, 1,
                        context, recv_cb, NULL);
nice_agent_gather_candidates (agent, stream_id);

/* Send credentials to the other client via the
 * signalling channel. */
nice_agent_get_local_credentials (agent, stream_id,
                                  &ufrag, &pwd);

g_main_loop_run (loop);
```

Show me the code!

```c
static void
new_candidate_full_cb (NiceAgent *agent,
                       NiceCandidate *candidate,
                       gpointer user_data)
{
    /* Send the @candidate to the other client
     * over the signalling connection. */
}

static void
candidate_gathering_done_cb (NiceAgent *agent,
                             guint stream_id,
                             gpointer user_data)
{
    /* Notify the other client that no more
     * candidates will be sent. */
}
```

Show me the code!

```
static void
component_state_changed_cb (NiceAgent *agent,
                            guint stream_id,
                            guint component_id,
                            guint state,
                            gpointer user_data)
{
    if (state == NICE_COMPONENT_STATE_READY) {
        /* Connection established!
         * You can start sending now. */
    }
}
```

If you understand ICE, the design of libnice should make sense pretty quickly, since its API is based directly on the ICE specification. A few introductory points: you create one `NiceAgent` per client. Each `NiceAgent` can contain multiple streams, and each stream has one or more components; this is just a way of multiplexing related connections between the same two clients. These are RTSP concepts. For most use cases, a stream will have a single component, and only a single stream will exist between each pair of clients.

libnice is quite specific about which GLib main contexts it uses for things, as it has a multitude of timers which run in the background to implement various bits of the ICE protocol. In order to use libnice, you must have a GLib main context running, in some thread, somewhere.

Practical example

- Client and agent
- Signalling server
- Relay server controlling multiple TURN servers

To make a practical system using libnice, you need more than the two peer implementations. You definitely need some kind of signalling server which is hosted at a well known address accessible from behind all NATs — this will marshal the out-of-band signalling information needed by ICE.

If TURN relaying is to be used (and it should be, if you want your peer-to-peer connection to work for all NATs), you will need a TURN relay server such as coturn. You will probably want several instances in order to perform load balancing, because TURN servers get heavily loaded. These will need a control server to keep them all in check and to synchronise with the signalling server.

So at its core, a libnice-based peer-to-peer system is quite simple (a few hundred lines of C to implement the core of a peer), but the housekeeping and ancillary code builds up a little, unfortunately.

The future

- TURN-TCP support
- High-level API for candidate gathering and negotiation

- Pseudo-TCP performance improvements

There are a few things which are on the roadmap to be implemented when time allocations allow. TURN-TCP support would allow TURN to be used in networks where UDP is blocked (since TURN normally uses UDP). A high-level API for candidate gathering would eliminate the manual tracking of stream and component IDs, and the requirement to implement your own gathering state machine in the signal handlers from `NiceAgent` when implementing a simple peer-to-peer ICE system. Pseudo-TCP is a sub-library within libnice which implements TCP over UDP (originally borrowed from libjingle). It's a bit slow at the moment and could benefit from implementation of newer TCP specification performance enhancements, and some general profiling work.

Miscellany

RFC 5389 (STUN) `http://tools.ietf.org/html/rfc5389`

RFC 5766 (TURN) `http://tools.ietf.org/html/rfc5766`

RFC 5245 (ICE) `http://tools.ietf.org/html/rfc5245`

libnice `http://nice.freedesktop.org/`