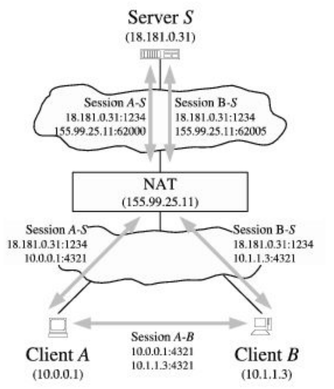
Enhanced CoVim

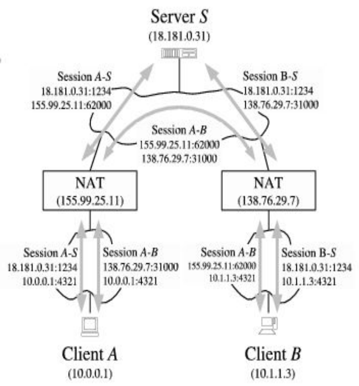
B04902017 李立譽 B04902025 施博瀚 B04902037 顏子斌
B04902043 謝宏祺   B04902045 孫凡耘 B04902096 陳力榕

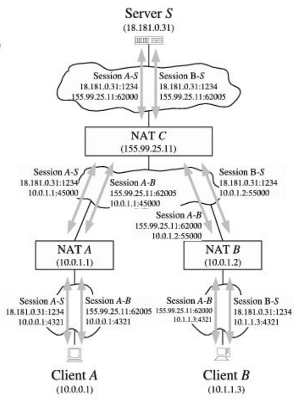
1. Motivation
   Not long ago, Microsoft introduces Code Sharing functionality. However, not many people use VSCode while coding. So why not make this functionality on Vim?

   For more information regarding of VSCode's Live Code sharing, refer to the two following links.
   https://docs.microsoft.com/en-us/visualstudio/liveshare/faq
   https://www.youtube.com/watch?v=GQxOGL-dghs

2. Technical
   a. NAT
      i. Full name of NAT is Network Address Translation, it is used to solve the problem of lack of ipv4 address.
      ii. But when we use P2P connection between 2 host under NAT, it will cause some problem.
      iii. Therefore we need a technique called hole punching to resolve the problems.
   b. UDP Hole Punching
      i. There are three parts in UDP Hole Punching. They are Rendezvous server S, Client A, and Client B, respectively.
      ii. Procedure as below:
      First, A asks S to establish session with B.
      Second, S will reply A about B's info, and also tell B that A want to connect with B by telling A's info.
      Third, A starts sending UDP packet to B, waiting for valid response from B and finally "locks in". And so does B. Then the job is done. The reason why we can have each others' UDP packets reach each other is that by continuously sending UDP packets, each side effectively sets up an entry in their corresponding NAT translation table that maps a public address to their private address; thus by sending UDP packets to the public address of the translation entry, the packets will be passed to the corresponding hosts behind their NAT and therefore creates a tunnel that can be used to pass data.
      iii. performance in different scenario



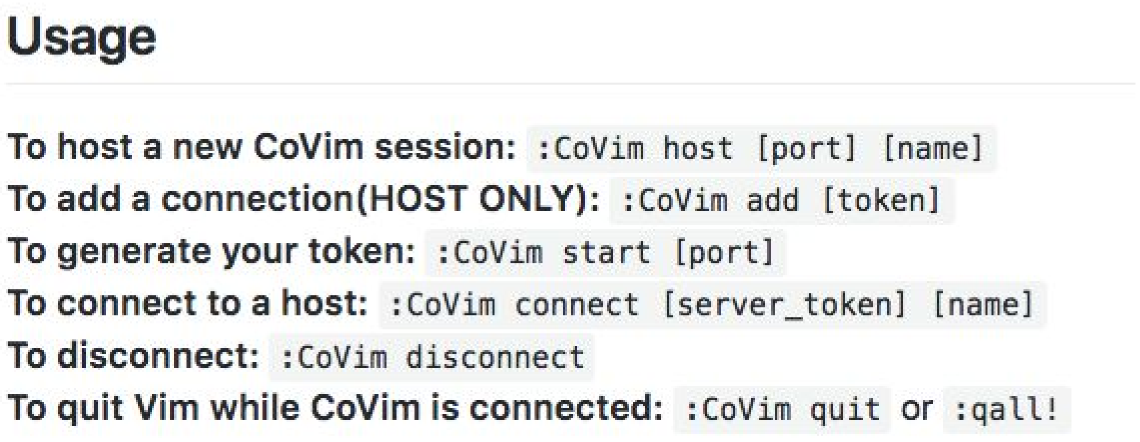




c. TCP Hole Punching
   i. It is not as well-understood as UDP hole punching, so it is supported by fewer existing NATs.
   ii. Similar hole punching process as that of UDP.
   iii. Corresponding to the repeatedly sent UDP packets in UDP hole punching, in TCP hole punching the SYN packet is used instead.
   iv. Two different scenarios according to endpoint TCP implementation. Suppose that A's SYN packet is dropped by B's NAT and B's SYN packet get through A's NAT via the hole previously created by A's SYN packet:
      1. Linux & Windows: A notices the received SYN packet's source matches one of outbound session's destination. Thus it responses by having the outbound connect session enter the connected state.
      2. BSD: A notice that there is an active listen socket listening at the destination of the SYN packet and thus create a new stream socket with A's connect attempt failing.

3. Implementation
   a. Design & Architecture

We changed the "api" of CoVim to the following:



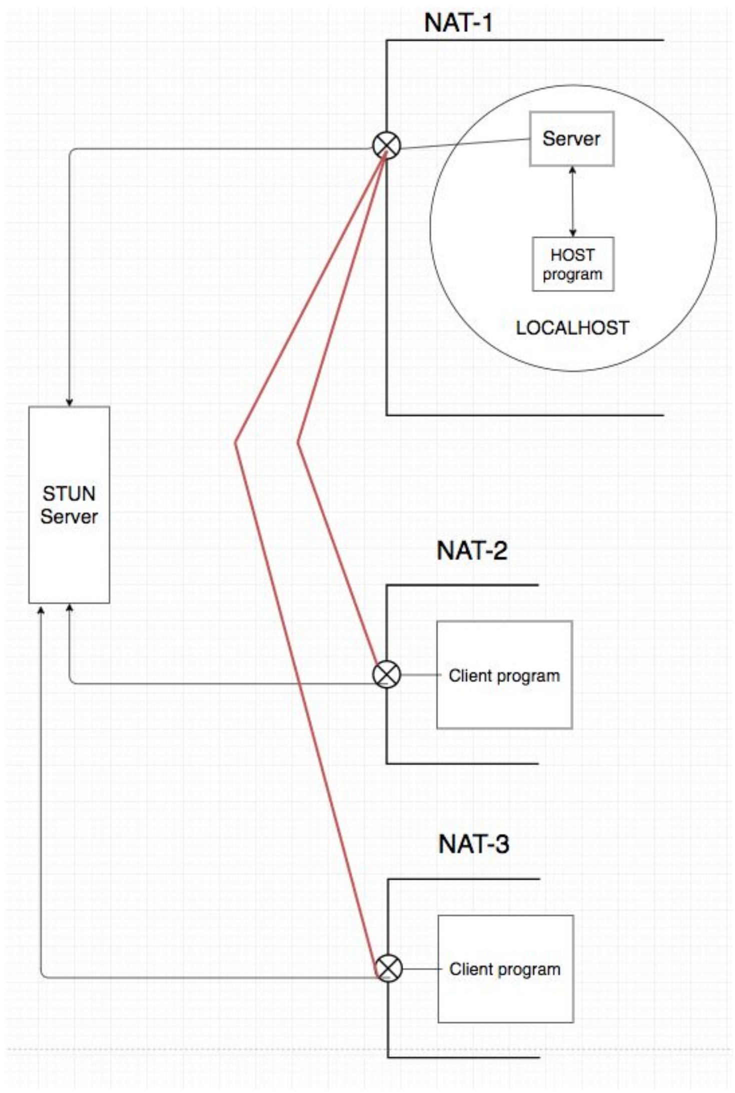
Usage

To host a new CoVim session: `:CoVim host [port] [name]`
To add a connection(HOST ONLY): `:CoVim add [token]`
To generate your token: `:CoVim start [port]`
To connect to a host: `:CoVim connect [server_token] [name]`
To disconnect: `:CoVim disconnect`
To quit Vim while CoVim is connected: `:CoVim quit` or `:qall!`

Basically, the workflow goes like this:

   i. Establish connection. (TCP hole punching)

ii. Hand socket connection to Twisted (Protocol).
iii. Server "listens" for any "movement" from all clients and broadcast it.
iv. Server quits after all users are disconneced.



This illustrates the whole architecture.

b. Difficulties
   i. The first problem is that how should we design the architecture. Should we use peer-to-peer completely or should we adopt a server the handles all events. Last, we adopted mostly the original structure from CoVim project but we tweaked it(as the figure above) a little bit to fit our design into it.

   ii. There are different types of NAT. Not only is there different types of NATs, even the same type of NAT can be implemented differently. So we have a hard time ensuring that this technique can work in most cases.

```
Blocked = "Blocked"
OpenInternet = "Open Internet"
FullCone = "Full Cone"
SymmetricUDPFirewall = "Symmetric UDP Firewall"
RestricNAT = "Restric NAT"
RestricPortNAT = "Restric Port NAT"
SymmetricNAT = "Symmetric NAT"
```

A great explanation of them can be found here(http://www.cs.nccu.edu.tw/~lien/Writing/NGN/firewall.htm)

iii. Sincer we are using Twisted as the main framework. It is inheritly an event-driven networking engine. It took me a long time to figure out how to use my customized connection method(for TCP hole punching) and then hand it over to the framework. Twisted is a enormous framework, it gives us a lot of flexibility. At the same time, it brings more difficulties.

iv. There are different cases in the connection that we must consider. Two computers can be both on public IPs, one with public IP and the other behind a NAT, or both of them are behind NAT. If both of them are behind NAT, there are cases that they share the same router(same private network) or they are in completely different NATs. There are all sorts of cases and we had to make sure that our code works in all cases. At first, I found out that my code only works in certain scenario(as the following):
   1. Same computer behind NAT(same NAT router) (X)
   2. Two public IPs (O)
   3. One public IP, one behind NAT(O)

   So I had to fix it to make sure everything works nicely.

v. In order to make everything work nicely, threading and multiprocessing are considered during implementation. That's because the host has to run his own client-side program as well as the server-side program. Should we run it separately as different programs or should we use Python's Multithreading/Multiprocessing module? I tried to use threading and multiprocessing first but I figured that it can be problematic. At last, we run client program and server program separately(only the HOST has to run the server program). However, threading may still needed for hole punching. In fact, the original implementation of the TCP hole punching paper that we mentioned above uses threading.

vi. Hard to debug. There are times when we cannot reproduce issues. Sometimes, is may be caused by different implementations and behaviors on different platforms(Windows, Linux, e.t.c).Sometimes, it's
because some sockets that are not handled correctly. So I learned two commands to

Show using ports

```
netstat -lntu | grep [port]
```

To Kill sockets(because they are not intended to be left there.)

```
fuser -k -n [protocol] [portno]
```

4. Reference
   a. NAT
   b. UDP Hole Punching
   c. TCP Hole Punching
   d. Peer-to-Peer Communication Across Network Address Translators