

ECSE 416 - Final Project

Nathan Clairmonte - 260673075

Ali Shobeiri - 260665549

P2P Chat with NAT Traversal

Problem Statement

It is very common that a user sits behind a network address translator (NAT) on a home networks. A common problem that P2P applications must deal with is that they need to detect users that have NATs and implement some procedure to get around this. Several methods are a NAT punch-through or NAT traversal mechanism [1]. A NAT is used to conserve IP addresses. It allows several unregistered private IP addresses to connect to the public internet. IP addresses on the private network will not be globally unique but will be locally unique. Anytime a device on the private network wants to connect to the outside internet, it must first go through the NAT. The NAT also provides security services [2].

When a device on a private network wants to connect with the outside world it goes first through its NAT. The device's NAT then replaces the source address and port of the packet with its own IP address and a generated port mapping and then sends the packet to the desired destination. This approach works for client server architectures but becomes problematic in peer to peer applications. Consider the case where peer A is trying to establish a connection with peer B using peer B's public address. Peer A first must pass it's message through it's NAT. Peer A's NAT performs the necessary NAT operations, namely changing the source IP address of the message to its own and assigning a different port. The NAT also maintains a mapping of the assigned ports so that it can map it to original sender when it receives a reply. When the NATed message arrives at Peer B's NAT, Peer B's NAT is unable to resolve which device within its private network the message should be sent to and because of this, discards the message [3].

To avoid this problem, we need to implement a third party observer that exists within the global internet. In this manner, each peer could pass through their NATs and generate the necessary port IP address mappings and then share this info with the third party service. The third party service could then notify all the constituent peers in the peer to peer network of the IP addresses and port mappings of all the other devices in the network. It is also important to consider that a user's NAT will reject incoming connections from previously unrecognized destinations, therefore both users will have to send messages to each other simultaneously to bypass this. Using this approach, Peer A in our example could send its message to peer B, this is explained further below.

Methodology

A NAT punch-through or hole-punch is a method where a central server will act as an introducer to two peers [4]. For example, when peer A wants to send a message to peer B, it does not know the public IP address and NAT port mapping of peer B itself. To solve this problem, we introduce a central server hosted using NodeJS on the cloud hosting platform DigitalOcean. In this case, before a

chat connection can be established, each peer will contact the server and register their public IP addresses and port mappings with the server. The server will then notify all new connecting peers of the details of all the other existing peers registered on the network and notify all already connected peers that a new peer wishes to join the chat room. The server will then send all peers, whether new or existing a list of all the peers in the network.

Each peer will then attempt to hole punch with every other peer on the network, including new and already connected peers. This constitutes sending ten UDP packets in a sequence to each peer. At the same time, every other peer will attempt to send ten UDP packets to one another as well. This methodology allows us to bypass the NAT firewall by masquerading our UDP packets as a response to a message already sent out by one of the peers. The number ten was chosen to allow for reasonable time in establishing a connection and to account for potential delays in each peer beginning the hole punching procedure with other peers.

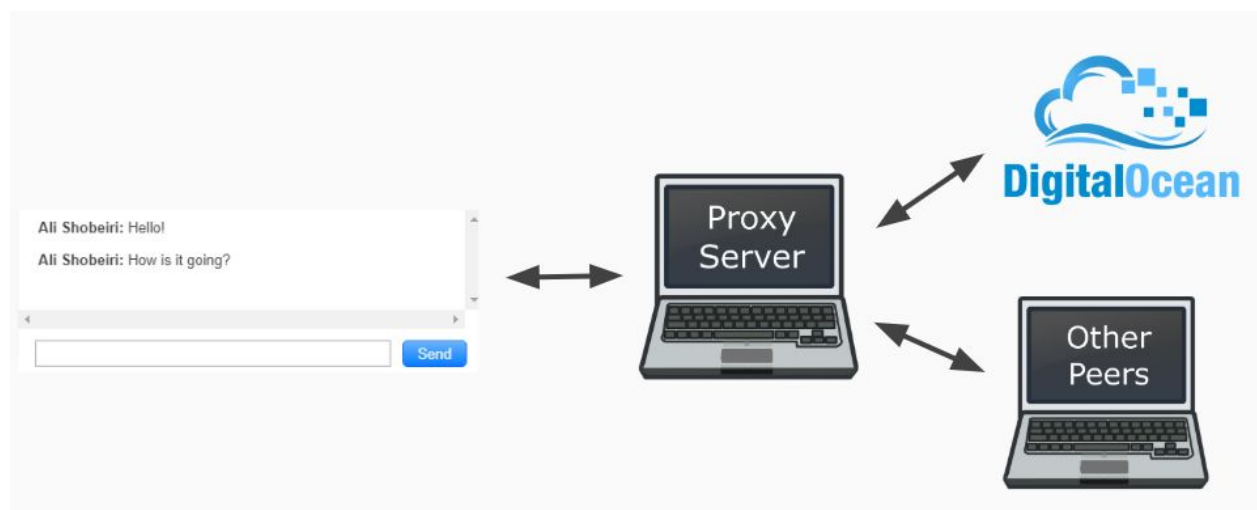


Figure 1: System Architecture

Our entire system architecture is seen above. The frontend system we developed was built using the web development framework React. This is seen on the left hand side of the image. The React is used in a desktop app using Electron. A limitation of this architecture means that we cannot directly use our frontend to generate and send UDP packets. This is because of limitations placed on browsers and their ability to generate UDP packets. To get around this, we create proxy server that is directly connected to the frontend application using a socket connection.

Our proxy server is our controller performing the logic on the client side for all the peers. Our proxy server is able to handle events triggered using our frontend as well as serve data to the frontend for displaying. More importantly, our proxy server is used to hole punch with other peers and is also used to connect to our cloud hosted server on DigitalOcean. Our proxy server was built using NodeJS and generates UDP packets using dgramJS.

The final part of architecture is the server hosted on DigitalOcean. This server, also built in NodeJS is able to receive requests from users and to persist the data of all the peers connected to the server. Furthermore, the server is able to periodically ping the proxy servers of all connected peers to see whether they are still connected and if they are not remove them from the list of all connected peers. The server is also able to send messages to every peer's proxy server when a new peer aims to join the network.

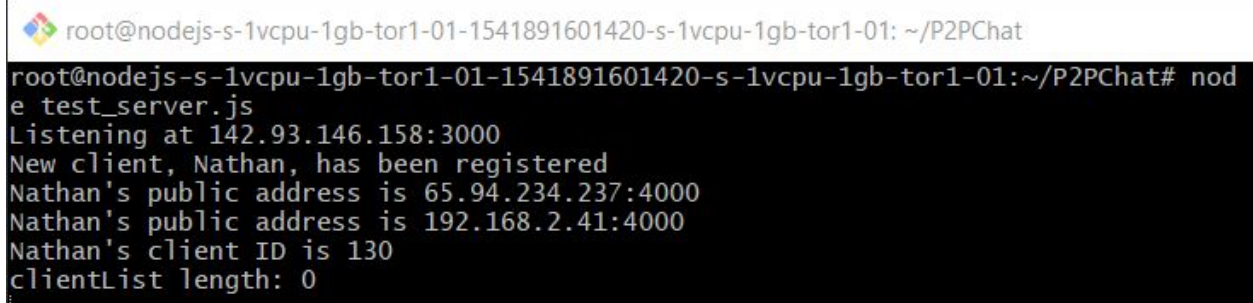
Implementation

The flow of our implementation begins with a server file. This Node file is hosted on a DigitalOcean virtual machine and is continuously running. Our implementation continues with a file *client.js*. Running this file opens a UDP socket on the client computer that is listening for incoming messages from either the frontend or the P2P server. Then, the frontend application is started. As seen in Figure 2, the frontend application begins with a prompt screen asking for a username.



Figure 2: Frontend application on the login screen

Once a username is submitted, the frontend sends a message back to *client.js*, informing the client of its selected username. Once the client receives this message, it calls the function *registerSelf* (using hard coded server IP and port values). If the server successfully receives and responds to the client's registration message, the server output will be as shown in Figure 3, and the client output will be as shown in Figure 4. The server response contains the current client list held by the server, as well as a client ID for the client that has been assigned by the server.



```
root@nodejs-s-1vcpu-1gb-tor1-01-1541891601420-s-1vcpu-1gb-tor1-01: ~/P2PChat
root@nodejs-s-1vcpu-1gb-tor1-01-1541891601420-s-1vcpu-1gb-tor1-01:~/P2PChat# node test_server.js
Listening at 142.93.146.158:3000
New client, Nathan, has been registered
Nathan's public address is 65.94.234.237:4000
Nathan's public address is 192.168.2.41:4000
Nathan's client ID is 130
clientList length: 0
```

Figure 3: Server output on new client registration

```
MINGW64:/c/Users/natha/Desktop/ECSE 416 Project/P2PChat/backend
natha@DESKTOP-ED4PTN5 MINGW64 /c/Users/natha/Desktop/ECSE 416 Project/P2PChat/ba
ckend (master)
$ node client.js
server is running on port 8080
Nathan, you are about to register yourself with the Server...
Sending client packet to server...
Client packet sent, waiting on server response...
Server response received. You are now connected to the chatroom!
Reg_Response MSG: { msg_type: 'reg_response', clientList: [], client_id: 130 }
```

Figure 4: Client proxy server response on registration to server

After registration is complete, we move onto the hole punching. When a new client joins the network, the server will alert all existing clients on the network of the addition of a new client. Subsequently, existing clients will attempt to hole punch with the new client to establish a direct connection with them. The output messages from this hole punch attempt are shown in Figure 5.

```
AliShobeiri has joined the chatroom
Client List: [ { public_ip: '65.94.234.237',
  public_port: 4000,
  client_name: 'Nathan',
  private_ip: '192.168.2.41',
  client_id: 170,
  private_port: 4000,
  send_ip: 0,
  send_port: 0,
  ping_sent: false,
  ping_received: false },
  { public_ip: '65.94.234.237',
    public_port: 1024,
    client_name: 'AliShobeiri',
    private_ip: '192.168.2.180',
    client_id: 127,
    private_port: 4000,
    send_ip: 0,
    send_port: 0,
    ping_sent: false,
    ping_received: false } ]
Attempting holepunch SYN with AliShobeiri
Received holepunch SYN from AliShobeiri (192.168.2.180:4000)
Sending IP and port have been defined:
send_ip: 192.168.2.180
send_port: 4000
Received holepunch ACK from AliShobeiri (192.168.2.180:4000)
```

Figure 5: Client hole punching procedure with new peer

When the hole punching process is complete, assuming it is successful, all clients in the server's client list are now connected to the P2P chat room and can send messages that will be received by

everyone else in the chat room. To illustrate the message sending functionality, a message was sent from client “Nathan” to client “AliShobeiri”. The client side output of this action is shown in Figure 6, and the frontend output is shown in Figure 7, both on the side of client “Nathan”.

```
{ id: 0, author: 'Nathan', message: 'Hey' }  
Sending clients message  
Msg: { "msg_type": "chat_message", "id": 0.1286400777105503, "message": "Hey", "author": "Nathan" }  
Peers: [ { public_ip: '65.94.234.237',  
  public_port: 1024,  
  client_name: 'AliShobeiri',  
  private_ip: '192.168.2.180',  
  client_id: 236,  
  private_port: 4000,  
  send_ip: '192.168.2.180',  
  send_port: 4000,  
  ping_sent: false,  
  ping_received: false } ]  
Peer: { public_ip: '65.94.234.237',  
  public_port: 1024,  
  client_name: 'AliShobeiri',  
  private_ip: '192.168.2.180',  
  client_id: 236,  
  private_port: 4000,  
  send_ip: '192.168.2.180',  
  send_port: 4000,  
  ping_sent: false,  
  ping_received: false }  
AliShobeiri's send_port: 4000  
AliShobeiri's send_ip: 192.168.2.180
```

Figure 6: Sending message (client side).

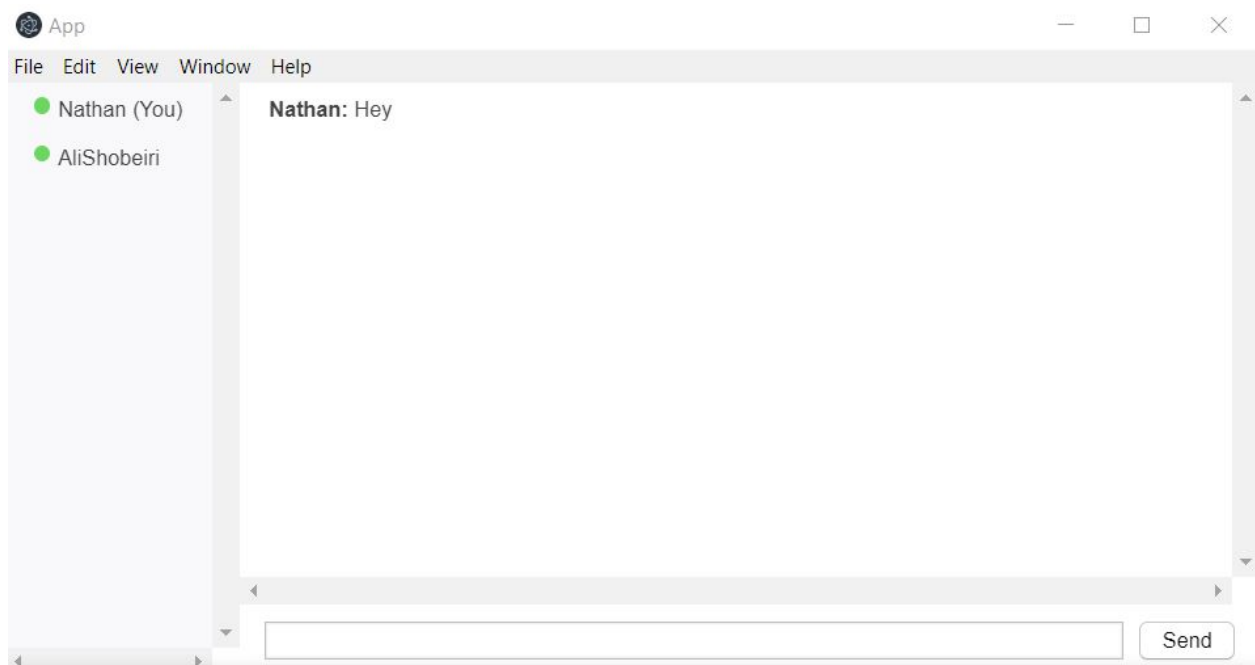


Figure 7: Sending message (frontend).

To illustrate the receiving end of the messaging functionality, a message was sent from client “AliShobeiri” to client “Nathan”. The client side output of this action is shown in Figure 8, and the frontend output is shown in Figure 9, both still on the side of client “Nathan”.

```
AliShobeiri: Hey
{ msg_type: 'chat_message',
  id: 0.5321729479338513,
  message: 'Hey ',
  author: 'AliShobeiri' }
```

Figure 8: Receiving message (client side)

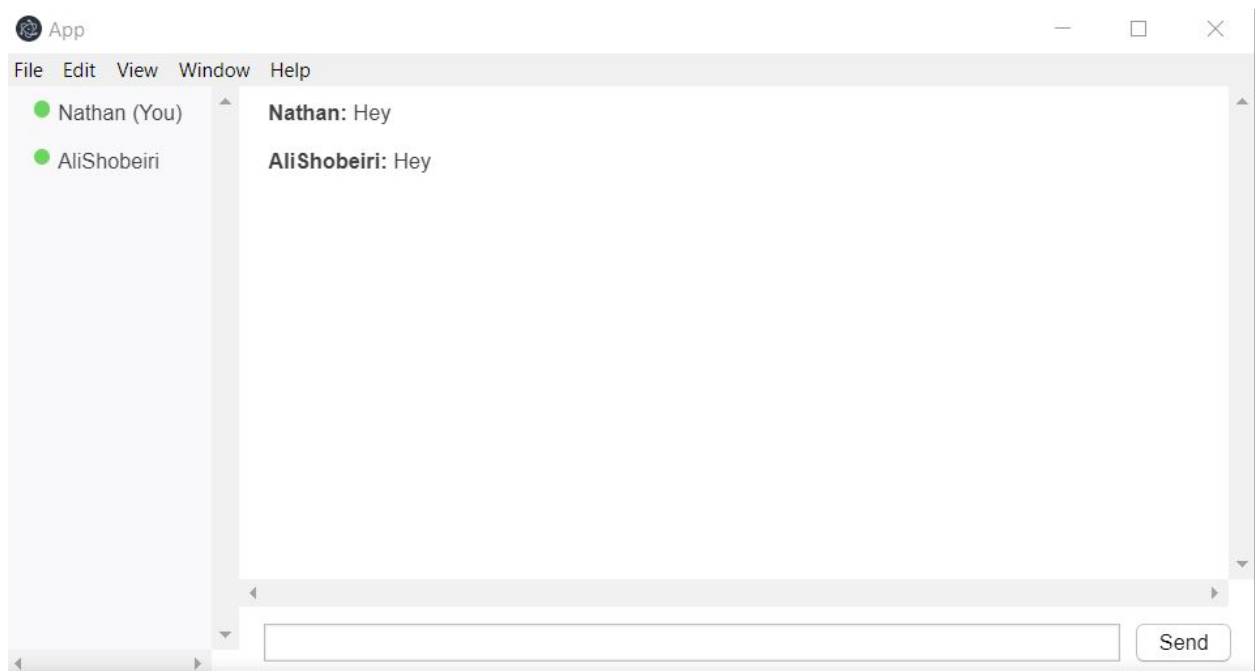


Figure 9: Receiving message (frontend)

In addition to messaging functionality, the chat room also facilitates a makeshift peer detection system. The way this works is as follows: the server continually pings all existing clients in the network (once every 10 seconds), and then waits for their responses (Figure 10). If the server sends a ping to any given client and does not receive a response within 5 seconds of sending, the server marks that client as disconnected (Figure 11) and alerts all existing clients of the fact that a client has left the network (Figure 12).


```

Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, AliShobeiri is still connected
Ping ACK received, Nathan is still connected
Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, AliShobeiri is still connected
Ping ACK received, Nathan is still connected
Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, Nathan is still connected
Ping ACK received, AliShobeiri is still connected
Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, Nathan is still connected
Ping ACK received, AliShobeiri is still connected
Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, Nathan is still connected
Ping ACK received, AliShobeiri is still connected

```

Figure 10: Output of server showing pinging mechanism.

```

Ping sent to Nathan
Ping sent to AliShobeiri
Ping ACK received, Nathan is still connected
A client has left. There were 2 clients, now there are 1 clients
Current clientList: [ { public_ip: '65.94.234.237',
  public_port: 4000,
  client_name: 'Nathan',
  private_ip: '192.168.2.41',
  client_id: 205,
  private_port: 4000,
  send_ip: 0,
  send_port: 0,
  ping_sent: true,
  ping_received: true } ]

```

Figure 11: Output of server showing client disconnection.

```

AliShobeiri has left the chatroom
Peers list after lost client message: []

```

Figure 12: Output of client “Nathan” showing that client “AliShobeiri” has left.

Conclusion

In this project, we were able to use several existing technologies, namely React, Electron and NodeJS to develop a peer to peer chat system using NAT traversal and specifically UDP punchthrough. This type of NAT traversal has lots of applications to other important projects as well and can be very useful for almost any type of peer to peer communication. As an extension of this project, we would like to potentially extend our functionality past just messaging and bring in file sharing and potentially voice over IP as well.

Source code

<https://github.com/alishobeiri/P2PChat>

References

- [1] Ford, Bryan & Srisuresh, Pyda & Kegel, Dan. (2006). Peer-to-Peer Communication Across Network Address Translators.
- [2] Halkes, G. and J. Pouwelse (2011). UDP NAT and Firewall Puncturing in the Wild, Berlin, Heidelberg, Springer Berlin Heidelberg.
- [3] Srisuresh, P., Ford, B., and D. Kegel, "State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)", RFC 5128, DOI 10.17487/RFC5128, March 2008, <<https://www.rfc-editor.org/info/rfc5128>>.
- [4] D. Maier, O. Haase, J. Wäsch and M. Waldvogel, "NAT hole punching revisited," 2011 IEEE 36th Conference on Local Computer Networks, Bonn, 2011, pp. 147-150.
- [5] Holdrege, M. and P. Srisuresh, "Protocol Complications with the IP Network Address Translator", RFC 3027, DOI 10.17487/RFC3027, January 2001, <<https://www.rfc-editor.org/info/rfc3027>>.