Data Networks, Final Project

Instructor: Dr. MohammadReza Pakravan

Simple Chat Application

Spring 2018

Beginning Notes: In this project we are going to develop a simple chat application. Using this application you can text message your friends and also transfer files by peer to peer communication. Although our intended application is so simple but we face several difficulties in doing the job. First, a common problem in implementing any peer-to-peer protocols over the Internet is the existence of NATs in the networks. In the course you have learned that NATs translate your local private IP address to a public one over the Internet. The question now is, how two nodes are able to communicate directly if one or both of them are behind NATs? To answer this question, is the first problem in this assignment. The second problem is that your application, i.e., chat, runs on top of a UDP connection so you have to implement a reliability and congestion control algorithm.

1 Implementation Tools

For this project you are free to choose any programing language you prefer (even Matlab, but not recommended). If you do not have any programming experience, we suggest that you choose Python, which is an easy language to learn and there are many online tutorials for it.

2 NAT Traversal

NAT Traversal is a computer networking trick for establishing end to end connectivity across the NATs. Before talking about NAT Traversal techniques, we should know about different types of NATs. Search about NAT types and answer the following questions (Include your reference.).

What are the differences between the following NAT types and how do they work?
1. Full Cone NAT (Static NAT)
2. Restricted Cone NAT (Dynamic NAT)
3. Port Restricted Cone NAT (Dynamic NAT)
4. Symmetric NAT

A well-known NAT traversal technique is "UDP hole punching". This technique enables two clients A and B to set up a direct peer-to-peer UDP session with the help of a rendezvous server S, even if the clients are both behind the NATs. Hole punching assumes that the two clients, A and B, already have active UDP sessions with a rendezvous server S. When a client registers with S, the server records two endpoints for that client:

- the (IP address, UDP port) pair that the client believes itself to be using to talk with S,
- the (IP address, UDP port) pair that the server observes the client to be using to talk with it.

We refer to the first pair as the client's private endpoint and the second as the client's public endpoint. The server might obtain the client's private endpoint from the client itself in a field in the body of the client's registration message, and obtain the client's public endpoint from the source IP address and source UDP port fields in the IP and UDP headers of that registration message. If the client is not behind a NAT, then its private and public endpoints should be identical. Suppose client A wants to establish a UDP session directly with client B. Hole punching proceeds as follows:

- A initially does not know how to reach B, so A asks S for help establishing a UDP session with B.
- S replies to A with a message containing B's public and private endpoints. At the same time, S uses its UDP session with B to send B a connection request message containing A's public and private endpoints. Once these messages are received, A and B know each other's public and private endpoints.
- When A receives B's public and private endpoints from S, A starts sending UDP packets to both of these endpoints, and subsequently locks in whichever endpoint first elicits a valid response from B. Similarly, when B receives A's public and private endpoints in the forwarded connection request, B starts sending UDP packets to A at each of A's known endpoints, locking in the first endpoint that works. The order and timing of these messages are not critical as long as they are asynchronous.

Read [1] to learn the details of hole punching technique and answer the following questions,

---

Describe the UDP hole punching mechanism in the following scenarios:
1. Peers are behind a common NAT.
2. Peers are behind different NATs.
3. Peers are behind multiple levels of NATs.

There are several standardized protocols in IETF specifications for NAT traversal. STUN, ICE and TURN are three protocols that jointly provide a complete NAT traversal solution.

Briefly describe the role of the following protocols in initiating a peer to peer communication,
1. RFC 5389: Session Traversal Utilities for NAT (STUN),
2. RFC 5766: Traversal Using Relays around NAT (TURN),
3. RFC 5245: Interactive Connectivity Establishment (ICE).

---

3 Simple Text Chat Application

Now its code time! in this part we develop a simple text chat application over Internet. For this project we have provided a server which helps you a lot. The server is running on "dnrl.ir (ip address: 88.99.8.253)" and is listening to port "13570". Suppose that "Amir" wants to chat with "Reza". Both Amir and Reza have a valid 16 byte token. Tokens are used for authentication, only users with valid tokens can communicate with the server (you will be provided with two tokens, DO NOT share your tokens with anyone). As the first step Amir and Reza must register with the server so they both send a register message to the server over a UDP connection (replace [token] with your 16 byte token),

```
REGISTER:[token]:Amir # Amir sends this message, saying my id is ''Amir''
REGISTER:[token]:Reza # Reza sends this message, saying my id is ''Reza''
```

Note that the clients are required to send packets in the same ASCII format as above. Once the users are registered on the server they receive an acknowledgment from the server,

```
Registered # Amir and Reza  receive this if they are successfully registered.
```

If the username is already taken, you receive `DuplicateUserName`.Note that for all commands if you send the message in wrong format you will receive `UnknownCommand` and if you send message with an invalid token you will receive `UnAuthorized`. Once a user is successfully registered, it should periodically a send a `HEARTBEAT` message to confirm that it is still active. If no `HEARTBEAT` is received by the server within one minute it assumes that the user is no longer available.

```
HEARTBEAT:[token]  # users should send this message to confirm that they are
    active.
```

For starting a peer to peer communication Amir sends an invite message to Reza,

```
INVITE:[token]:Reza # this is the invite message format INVITE:[Caller token
  ]:Callee username.
```

when the server receives the INVITE message, it sends a `START` message to Reza and another message to Amir containing the Port and IP address of the other peer. These Port and IPs are the translated Port and IPs by the NATs. The server has extracted and recorded these addresses from the ip header of the packet carrying the `REGISTER` messages.

```
START:{{213,233,188,133},54680} # By receiving this message Amir understand
  that to which address and port he should send messages .
START:{{210,15,140,153},43740} # and also Reza.
```

Now hopefully Reza and Amir are able to start talking by sending their messages to the obtained ip and port addresses, but there are always barriers for conversations. By now you should know that symmetric NATs do not permit communication by these obtained addresses (do you really know?!). In this case the server can help you by relaying the messages. Amir sends a request for to the server,

```
MAKE_RELAY:[token]:Reza # Amir sends this request to server
```

Upon receiving this message the server will check if Reza is registered in the server. If so the server sends a `RELREQ_FROM` message to the Reza and a `SUCCESS` to Amir, otherwise it sends `PeerNotFound` to Amir.

```
RELREQ_FROM:Amir# Reza receives this request
```

Reza acknowledges this request by sending an `ACK_RELAY` message

```
ACK_RELAY:[token]:Amir # Reza sends an acknowledgment
```

when the server receives the acknowledgment, it assigns a random 16 byte session key to Amir and Reza and sends a `RELTO` message to Amir and Reza informing them their assigned session key

```
RELTO:Reza:[session key] # Amir receives this message
RELTO:Amir:[session key] # Reza receives this message
```

Now Reza and Amir can communicate with each other through the server, by sending their message in following format

```
RELAY:[session key]:Message Body # Amir and Reza can send their messages to
  other leg in this format.
```

The server will respond to the sender with `RELAYED` if it successfully receives the message and forwards it to the peer.

Message	Direction	Purpose
REGISTER	To Server	registering
HEARTBEAT	To Server	confirming activation
REGISTERED	From Server	acknowledging the registration
INVITE	To Server	requesting for a call
START	From Server	starting direct link
MAKE_RELAY	To Server	requesting for a relay link
RELREQ_FROM	From Server	indicating a relay link from the callee is requested
ACK_RELAY	To Server	acknowledging the relay request
RELTO	From Server	starting the relaying operation
RELAY	To Server	sending message to other peer

After initiating a link (peer to peer or relay), a predefined transport layer protocol is required for the peers to be able to translate the messages. This protocol must be reliable and guaranty in sequence delivery of packets. We fix the following message body for all data packets,

1. Magic Number [2 Bytes]
2. Packet Type [1 Byte]
3. Ack Flag [1 Byte]
4. Total Packet Length [4 Bytes]
5. Sequence Number [4 Bytes]
6. Acknowledgment Number [4 Bytes]
7. Data

32 Bits

Magic Number	Packet Type	ACK Flag
Total Packet Length		
Sequence Number		
Acknowledgment Number		
Data		

Note that all fields are transmitted in “big-endian” order.

Packet Structure

- Magic Number: the first field is a magic number for identifying the packets, it is always set to 0x73DF in hex.
- Packet Type: this field identifies the type of payload carried in the packet. For text messages set this field to 0x01.
- ACK Flag: It indicates the validity of Acknowledgment Number. In case it is set to 0x01, Acknowledgment Number is valid. Otherwise the, Acknowledgment Number is discard.
- Total Packet Length: Indicates the total packet length in bytes, including the 16 byte header the data field.
- Sequence Number: specifies the number assigned to the first byte of data in the current message.

- Acknowledgment Number: contains the value of the next sequence number that the sender of the packet is expecting to receive, if the ACK Flag is set

For the text message application the transmitter follows stop and wait protocol, *i.e.*, new data packet is sent whenever all the previously sent data packets are acknowledged. When the sender sends a data packet it starts a timer for it. It then waits for a fixed amount of time to get the acknowledgment for the packet. If the timer is expired the sender resends the packet. The receiver sends back an Acknowledgment packet upon receiving any packet with data, the acknowledgment may be sent by a packet containing data if the peer has data to send. Otherwise the receiver may send a packet with empty data field and a valid Acknowledgment Number .

4 File Transferring

In the next step we add file transfer capability to the clients. The client reads bytes from a file and transmits the data to its peer over several datagram packets. In previous part of the project you implemented the stop and wait protocol. The task of this part is to implement a sliding window protocol and control the window size by a congestion control algorithm. You should at least implement the TCP Tahoe which you have learned in the class. As a bonus you can read about other congestion control algorithms and implement them.

(Bonus) Explain the following congestion control algorithms and implement them in your code,
1. TCP Reno,
2. TCP Vegas,
3. TCP Hybla,
4. TCP Illinois.

For this part you should only send the packets in relay mode, our server will drop some packets randomly to simulate the congestion in the network. The packet formats are the same as last part, but you should set the Packet Type field to 0x02 for this application.

Creat a log of the congestion window size and plot a sample of the congestion window size in your report.

---

5 Bonus: Misbehaving TCP Receiver

Many of the Internet protocols work well if both of the receiver and transmitter behave as they should behave. Misbehaving devices are often used to attack the networks. There are many simple attacks that allow a misbehaving receiver to drive a standard TCP sender arbitrarily fast, without losing end-to-end reliability. Many researchers have studied these attacks and suggested modifications for TCP to eliminate this undesirable behavior. See [2] to learn more about these attacks.

In this part, your task is to review the misbehaving receiver attacks and the proposed solutions for preventing them. You should simulate one of these attacks in your code, and generate plots that show the effects of this attack.

Note: For Bonus parts we do not expect a unique answer, and you should provide your answers based on your understandings. The grading is with respect to your efforts.

6 What SHOULD I prepare?

You should upload you codes and a report file in .pdf format. In presentation session you run your codes. In the report file you should answer the questions asked in project.

7 Grading Policy

This information is subject to change, but will give you a high-level view of how points will be allocated when grading this assignment.

- **Communication with server [20 points]:** you code should be able to sent proper requests and correctly response to server.
- **Text messaging [20 points]:** your code should be able to send and receive the messages. It should properly send acknowledgments, and handle data losses.
- **File transfer [20 points]:** your code should be able to perform basic congestion control. It should be robust against lost acks or out of order packets, etc.
- **Robustness [10 points]:** your code should be crash free and it should be resilient to peers that send corrupt data.
- **Coding style [10 points]:** well documented, well structured, clean codes are welcomed.
- **Report [20 points]:** your report should contain proper answers to question and well written.
- **Bonus [$\leq$ 50 points]:** this is for the parts of the project marked as Bonus. These points will be allocated based on the results you provide.

References

[1] Bryan Ford, Pyda Srisuresh, and Dan Kegel. 2005. Peer-to-peer communication across network address translators. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, Berkeley, CA, USA, 13-13.

[2] Stefan Savage, Neal Cardwell, David Wetherall, and Thomas Anderson. TCP Congestion Control with a Misbehaving Receiver. ACM SIGCOMM Computer Communications Review, vol. 29, no. 5, October 1999, pages 71 - 78.