CSC/ECE 573 - Internet Protocols

Project Report

Chat Application using UDP with Go-Back-N

Submission Date: 3rd December 2017

Team Members

Pranav Vaidya (psvaidya)

Karan Jadhav (kjadhav)

Sriram Guddati (skguddat)

INTRODUCTION

Chat applications are used to communicate with each other. However, for the chat applications to be considered reliable, they must have a guaranteed message delivery service. Chat applications using TCP can provide reliability, but TCP is a connection oriented service which uses 3-way handshake before sending data. This handshaking introduces additional overhead which contributes toward the end-to-end delay. In chat applications where the data to be transferred is considerably small this overhead plays a role in the end-to-end delay. Due to initial handshaking TCP provides data integrity, but usually have longer latencies. So, our objective in this project is to create a chat application that uses reliable UDP instead of TCP to transfer messages.

Our implementation (using UDP over TCP) also ensures that all the messages sent are received at the receiver, which makes it reliable. Our implementation not only provides reliability but also ensure that the packets sent by the sender will be received in-order at the receiver. This would ensure both functionalities associated with TCP i.e. reliability and in-order delivery and at the same time avoiding the overhead associated with the use of TCP which makes our implementation faster than TCP. So, we made UDP, which is an unreliable connectionless protocol to a reliable protocol with high performance.

Our implementation is based on client server architecture. We will have a client application program running on the user side and a server application which runs on the server. All clients must first register themselves with the server with a username, associated password and port number to communicate with others who are connected to that server.

DESIGN

¡ TO BE COMPLETED ¿

IMPLEMENTATION

Our client-server program is written in Python programming language we used is python. The code contains 2 files, client.py which is the client source code, and server.py which is the server source code. These are explained in detail below:

Client Implementation

Each client requires a unique username and a password. User needs to pass their username and password along with the server ip address when starting the client. The usage is python client.py username password server-ip".

When a user starts the client, it chooses a random port number and starts listening on that port, waiting to receive any data from the server. Then the client sends its own IP address, the random port number, username and password to the server on the given server IP address on port number 5005 (which the server is listening on). If it successfully connected to the server, then user receives a message Connection Successful. If the user connects using an already existing username but wrong password, then user receives the message Connection Failed: User exists, wrong password used. Please reconnect. If the user does not receive any data, this means that the server is down or wrong server IP address was used. All this is received by the client on the random port number created earlier. All further communication (messages, acknowledgements) from the server is also received on this same port number.

Once the client is connected to the server, the user can send 3 types of messages. When the client types 'list', it receives a list of all the users that are stored on the server except self. When the user wants to chat with 'user1', he/she types 'user1: ¡message¿'. After sending the message, the client receives 2 acknowledgements from the server, one when server receives it and other when receiving client (user1) receives it. If the server is down, then after 3 retry attempts, each with 2 seconds timeout, the client will receive a message Message Not Sent: (msg): Server or Network is down. Finally, 'exit' command exits the client.

The client code uses 2 queues. One queue is gbn queue which has the max size of go-back-n window size (in our implementation, this is 4). Other queue is an unlimited queue containing all the remaining messages. When a user types a message to another user, the message is stored in this gbn queue along with the next sequence number (which is a global variable) unless it is full. The next sequence number is incremented (with modulus window size). If the gbn queue is full, the messages are saved in the other queue.

After storing the message, a new thread is started which tries to send the messages in its gbn queue to the server. Because this is a parallel thread, next upcoming messages can get added to the gbn queue before the previous message is sent. So, in this way, max 4 messages can be sent at the same time (4 is the go-back-n window size we used, but this can be set to any value).

When the client receives an acknowledgement from the server (about the server receiving the message), the client checks if this acknowledgement has the expected sequence. Expected ack sequence is maintained as a global variable. If the ack is as expected, that message is removed from the gbn queue (first message in the queue), expected ack sequence number is updated and next message from the other queue (if not empty) is added to this gbn queue with the next sequence number. If the ack received is not of the expected packet, client just ignores the packet as per Go-Back-N policy.

Whenever a message is sent from the gbn queue of the client to the server, the client waits for a timeout value of 2 seconds. At the end of this, if the expected ack sequence number is different from the sequence number of the message sent, client knows that an ack was received for this message. If the expected ack sequence number is same as the sequence number of the message sent, then the client knows that it did not receive an ack for this message till timeout. This means the packet is lost or server is down. As per Go-Back-N policy, in this scenario, the client resends all the packets in its gbn queue. It also updated the retry count value of the first message in gbn queue. It tries to resend the packets 3 times, after which, if it still did not receive an ack from the server, it will remove all the messages from gbn queue and notify the user that the message could not be sent. When this happens, user needs to retype the message, possibly after some time as the server could be down.

Now, when a client receives a message from the server which is sent by another user, it prints this message to the user and send an ack back to the server like ack:Message Received:gbn:2. As you can see, the ack contains the gbn sequence number received with the message, so that the server can identify the ack. Normal messages and acknowledgements are distinguished based on the content of the message. The server always sends messages/acks in the expected format.

This is how our the client code is implemented. In short, it uses UDP to send and receive messages, uses Go-Back-N policy with window size 4, next sequence number and expected ack number are maintained globally, max 4 messages are stored in gbn queue and rest are stored in another queue and after receiving the ack for expected package, next message from the other queue is pushed into the gbn queue. Parallel threads help in sending and receiving multiple messages at the same time.

Server Implementation

When the server is started, it listens on port 5005 for any incoming messages. At the same time, it starts a parallel thread which keeps checking for any messages in its gbn queue every second and sends the messages in this queue.

The list of clients is maintained in a map. For each client, its details including username, password, IP, port and a list of outgoing messages which are not yet sent are stored in the map. When a client sends its details to the server, server checks if this client is already existing in the map. If not, it saves all the details of this client in the map as mentioned above. If it already exists, server checks if the password is matching. If not, server notifies the client. If yes, server updates the IP and port of the user in the map. If the user has any pending messages sent by other users, server adds these messages to the gbn queue (till the queue is full). The parallel running thread then sends these messages to the user. If the client requests for the list of all the users, the server sends the list of users maintained in this map.

When the server receives a message from 1 client meant for another client, it first stores the message in the receivers queue. It also stores the message in gbn queue along with the sequence number. If the gbn queue is full, then the message is stored in another queue. After this, the server sends an ack back to the sending client with the same sequence number of the message. As the sending part is done by a parallel thread, multiple messages can enter the gbn queue (max 4 = window size).

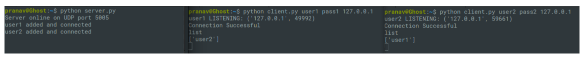
The parallel thread sends messages from the gbn queue to the server with the sequence number. Every time it sends a message, another parallel thread waits for a 2 second timeout. The timeout functionality is similar as in the client implementation explained above. After 2 seconds, if the message sequence number is the same as the expected sequence number, the server knows that the ack is not received and it resends all the messages. Difference with client code is that after 3 retries, the messages are removed from gbn queue and entered into the other queue so that they can be sent again at a later time (when the receiver comes back online).

When the server receives an ack from the receiving client, if the ack number received is not same as the expected ack number, it is ignored as per Go-Back-N policy. If it is same, that message is removed from gbn queue and also from the receiving clients map. After this, the server sends an ack back to the sending client which states that the receiver has received the message. As the gbn queue has an extra slot open now, next message from the other queue is pushed into the gbn queue and the process continues.

This is how our the server code is implemented. It is similar to client code in terms of Go-Back-N implementation.

RESULTS

Test Case 1: 2 clients connect to server and each can see list of other clients.

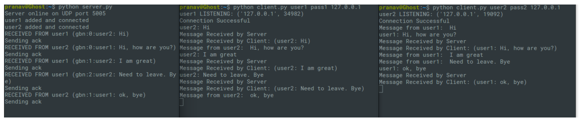


```
pranav@Ghost:~$ python server.py
Server online on UDP port 5005
user1 added and connected
user2 added and connected
```

```
pranav@Ghost:~$ python client.py user1 pass1 127.0.0.1
user1 LISTENING: ('127.0.0.1', 49992)
Connection Successful
list
['user2']
```

```
pranav@Ghost:~$ python client.py user2 pass2 127.0.0.1
user2 LISTENING: ('127.0.0.1', 59661)
Connection Successful
list
['user1']
```

Test Case 2: 2 clients communicate with each other. After sending message, sender gets ack when server receives the message and another ack when receiver receives the message.

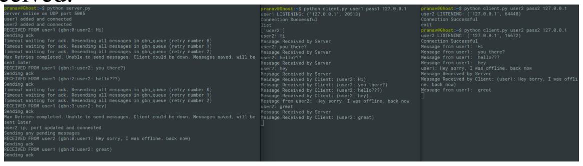


```
pranav@Ghost:~$ python server.py
Server online on UDP port 5005
user1 added and connected
user2 added and connected
RECEIVED FROM user1 (gbn:0:user2: Hi)
Sending ack
RECEIVED FROM user2 (gbn:0:user1: Hi, how are you?)
Sending ack
RECEIVED FROM user1 (gbn:1:user2: I am great)
Sending ack
RECEIVED FROM user1 (gbn:2:user2: Need to leave. By
e)
Sending ack
RECEIVED FROM user2 (gbn:1:user1: ok, bye)
Sending ack
```

```
pranav@Ghost:~$ python client.py user1 pass1 127.0.0.1
user1 LISTENING: ('127.0.0.1', 34982)
Connection Successful
user2: Hi
Message Received by Server
Message Received by Client: (user2: Hi)
Message from user2:  Hi, how are you?
user2: I am great
Message Received by Server
Message Received by Client: (user2: I am great)
user2: Need to leave. Bye
Message Received by Server
Message Received by Client: (user2: Need to leave. Bye)
Message from user2:  ok, bye
```

```
pranav@Ghost:~$ python client.py user2 pass2 127.0.0.1
user2 LISTENING: ('127.0.0.1', 19092)
Connection Successful
Message from user1:  Hi
user1: Hi, how are you?
Message Received by Server
Message Received by Client: (user1: Hi, how are you?)
Message from user1:  I am great
Message from user1:  Need to leave. Bye
user1: ok, bye
Message Received by Server
Message Received by Client: (user1: ok, bye)
```

Test Case 3 and 4: 1 client goes offline (or disconnected or having network issues). Other client sends messages and gets ack that messages are received by server. Server tries to send messages to receiver till timeout (3 times). Then all messages from gbn queue are dropped and saved on server. When receiver comes back online, it receives all the pending messages and sender gets acknowledgement that all the messages are received by the received.



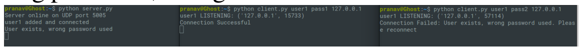
```
pranav@Ghost:~$ python server.py
Server online on UDP port 5005
user1 added and connected
user2 added and connected
RECEIVED FROM user1 (gbn:0:user2: Hi)
Sending ack
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 0)
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 1)
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 2)
Max Retries completed. Unable to send messages. Client could be down. Messages saved, will be
sent later
RECEIVED FROM user1 (gbn:1:user2: you there?)
Sending ack
RECEIVED FROM user1 (gbn:2:user2: hello???)
Sending ack
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 0)
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 1)
Timeout waiting for ack. Resending all messages in gbn_queue (retry number 2)
RECEIVED FROM user1 (gbn:3:user2: hey)
Sending ack
Max Retries completed. Unable to send messages. Client could be down. Messages saved, will be
sent later
user2 ip, port updated and connected
Sending any pending messages
RECEIVED FROM user2 (gbn:0:user1: Hey sorry, I was offline. back now)
Sending ack
RECEIVED FROM user1 (gbn:0:user2: great)
Sending ack
```

```
pranav@Ghost:~$ python client.py user1 pass1 127.0.0.1
user1 LISTENING: ('127.0.0.1', 20513)
Connection Successful
list
['user2']
user2: Hi
Message Received by Server
user2: you there?
Message Received by Server
user2: hello???
Message Received by Server
user2: hey
Message Received by Server
Message Received by Client: (user2: Hi)
Message Received by Client: (user2: you there?)
Message Received by Client: (user2: hello???)
Message Received by Client: (user2: hey)
Message from user2:  Hey sorry, I was offline. back now
user2: great
Message Received by Server
Message Received by Client: (user2: great)
```

```
pranav@Ghost:~$ python client.py user2 pass2 127.0.0.1
user2 LISTENING: ('127.0.0.1', 64448)
Connection Successful
exit
pranav@Ghost:~$ python client.py user2 pass2 127.0.0.1
user2 LISTENING: ('127.0.0.1', 16672)
Connection Successful
Message from user1:  Hi
Message from user1:  you there?
Message from user1:  hello???
Message from user1:  hey
user1: Hey sorry, I was offline. back now
Message Received by Server
Message Received by Client: (user1: Hey sorry, I was offli
ne. back now)
Message from user1:  great
```

Test Case 5: Server goes down. 1 client tries to send message to another client. 1st Client is notified that the messages could not be sent.

Additional Test Case: If user tries to login with wrong password or a different user uses existing username with wrong password, user gets Connection Failed notification.



```
pranav@Ghost:~$ python server.py
Server online on UDP port 5005
user1 added and connected
User exists, wrong password used
```

```
pranav@Ghost:~$ python client.py user1 pass1 127.0.0.1
user1 LISTENING: ('127.0.0.1', 15733)
Connection Successful
```

```
pranav@Ghost:~$ python client.py user1 pass2 127.0.0.1
user1 LISTENING: ('127.0.0.1', 57114)
Connection Failed: User exists, wrong password used. Pleas
e reconnect
```