# Project 1: Chat Application

## Project Description

A chat program is an application that enables communication between multiple chat clients. There are two typical paradigms to enable such communications. For the client-server paradigm, it is the responsibility of the server to relay the messages that are exchanged between clients. On the other hand, peer-to-peer (P2P) paradigm enables the application to be symmetric, where there is no dedicated server to respond to client's service requests. Instead, every participant serves as both server and client.

For this project, you need to develop a UDP-based chat application, called ***netchat***, following the P2P communication paradigm. This application will use sockets for communications. Each functionality of this chat application is described below.

### 1. start a chat session



The chat program is invoked from command-line with a single port parameter, e.g.,



```
$ ./netchat 5555
```

The port number specifies the port that *netchat* will use to communicate with others using UDP. It will bind itself to this port when *netchat* runs. The screenshot after invoking the program is below:



```
eecs325@eecs325-VirtualBox:~/s2020-proj1/$ ./netchat 5555
#chat with?:
```

Then the application waits for user input from keyboard. The user could initiate a chat session with another user by typing their IP address (dotted decimal notation) and port number. For example,

```
#chat with?: 127.0.0.1 6666
```

*netchat* then will wait for the other party to respond to this connection request. Upon success, the sender will see the below message on his screen:

```
#success: 127.0.0.1 59051
```

Note that the response will be sent back from a different port other than `6666` . This UDP-based application is designed to behave like a TCP-based application. (You

could review the TCP socket programming example on the slide to learn how TCP handles connection requests). *netchat* should implement a timeout mechanism to enable request retransmissions if the sender could not receive any response from the receiver within 5 seconds. *netchat* will give up after two retransmissions and notifies the sender of the failed connection, e.g.,

```
#failure: 127.0.0.1 6666
```

On the receiver end, the client will receive 3 notifications of the connection initiation requests, at 5-second intervals, so that the receiver could decide whether to establish a connection with the sender or not, e.g.,

```
#session request from: 127.0.0.1 55833
#chat with?:
#session request from: 127.0.0.1 55833
#chat with?:
#session request from: 127.0.0.1 55833
```



Note that the connection request does not come from port 5555. If the receiver decides to accept the connection, he could type the IP address and the port number of the connection request:



```
#session request from: 127.0.0.1 41625
#chat with?: 127.0.0.1 41625
#session request from: 127.0.0.1 41625
#chat with?:
[127.0.0.1:41625] accept request? (y/n)y
```



*netchat* will ask for a confirmation for accepting the connection request. The receiver will type `'y'` to accept the connection request, and `'n'` to deny it. Once a session is established between the sender and receiver, they could start exchanging messages with each other.

## 2. interaction for established session

During the session, sender/receiver need to handle three types of events: 1. message events 2. time out event 3. signal events, which will be explained following.

### 2.1 Message Events

Message events are the message packets being exchanged between the sender and receiver. *netchat* will read **up to 50-byte-long** message from user's input (excluding `'\n'`, i.e., ENTER key), and send the message to the receiver via UDP.

The communication between a sender and receiver is asynchronous, i.e., the receiver could decide to respond to the sender later. For example,

Sender:

```
#success: 127.0.0.1 34048
#chat with?: 127.0.0.1 34048
[127.0.0.1:34048] your message: hello
#chat with?:
```

Receiver:

```
[127.0.0.1:41625] accept request? (y/n)y
#chat with?:
#[127.0.0.1 41625] sent msg: hello
```

If the receiver decided to respond, the receiver need to specify the sender's IP and port before he types the message from keyboard:

```
#chat with?: 127.0.0.1 41625
[127.0.0.1:41625] your message: hi
```



The sender will receive:



```
#chat with?:
#[127.0.0.1 34048] sent msg: hi
```



As input and messages are asynchronous, interleaving of output to `stdout` may occur as below:

```
#session request from: 127.0.0.1 50011
#chat with?: 127.0.0.1
#session request from: 127.0.0.1 50011
#chat with?: 50011
[127.0.0.1:50011] accept request? (y/n)
```

- **bonus points (2 pts):** While waiting for user input from keyboard (i.e., waiting for accepting connection request or typing messages from keyboard), you could set up a timeout value, e.g., 5 seconds, to avoid being blocked. For example:

```
#session request from: 127.0.0.1 40416
#chat with?: 127.0.0.1 40416
[127.0.0.1:40416] accept request? (y/n)
#chat with?:
```

In this example, the application accepts new commands as the client did not accept or deny the connection request within 5 seconds.

### 2.2 Timeout Events

As mentioned in Section 1, *netchat* should implement timeout and retransmission mechanisms when a sender is trying to establish connections with a receiver. In addition, since UDP is "connectionless", *netchat* also should implement a heartbeat mechanism to keep track of the connection status with the other party. For example, if a sender could not receive a heartbeat message within 5 seconds for 3 consecutive epochs, the sender will assume that the session has been terminated by the receiver. Then a warning will show up on the screen and the sender will also terminates the session:

```
[127.0.0.1:45435] accept request? (y/n)y
#warning: peer 127.0.0.1 45435 might be offline; terminating
session...
#chat with?:
```

In the example above, the receiver terminates all the sessions with CTRL-C (i.e., control key and C).



### 2.3 Signal Events



The user may want to terminate an established session or terminate all sessions and quit the application. To terminate all sessions and quit the application, users could use CTRL-C, which raises the SIGINT signal:



```
#chat with?: ^C
terminating all sessions...
```

Note that this termination only closes all the sessions on the sender side, the other side of the sessions will figure this out with the heartbeat mechanism mentioned in Section 2.2. To terminate one session, users could use CTRL-\ (i.e., control key and backslash), which raises SIGQUIT signal. Then the user needs to specify the IP and port number of the session that he wants to be terminated:

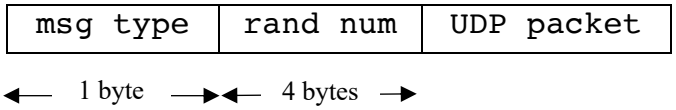
```
#chat with?: ^\
#terminate session (for help <h>): 127.0.0.1 55097
#terminating session with 127.0.0.1 55097
#chat with?:
```

Note that `netchat` will send a message to the other party to notify this termination such that the receiver could also terminate this session:

```
#[127.0.0.1:54863] session termination received
#chat with?:
```

## 3. `netchat` protocol format

The `netchat` packets have the following formats:




`msg type` specifies the type of the message being exchanged, e.g., normal message or heartbeat message; `rand num` is a random number generated by the session initiator. This value could be used as a simple authentication mechanism for the receiver to verify that this message is sent from the actual sender (although `netchat` has not implement encryption yet); then the UDP packet is encapsulated in this packet header.



## 4. Bonus Points(2 pts)



There are two options for gaining this bonus points. 1) implement a menu for `netchat`, that could list all the active sessions (that it knows of) for users; 2) If you are familiar with GUI programming using C, you could try to address the interleaving problem of the shared `stdout` file descriptor that could produce messy output.



For this project, you will be provided with the executable `netchat`. You could use that as a reference when you implement your own application.

## REMINDERS

- Submit your code to Canvas in a tarball file by running command in terminal:
  `tar -zcf [you-case-id]-proj-1.tar.gz project-1`
- If your code does not **compile** or could not **run**, you will not get credits.
- Document your code (by inserting comments in your code)
- DUE: 11:59 pm, Monday, Feb 24th.