Assignment 2: Chat Application
Due: 11:59pm, October 10th
Lead TA: Ali Musa Iftikhar

In this assignment, you will build a chat application based on the client-server model. All clients will communicate with each other through the server i.e., the server will forward message from one client to the other. The server should be able to handle multiple clients at the same time. To achieve this, you MUST use the *select* system call (rather than threads). (See Piazza for references on how *select* works). Your application must be able to handle the arrival and departure of clients (both graceful exit as well as uninformed).

You will submit only the server as a2.c. All of the required functionality will be in the server but you will also need to write a client which can effectively test the server. All communication will be based on TCP.

Details:

Bootstrapping: The server should take a command line argument for the port on which to listen.

The server maintains a record of all the ClientIDs currently participating along with their appropriate status fields (Active or Inactive). It should be able to handle at least four types of messages from the clients (the message formats are described later).

1. A HELLO message (clients tell the server that they are here).
   a. This is used by new clients as well as those that were temporarily disconnected.
   b. When the server receives a HELLO message, it will register the client by keeping track of its ClientID and send back a HELLO_ACK message. Additionally it will also send a CLIENT_LIST message which contains the information about existing clients.
   c. If a ClientID already exists and the client is active, the server responds with an ERROR(CLIENT_ALREADY_PRESENT) message and closes the TCP connection.
   d. If a ClientID already exists and the client is inactive (case of client returning from a disconnection), server sets the client status to active and responds with a HELLO_ACK and CLIENT_LIST as in (b).

2. LIST_REQUEST message
   a. A client can also explicitly request the list of active clients by sending a LIST_REQUEST message. The server will respond through the CLIENT_LIST message.

3. A CHAT message (from one client to the other).
    a. Clients can send messages to other clients by sending a CHAT message to the server.
    b. The server will forward this message to the intended recipient client in another CHAT message.
    c. If the recipient client doesn't exist or is inactive (disconnected), the server discards the message and sends an ERROR(CANNOT_DELIVER) message, which has the Message-ID appropriately set, back to the sender.

4. An EXIT message:
    a. Signals that the client is leaving. In this case the server removes the client from its record.

Message Format:

Every message has a header followed by an **optional** data part.
The header has the following format:

1) Type: short int (2 bytes)
   Values defined later.
2) source: char[20], null terminated string specifying the source. message *originating* from server uses the id *"Server"*
3) destination: char[20], its semantics are the same as the source field. Message destined to the server should have the id "Server".
4) Length: int (4 bytes), specifies the length of the data part. Value should be zero if there is no data part.
5) Message-ID: int (4 bytes), this is set by the client for every chat message. The server echoes it back in case the message wasn't delivered i.e., ERROR(CANNOT_DELIVER) scenario. This allows the client to identify which message wasn't delivered (*in case* the client wants to retransmit). The default value is zero and for chat message a value >= 1 should be used.

Data part of any packet can be at most 400 bytes. For the CLIENT_LIST message, the data part will contain null terminated strings of ClientIDs (assume that this message will contain the list of all the active clients and will not exceed 400 bytes). For CHAT messages, the data will contain the actual chat message.

Types:
1 - HELLO
2 - HELLO_ACK
3 - LIST_REQUEST
4 - CLIENT_LIST
5 - CHAT
6 - EXIT
7 - ERROR(CLIENT_ALREADY_PRESENT)
8 - ERROR(CANNOT_DELIVER)
Please strictly follow the message format as diagrammatically specified below (the sequence of the fields and type is **critical**). Note: NULL means no data, so in this case your message will be shorter than the message with data.

Type	Source	Destination	Length	Msg ID	Data
1	ClientID	Server	0	0	NULL

Type	Source	Destination	Length	Msg ID	Data
2	Server	ClientID	0	0	NULL

Type	Source	Destination	Length	Msg ID	Data
3	ClientID	Server	0	0	NULL

Type	Source	Destination	Length	Msg ID	Data
4	Server	ClientID	<size of data>	0	<data>

Type	Source	Destination	Length	Msg ID	Data
5	ClientID	RecipientID	<size of data>	>=1	<data>

Type	Source	Destination	Length	Msg ID	Data
6	ClientID	Server	0	0	NULL

Type	Source	Destination	Length	Msg ID	Data
7	Server	ClientID	0	0	NULL

Type	Source	Destination	Length	Msg ID	Data
8	Server	ClientID	0	>=1	NULL

Error Handling:

The server should be robust enough to handle all sorts of errors not mentioned above. It should do this by closing the socket and removing the client from its record (unless otherwise specified), if it is present.
Some examples of errors or unexpected behavior:

- Client arbitrarily closing the socket (this should be treated as a temporary disconnection, and its client record must not be removed)
- A new client sends a CHAT message instead of HELLO.
- A CHAT message without a destination or client trying to chat with herself.
- Data part of a message is not equal to the bytes specified.

Submission Instructions:

Same as assignment 1 except you need to name your server file a2.c

Additional Information:

- For any data types that are > 1 byte (e.g., short, int, etc) it is important to convert from host to network byte ordering and the other way around. For your application data (e.g., some of the header fields in the assignment), it is a *good practice* to convert the above data types to network order before you send them over the network and convert them back on the receiver side to the host order. this ensures that your code can run on any machine. for the assignment, you MUST do these conversions wherever it is required.
- one way to deal with the messages is to create a struct with appropriate fields in there. when sending (write call) you cast the struct pointer to char *, on the other side, you convert the char * back to struct *.

  if you use structs, you must turn off padding, so that the message format is exactly similar to what is specified in the assignment. here is how you can turn off padding:

  Example:

```
struct __attribute__((__packed__)) header {
unsigned short a;
char b[20];
char c[20];
unsigned int d;
unsigned int e;
};
```