Client-Server Chat Application

PREFACE

designed around using the TCP transport protocol. Injecting networking calls into a real time application can be detrimental to its performance in multiple ways. For one, networking API calls are usually blocking by default. This is designed around the asynchronous behavior of your computer network devices running alongside the majority of your program on the CPU.

In summary, if you request a networking operation to happen, the active thread that the call is being made on will block until the conditions are fulfilled or fail for whatever reason. If you already had a single threaded program and you were to add blocking related networking between some of the logic, the program would only be as responsive as the networking performance. If the program requests data, it cannot continue until the data it requested arrives. This is why a simple fix for this problem can be to make the network calls not block, however this introduces a problem from the opposite angle. If a networking call was blocking, it is sure to deliver the request as soon as a low-level interrupt resumes your application thread when delivering the results of the request to your application. When the network call blocking is turned off, if the network call request can't be completed, then the control is instead returned back to the application making the call. The application can then go about processing any other logic that needs to be done such as updates and real time rendering calls. The other angle issue here is that every time an incomplete net operation happens, it has to wait for other processing before it can attempt to make a request again. This is problematic for software that wants to show the results of networking as soon as it can. A real time networked program using non-blocking TCP calls would also have to split up partial messaging on a frame to frame basis, because doing the full partial message loop described to you so far would almost be as detrimental as blocking, causing unstable frames and an inconsistent flow of the other real time operations.

The solution to these problems you might have guessed is to place the networking calls on separate threads and retain their ability to block. In fact, blocking is good when dealing with other threads, because they are de-scheduled when the blocking conditions aren't met. This allows the CPU to focus performance to other threads and processes running. When any network condition is met, any operations that need to interface with the unhinged software running on the original thread can be immediately instigated.

The TCP transport protocol provides guaranteed in order delivery of the data. This is often important for certain types of data in games such as communication related operations like chat functionality. The UDP transport protocol doesn't hold these features over the data transport which can cause performance gain provided guaranteed in order delivery is not required. If UDP was being used and data needed to be guaranteed, your application layer would need to provide the understanding of this and potentially handle extra network operations to get the required data. Doing so can sometimes enact a performance degradation greater than the built in TCP protocol being performed at a lower level, especially during times of unstable network conditions, when networking that requires guarantees will need to play catch up.

This lab is also designed to give an insight when attempting to choose the right transport related protocols depending on the networking problem. TCP is chosen because chat messages should not be dropped and should be delivered in order.

OBJECTIVES

- Implement TCP and UDP sockets in a single application
- Demonstrate an advanced knowledge in winsock
- Revive programming knowledge in C++
- Apply project and time management

PROJECT OVERVIEW

You will be creating two console applications for the client and server.

The server program should advertise the TCP information over a **UDP connection**, and connect the clients to chat and exchange messages over a **TCP connection**.

Every 1 second, the server broadcasts an advertisement message to the entire network over a UDP socket.

Over the TCP connection, the server should respond to commands, and echo chat messages back to all connected clients.

The server should also perform the following actions:

- **Display the clients' actions** (connect, disconnect, register, get list, and get log). For example, if a client has successfully registered with the server, a message indicating that "<user x> has joined the chat!", is displayed on the server console.
- **Log all the commands and chat messages** exchanged by the clients in a .txt file.

The client program shall perform the following actions:

- Listen on a UDP connection to read the server's TCP information
- Prompt the user to choose a username
- Establish a TCP connection with the server
- Run commands $register, $exit, $getlist, and $getlog
- Send and receive chat messages
- Display the messages echoed by the server onto the console

Note: the user can run the command $exit without being registered with the server. The rest of commands as well as the chat messages are only permitted after successful registration.

PROJECT IMPLEMENTATION

To simplify the implementation of the requirements, the project is divided into two phases:

Phase 1: implementation of TCP socket

Phase 2: implementation of UDP socket

Phase 1: Chat over a TCP connection (week 2 – Week 3)

Assumption:
the information required to create a TCP socket (IP address and port number of the TCP server) is available and entered by the user via the stdin (cin, getline, etc.)

Steps for the client program:

1. Prompt the user for the server's IP address and port number (this step will be replaced by the information received on the UDP connection in phase 2)
2. Create a TCP socket
3. Connect to the server using the information provided in step 1
4. Prompt for a username
5. Call the command $register. The username must be automatically amended to the command before being sent. E.g. $register <username>
6. Receive the server response on the socket. If sv_full, close the socket and exit program
7. If sv_success, the client can execute commands and send chat messages.

Steps for the Server program:

1. Create the TCP listening socket
2. Bind the socket
3. Set the server in listening mode
4. Create the file descriptor sets master set and ready set to multiplex the connected and connecting clients.
5. Call select and return after timeout 1 second (NB: if timeout is null, select will note return unless

```
int rc = select(0, &readySet, NULL, NULL, &timeout)
```

6. Check if the listening socket is SET in the ready set using the macro FD_ISSET(). If so, accept the connection, add the client to the list of connected clients[1], and add the new socket to the master set.

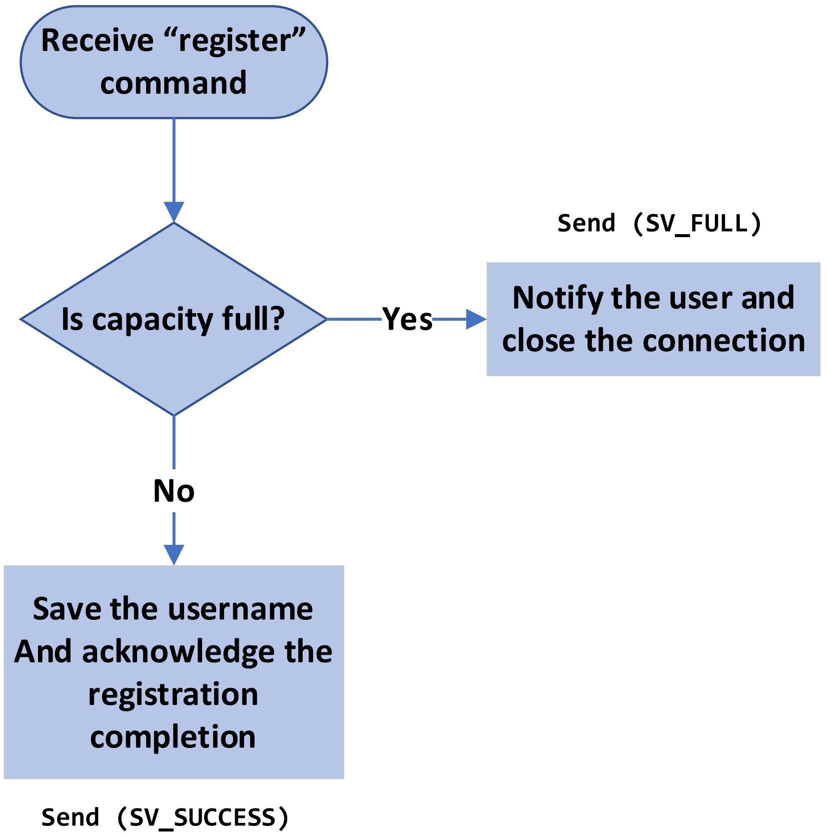
[1] To keep track of the registered clients, you should store the clients' information (username and socket number) in a data structure. Possible solutions include: array, vector or list* of a struct {username, socket}, or a map<SOCKET, string>

7. Run through the ready set and recv data from the socket. Make sure to check if the current socket is the listening socket before you call recv().

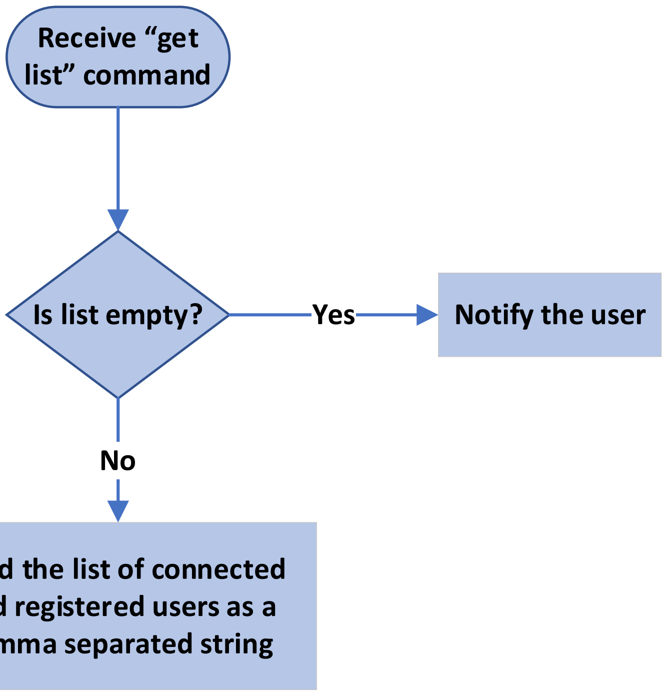
```
typedef struct fd_set
{
  u_int  fd_count;
  SOCKET fd_array[FD_SETSIZE];
}

fd_count: The number of sockets in the set.
fd_array: An array of sockets that are in the set.
```

8. Parse the received message, and perform the correspondent action(s):
    - **Handle the $register command**: check the chat capacity, send an SV_FULL message if full, or SV_SUCCESS otherwise. In case of SV_FULL, close the socket.


Receive "register" command
Is capacity full?
Yes
Send (SV_FULL)
Notify the user and close the connection
No
Save the username And acknowledge the registration completion
Send (SV_SUCCESS)

- **Handle a user request for the list of connected users ($getlist)**


Receive "get list" command
Is list empty?
Yes
Notify the user
No
Send the list of connected and registered users as a comma separated string

- **Handle a user request for the activity log ($getlog)**

  All commands and messages of all users should be saved on the server in a log file. Upon receiving a get log request, the server should send the file to the corresponding client.

  Server:
  1. Open file on the server
  2. Send length of the file first to tell the client how much data to receive.
  3. Read line
  4. Send line
  5. Repeat 3 and 4 until end of file
  6. Close file

  On the client side:

  1. Receive length
  2. Open a new file
  3. Read from socket until all bytes are received
  4. Write to file after every recv() call
  5. Close file

- **Handle the user exit ($exit)**
    - Graceful disconnection is invoked by sending the ***"$exit"*** command, which should remove the user from the list, and close the socket.
    - In case of ungraceful disconnection, e.g. The user closes the window without executing the exit command, the server should remove the user from the list and close the connection.

Notes:

- All messages and commands received from the clients must be logged in a .txt file. The following function could be used to open a file locally:

```
FILE *fopen(const char *filename, const char *mode)
When initializing the server, open the file in write mode
"w" to override it if already created. In the server run,
the function should be called in append mode "a"
```

- Error check must be implemented at every step in the programs to ensure that all possible errors are handled properly.

Phase 2: UDP broadcast (4 days - week 4)

The server should broadcast every 1 second its address and port number for the TCP connection.

Clients receiving the broadcast message shall use the address and port information to establish a TCP connection with the server.

Tips for the server implementation:

- Create a UDP socket
- Enable the SO_BROADCAST option
- Bind the socket
- Add the socket to the master set
- Form the broadcast address in the sockaddr_in structure. Set the in_addr to the network broadcast address (e.g. 192.168.x.x).
- Send a broadcast message on the UDP socket advertising the server's IP address and its port number. Given that the timeout

for the server select() has already been set to 1 second in phase 1, you can call the broadcast function right after returning from select.

Tips for the client implementation:

- Create the UDP socket
- Bind the socket to address INADDR_ANY
- Enable the SO_BROADCAST option
- Set the socket to non-blocking
- Add the socket to the master set
- Call select with timeout 1 sec
- Call recvfrom on UDP socket after returning from select. Make sure you handle the WSAEWOULDBLOCK error for the recvfrom()

Remark:

- SO_REUSEADDR[2] should be enabled to allow a socket to forcibly bind to a port in use by another socket. The second socket calls setsockopt with the optname parameter set to SO_REUSEADDR and the optval parameter set to a boolean value of TRUE before calling bind on the same port as the original socket.

- SO_BROADCAST should be enabled to allow incoming and outgoing broadcast to and from a socket