COMP3331 - Major Assignment

Jarrod Cameron (z5210220)

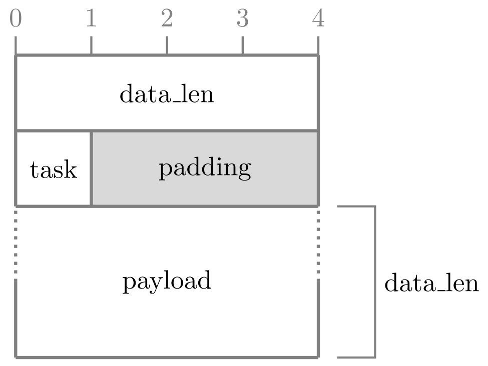
November 2019

Contents

1 Design Choices

The client and server programs are both written in **C** using **gcc** as the compiler.

All messages transmitted are refereed as **payload**'s. The content of the payload is determined by the type of payload being sent. Each payload has an eight byte header. The header and each payload is a multiple of four bytes long. The reason for this is gcc pads *C structs* with bytes until they are multiples of four which is optimised for the CPU. Each header and payload is sent together. The format can be seen below:



- The **data_len** is the length of the payload that is attached to the header.
- The **task** is an identifier which indicates the task to be performed (e.g. logging out). The three types of tasks are:
  - Identifiers that start with **client** are sent by the client to the server.
  - Identifiers that start with **server** are sent by the server to the client.
  - Identifiers that start with **ptop** (**p**eer **to** **p**eer) are sent from one client to another client.
- The **padding** ignored.
- The **payload** is defined in **header.h**. This is the data being transmitted (e.g. The name of the user for the *whoelse* command.

2 Internal Implementation

The server listens on the port number given via the command line. Each time a client connects to this port the server spawns a new thread to handle this client. Although this is not the most scalable solution this has the upside of simplifying the server's connection handling code. All data structures contain locks to prevent any race conditions.

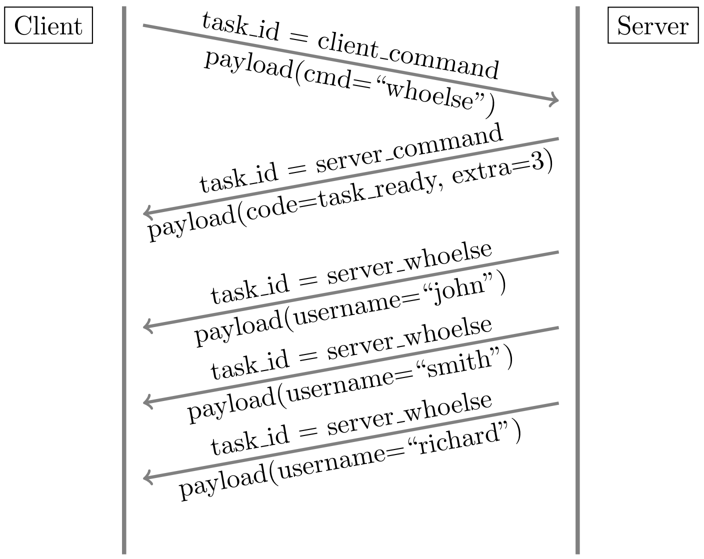
Each client only has one thread. The thread listens to incoming payloads from the server, standard input, incoming connections from clients, and payloads sent from clients.

3 Client to Server

All messages from the client to the server are initiated by the operator (i.e. the user) and are given by standard input. The task identifier is always **client_command**. The command itself is send in plain text to the server. Sending the entire message means that all the code to parse and evaluate the command can be stored on the server and thus keeping the code of the client cleaner. This also means that the server will expect same task identifier from the client when receiving a command, this reduces the possibility of receiving the wrong commands in the wrong order.

Once the client sends the message to the server the server must reply with task identifier **server_command**. The payload will contain a *status code* (defined in **status.h**. The status code should be **task_ready** to indicate the command is about to be satisfied. This payload contains an **extra** 64 bit field which may be used for the command. This is the initial handshake used for every communication from the client to the server.

After the hand shake the client and server will share data appropriately. For example the case of the *whoelse* command can be seen below:

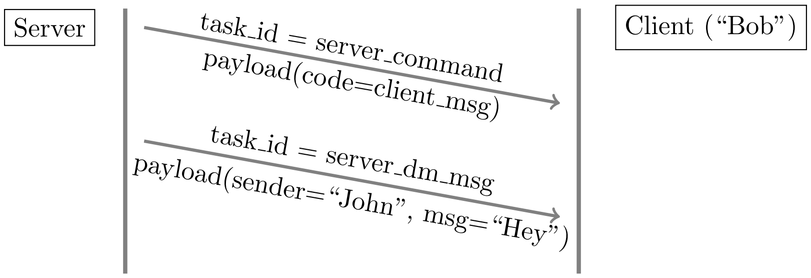


For the *whoelse* case the *extra* variable is used to indicate to the client the number of users. Here the three users are "john", "smith", and "richard".

4 Server to Client

Most payloads sent from the server to the client are the result of another client (e.g. user "A" sends a message to the server which is forwarded to user "B"). The only payload that the server sends to the client without being initiated by another client is the time out message (for when the user has been inactive for too long and is logged out by the server).

When the server sends a payload to the client the first task identifier is always **server_command**. This allows the parsing of payloads simpler on the client side (since the first identifier is always server_command). The server_command payload will contain a code which indicates the payload that is coming next. An example of a server sending the initial payload to the client can be seen below. Below is the result of user "John" sending the message "Hey" to "Bob".



5 Client to Client

When sending data from the client to another client the command given (via standard input) is the payload that is sent to the receiving client as plain text (just like when the client sent the first message to the server). Since all of the payloads between peers are simple a single payload is all that is needed. Sending a payload from one client to another requires sending the task identifier, **ptop_command**, which will contain the command entered in plain text.

Since initialising the connection is different to the average use case (sending private messages and closing the connection) the task identifier is **ptop_handshake** is used to send the connecting user's username and not the command given.

The process of starting a connection is not simple. The steps are outlined below:

1. Wait for a command via standard input.
2. If the command is not "startprivate <user>" handle the command and go to step 1.
3. Check if the connection is established. If the connection is already established go to step 1.
4. Query the server for the desired client's port number and IP address. If the query could not be satisfied (e.g. blocked by user, invalid user, user is not online, etc) return to step 1.
5. Establish connection with the peer to start the private session.
6. Once a connection is established send the user name (of the client that started the connection) using the **ptop_handshake** identifier.
7. Update appropriate data structures and return to step 1.

6 Improvements

- The server doesn't spawn a thread for each connection. A single thread could be used for receiving and sending payloads but is highly likely to become a bottle neck. However, two threads would be more efficient. This allows one thread to handle all IO requests (receiving payloads, sending payloads, establishing connections) and one thread to handle the computation of servicing these requests.
- When the client sends a message to the server this requires the server to acknowledge back often providing no useful information. The advantage of this is that it reduces the possibility of sending payloads in the incorrect order. The downside is data is be transmitted for no good reason.
- Each of the payloads is statically sized. This requires sending more than one payload when all data can't be stored in the single payload. This makes it easier to parse each of the payloads but has the downside of sending extra payloads. The most obvious example of this is the "whoelse" command where the server sends each name one payload at a time instead of a dynamically sized payload depending on the number of users.
- When the client sends their first payload for a command (whether it is to the server or another peer) the command is sent in plain text (similar to the HTTP protocol which inspired this design). The means a large payload is needed to convey a small amount of information which could be stored in a few bytes.

7 Miscellaneous

7.1 Directory Structure

The root directory contains two sub-directories, **include/** and **src/**, which contains the header and source files respectively. The credentials file needs to be stored in the root directory with the name “credentials.txt”. To use a different credentials file update the macro “CRED_LIST” in **config.h**. There should also be a make file, “Makefile”, and a run script, “run.sh”. In the root directory.

7.2 Compilation

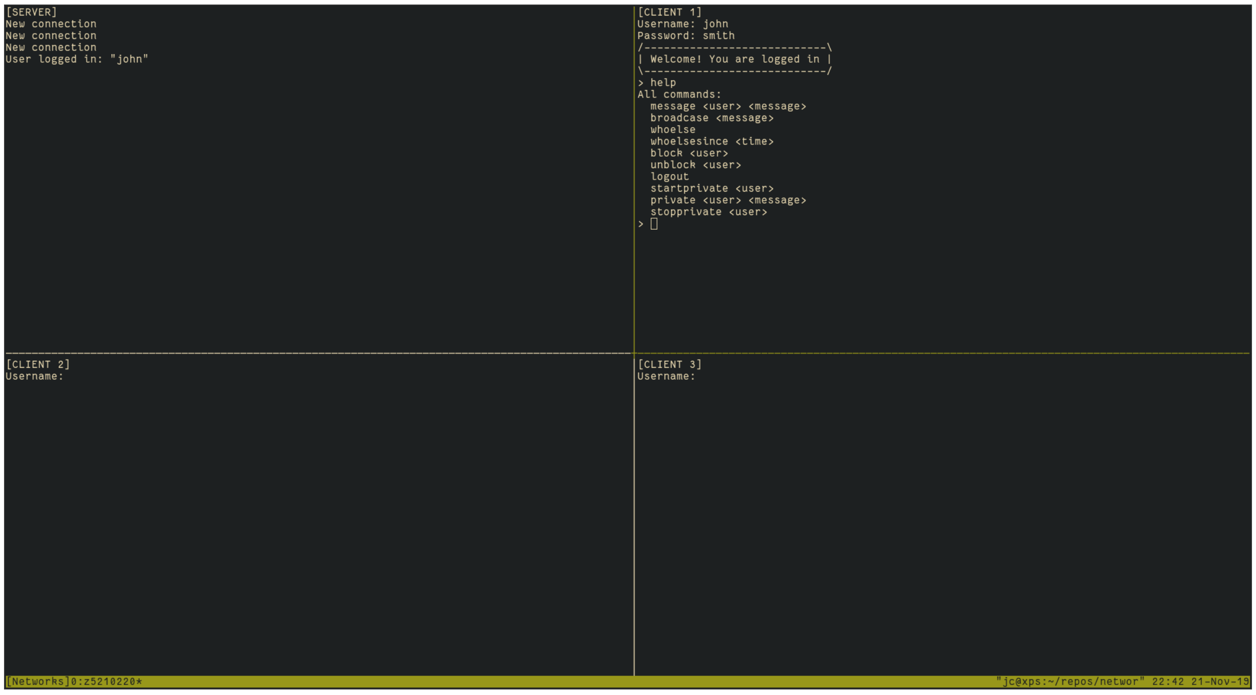
To compile the program run **make** in the root directory. The “client” and “server” binary will be placed in the root directory. A new directory will be created named **build/** which contains the object files. To clean the directory run **make clean**.

7.3 Run script

A run script, “run.sh”, is provided and can be run using the command **bash run.sh**. This will compile and spawn three clients and a server in a **tmux** session.

The server’s port is chosen by scanning a list of open ports and using the first closed port. The block duration and timeout are set as constants in the script. The block duration is defined as “BLOCK_DURATION” and the timeout is set as “TIMEOUT”. The IP address used is the local host address “127.0.0.1”.

The result of running the run script is seen below after the user “john” logs in:



```
[SERVER]
New connection
New connection
New connection
User logged in: "john"
```

```
[CLIENT 1]
Username: john
Password: smith
/----------------------------\
| Welcome! You are logged in |
\----------------------------/
> help
All commands:
  message <user> <message>
  broadcase <message>
  whoelse
  whoelsesince <time>
  block <user>
  unblock <user>
  logout
  startprivate <user>
  private <user> <message>
  stopprivate <user>
> 
```

```
[CLIENT 2]
Username:
```

```
[CLIENT 3]
Username:
```

```
[Networks]0:z5210220*                    "jc@xps:~/repos/networ" 22:42 21-Nov-19
```