ECEC 353
Programming assignment 2

Lucas David
ld492@drexel.edu

CHAT ARCHITECTURE

Messages (message.c)

A structure ChatMessage was defined to promote a way to communicate between server and clients.

```
typedef struct
{
    char username[MAX_USERNAME_SIZE];
    char message [MAX_MESSAGE_SIZE];

} ChatMessage;
```

Functions were also defined to operate over this struct.

```
ChatMessage ReadChatMessage ( int _pipe )
void SendChatMessage        ( int _pipe, ChatMessage _m )
void PrintChatMessage       ( ChatMessage _m )
```

Server side (server.c)

Server defines variables and functions to manage clients connections and messages sending between clients. The signal ctrl+c will stop the server, removing all the pipes opened.

What concerns the execution, the server program creates a thread to concurrently execute the function HandleNewConnections and then starts to handle messages from the clients thorugh the pipe CLIENT_SENDER. When a message is private – private messages always start with @<<username>> –, the server extracts the username of the message and find the correspondent communication channel for that user. Then, it sends the message for that user only. If a message is public, the message will be sent to all connected users.

To connect to the server, a client always send a message with CONNECTION_OPEN_TOKEN to the server, using the pipe CONNECTION_REQUESTER. The server will verify if the client can be connected (the number of clients connected at the same time cannot be higher than MAX_CONNECTIONS_ALLOWED) and then it will answer the client with CONNECTION_OPEN_SUCCESS or CONNECTION_OPEN_FAILURE through the pipe CONNECTION_PROVIDER.

Client side (client.c)

The client starts its execution asking the server if it can join to the chat. If the answer is negative, the client will print a message error and close. If otherwise, the client starts a thread to handle ClientReceiver function and then wait for the user to insert a message.

When the user types a message, the client program send this message thorugh CLIENT_SENDER pipe.

`ClientReceiver` will listen to the pipe “/tmp/ldlink” + *username*, which is the pipe used by the server to send messages to the user *username*.

CHALLENGES FACED

The variable `*connectedUsersCount` defines how many pipes inside the vector `sockets[MAX_CONNECTIONS_ALLOWED]` are valid. Although `*connectedUsersCount` is modified only by a single thread, it is read by two execution lines. Therefore, it has to be a shared variable. At first, the program was design to use the `fork ()` call, creating two different processes. However, after an unsuccessful period without positive result trying to use shared area memory, it was decided that it would be faster to change the program to address the problem with threads, instead of processes.

Chat messages were sent to all users, including the user who wrote it. This was easily solved with the following statement:

```
if ( GetClientIndex (current.username) == i )
       continue;
```

`read()` call did not block the execution of the thread when pipe was not open for write by another process. To solve this, the pipe is open and closed every time when a message must be read.