System Architectures

Centralized Chat System

NGUYEN Duc Tung
ICT.M7.003

TRAN Giang Son - DANIEL Hagimont
March 26th, 2018

1 Introduction

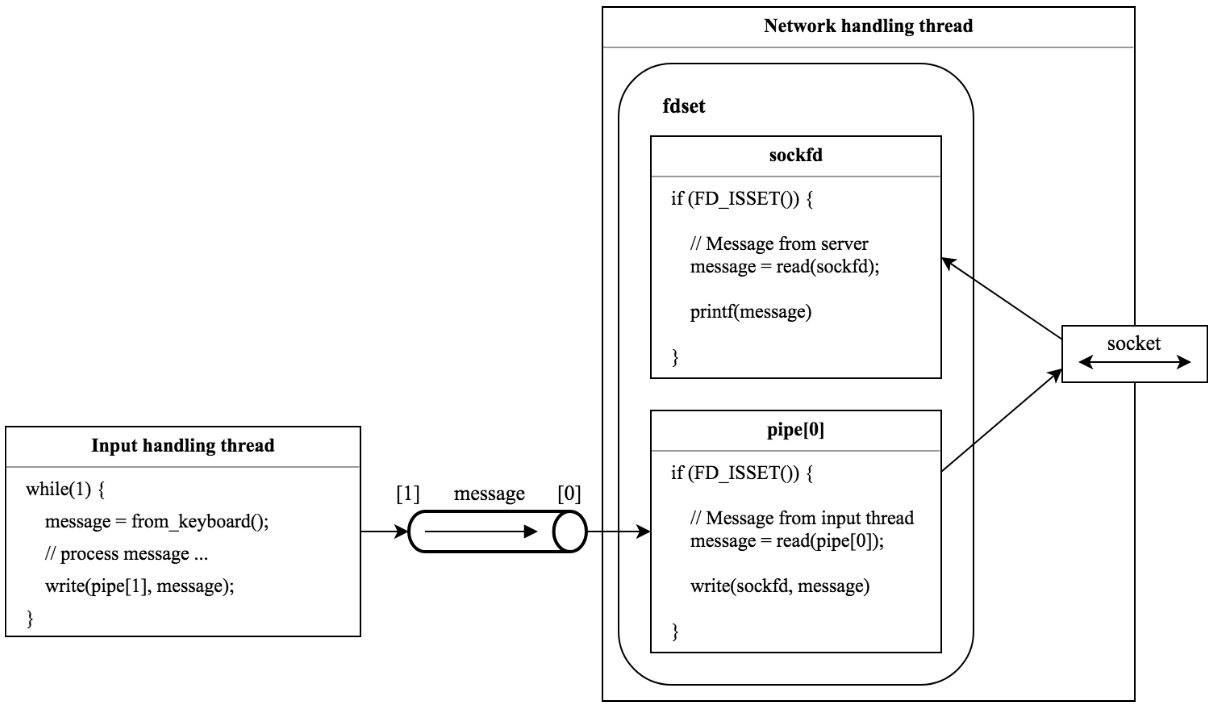
In this project, I developed a functional IRC system, with the required architecture. I tried to prevent and handle as many runtime errors could happen as I can. The system supports broadcast in a channel, send private message (PM) to a specific client. It also provides commands for client and server to utilise the system. Both server and client work with multiplexed, nonblocking TCP socket connection.

There is a short clip (1:30 mins) that demo my system: `https://youtu.be/vtqJ2ZCUiHs`

2 Client

The client firstly takes server hostname from STDIN or from arguments, and then try to connect. After successfully connected, the program creates two separated threads: input handling thread, and network handling thread. Input thread send messages from keyboard to network thread through a pipe.

The pseudo design is as follows:



```
while(1) {
    message = from_keyboard();
    // process message ...
    write(pipe[1], message);
}
```

```
if (FD_ISSET()) {

    // Message from server
    message = read(sockfd);

    printf(message)

}
```

```
if (FD_ISSET()) {

    // Message from input thread
    message = read(pipe[0]);

    write(sockfd, message)

}
```

Figure 1: Client's thread diagram

2.1 Input handling thread

The input handling thread continously getting message input from keyboard by user. The message runs through some processing such as remove '\n' at the end, . . . After processed, message is sent to the pipe to be read by the network thread.

2.2 Network handling thread

The network handling thread have a **fdset** for multiplexed connection. The set include two file descriptors:

1. **pipe[0]**: The read-end of the pipe between input thread and this thread. It is signaled when there are messages from the input thread. If the messages are successfully received, it will be send to the server.
2. **sockfd**: The socket file descriptor, it is signaled when there are messages from the server or when the connection is closed. If there are messages, it will be printed to the terminal. Otherwise, the program will stop.

3 Server

3.1 Overall design

The server will listen on a specific port for new connection. For each connection, it creates a child process to serve the client. A pair of pipe is used for each child process to communicate with the parent.

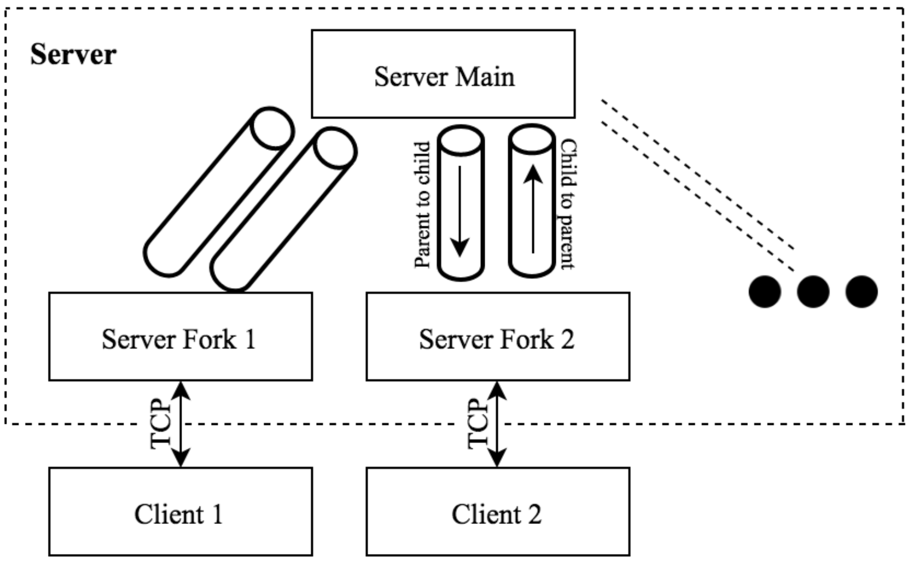


Figure 2: Server's design

3.2 Main process

When the server start, it will try to create a socket and listen for connections. The main process have an **fdset** including **STDIN**, **sockfd** (socket file descriptor), and **child_to_parent_pipe**[**0**](s)

1. **STDIN**: The **STDIN** reads commands from the administrator, and process it. Details about the commands will be discussed later.

2. **sockfd**: Signaled when there are clients want to connect, it will accept if possible. After that, pipes will be setup and the server create a child process to handle it.

3. **child_to_parent_pipe**[**0**](s): These **pipes** signaled when there are messages from the child process (fundamentally from the client). The server will process the message. If it is a command, the command will be executed and response back the results to clients through **parent_to_child_pipe**[**1**](s). If it is a normal message, server will broadcast it to all connected clients (thorugh **pipes** with child process).

3.3 Child processes

Each child process also have a have an **fdset** including **parent_to_child_pipe**[**1**](s), and its **clientfd** (client file descriptor returned by *accept()*)

1. **parent_to_child_pipe**[**1**](s): Signaled when there are messages from server to clients, just forward the message to its in-charged client.
2. **clientfd**: Signaled when trere are messages from the managing client. Simply send the messages to the server main process to handle.