Instant messenger

Why are stream sockets (i.e., TCP) well adapted to this kind of application?

UDP protocols are faster but lose some packets (for things like videos or big files that can be restores partially) but here we need TCP for reliability and no packet loss (3 way handshake), and since the packets are small, don't necessarily need fast packet transfer.

Test the provided executables. What TCP port is used by the application (use the ss command)? What options of ss did you use?

```
ss -Orlp
```

to get a detailed list of all opened ports with associated process name. If you're smart and a lazy reader:

```
ss -Orlp | grep "chat"
```

This gives us our answer

```
tcp  LISTEN  0  128  0.0.0.0:5015  0.0.0.0:*  users:
(("chat_serveur",pid=22611,fd=5))
```

So the port used is `5015`.

How many processes does the server program uses to handle 1, 2, n clients (use the ps command)?

```
    PID TTY          TIME CMD
25013 pts/1    00:00:00 bash
25043 pts/1    00:00:00 chat_serveur
25052 pts/1    00:00:00 chat_serveur
25058 pts/1    00:00:00 chat_serveur
25066 pts/1    00:00:00 chat_serveur
25080 pts/1    00:00:00 chat_serveur
25176 pts/1    00:00:00 ps
```

The program uses `n+1` processes for `n` clients.

2.1 - Client

Propose an organization of your program (with a scheme if needed) so that both interaction tasks are performed in parallel.

1. Create connexion (socket, connect)
2. Listen for messages
3. Write messages to server

To do `2.` and `3.` simultaneously, we should create a child process.

Implement a client program that can communicate with the provided server implementation. Use the source outline client.c and the indications below. Put the source code into your class report.

See file

2.2 - Server

Explain why the traditional architecture using a dedicated server process for each connection is hard to implement in the case of a chat application

It will be hard to implement as we need to keep track of every child process or socket that connects to our server to send the message to each and every one of them.

Why is the server process killed if it writes into a closed (or write-closed) socket? Propose a way to avoid this problem.

Writing onto a closed socket kills a program, that's how C is designed.

A solution is using multiplexing to check for open sockets only and track every socket.

Implement a server program that can communicate with your client. Use the provided source outline serveur.c. Put the source code into your class report.

See file