C & UNIX Project

ft_irc

42 staff staff@42.fr

Summary: This project is about implementing an IRC client and server allowing the exchange of messages on TCP/IP network

Contents

Chapter I

Foreword

Here is what Wikipedia has to say about Aligot:

Aligot or Aligote is a dish made from cheese blended into mashed potatoes (often with some garlic) that is made in L'Aubrac (Aveyron, Cantal, Lozère, Occitanie) region in southern Massif Central of France. This fondue-like dish from the Aveyron department is a common sight in Auvergne restaurants.

Traditionally made with the Tomme de Laguiole or Tomme d'Auvergne cheese, aligot(e) is a French country speciality highly appreciated in the local gastronomy with Toulouse sausages or roast pork. Other cheeses are used in place of Tomme, including Mozzarella and Cantal. The Laguiole cheese imparts a nutty flavour.

Aligot(e) is made from mashed potatoes blended with butter, cream, crushed garlic, and the melted cheese. The dish is ready when it develops a smooth, elastic texture. While recipes vary, the Larousse Gastronomique gives the recipe as 1 kg potatoes, 500 g tomme fraîche, Laguiole, or Cantal cheese, 2 garlic cloves, 30 g butter, salt, pepper.

This dish was prepared for pilgrims on the way to Santiago de Compostela who stopped for a night in that region. Originally prepared with bread, potatoes were substituted after their introduction to France. Today, it is enjoyed for village gatherings and celebrations as a main dish. Aligot(e) is still cooked by hand in Aveyron, at home as well as in street markets. Aligot(e) is traditionally served with Auvergne red wine.

Figure I.1: Aligot or Aligote

Chapter II

General Instructions

- This project will be corrected by humans only. You're allowed to organise and name your files as you see fit, but you must follow the following rules.
- The server's binary must be named `server`
- The client's binary must be named `client`
- A `Makefile` must compile both binaries and must contain the following rules: client, serveur, all, clean, fclean, re. It must recompile and re-link the programs only if necessary.
- Your project must be written in C in accordance with the Norm. Only norminette is authoritative.
- You have to handle errors carefully. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- You'll have to submit a file called `author` containing your usernames followed by a '\n' at the root of your repository.

```
$>cat -e auteur
xlogin$
$>
```

- Within the mandatory part, you are allowed to use the following functions:

  - socket(2), open(2), close(2), setsockopt(2), getsockname(2)
  - getprotobyname(3), gethostbyname(3), getaddrinfo(3)
  - bind(2), connect(2), listen(2), accept(2)
  - htons(3), htonl(3), ntohs(3), ntohl(3)
  - inet_addr(3), inet_ntoa(3)
  - send(2), recv(2)
  - exit(3), printf(3), signal(3)

- mmap(2), munmap(2), lseek(2), fstat(2)
- The authorized functions within your libft (read(2), write(2), malloc(3), free(3), etc. for exemple ;-))
- select(2), FD_CLR, FD_COPY, FD_ISSET, FD_SET, FD_ZERO
- and every functions used in bircd.tar.gz that isn't listed here.

* You are allowed to use other functions to complete the bonus part as long as their use is justified during your defence. Be smart!

* You can ask your questions on the forum, on slack...

Chapter III

Mandatory part

This project is about creating an IRC client and server (Internet relay Chat) that allows to send and receives messages between many different people, from within chat groups called “channel”.

You are however free to choose the protocol (you are not obligated to respect the RFC defining IRC, you can create your own chat protocol). You must however no matter your choice absolutely have a consistant relationship between server and client. They must communicate properly.

The communication between server and client must be in TCP/IP (v4).

You must implement the following features:

- `/nick <nickname>`: Implement the notion of nickname (the login by default, maximum 9 characters, like in the RFC).
- `/join <#chan>`, `/leave [#channel]` , etc.: Implement the notion of channel.
- `/who`: Who is connected on the channel?
- `/msg <nick> <message>`: Sends a message to a specific user on the server.
- `/connect <server> [port]`: Connects to the server from the client.
- Switch the code in non-blocking mode: the code provided is blocking in some cases. If a client does not respond, the server can block itself after a short delay. Be careful only one Select is authorized in your code.

`WARNING: This has nothing to do with non-blocking sockets which are forbidden (so no fcntl(s, O_NONBLOCK))`

Chapter IV

Server

Usage:

```
$> ./server <port>
```

Where “port” means the server’s port number.

The server must support multiple simultaneous clients through a select(2) in read and write. You can only use one select(2) for the server that must be non-blocking.

An example of server, bircd.tar.gz, is provided with its sources to get you started. You can use and submit these sources, but careful: this base won’t bring you points, and only uses select(2) for read, you need to fix that.

Your code will not only be non-blocking but will also have to use some round circular buffers to secure and optimize the sending as well as receiving of the different commands and responses. It’s up to you to produce a clean code, checking absolutely all the mistakes and all the cases that can bring errors (sending or receiving partial commands, low bandwidth, etc.). To check that your server uses correctly everything mentioned, a first test is to use `nc` with Control+D to send command parts:

```
$> nc 127.0.0.1 6667
com^Dman^Dd
$>
```

This will have the effect of first sending the letters “com” then “man” then “d\n”. You will therefore have to aggregate the packets in order to recreate the command “command” to be able to process it.

Chapter V

Client

Usage:

```
$> ./client <server> <port>
```

Where "server" is the name of the hosting machine on which your server is, and "port" the port number.

Considering that your client implements the command `/connect <server> [port]` the following synopsis is equally valid:

```
$> ./client [server [port]]
```

In general, make sure to only use client's commands close to those of a classic irc client

Be careful not to confuse the client's commands and the IRC protocol: http://en.wikipedia.org/wiki/List_of_Internet_Relay_Chat_commands

Chapter VI

Bonus part

⚠ We will look at your bonuses if and only if your mandatory part is EXCELLENT. This means that your must complete the mandatory part, beginning to end, and your error management must be flawless, even in cases of twisted or bad usage. If that's not the case, your bonuses will be totally IGNORED.

Here are couple of bonus ideas:

- Prompt management in the client
- File transfer (via the server)
- File transfer (without going through the server)
- RFC 1459 or better 2813 compliant
- Support of IPv6