⌠ Coursework 2: Socket Programming ⌡

2020-11-23

Introduction Most operating systems (if not all) provide network capabilities to user applications through interfaces called sockets. From a developer point of view, sockets are an API to the transport layer functionalities. While most libraries dedicated to application layer protocols handle the socket creation and management for you, knowing how they work can help when troubleshooting network communication problems. But sockets are not the only thing required in order to build a software capable of network communication, it also requires an application layer protocol and pieces of code to comply with that protocol. This has to be handled slightly differently depending if the software acts as a client (*i.e.* it initiates communications) or as a server (*i.e.* it waits for clients communications).

Goal In this project, you will have to understand the basics of a simple protocol used for achieving real-time group talk (Internet Relay Chat) and implement a basic client and a simplified server for this protocol. The basic client will be a robot which will reply to simple commands sent to group talk (called channels in IRC). When receiving private messages, the robot will also reply with random non-sense or fun-facts to the user who wrote. The server you will implement should be a very light version of a real server which will cover only the most basic functionalities of the IRC protocol: clients should be able to connect, join channels and interact with each other using channels or private messages.

⟨Part A⟩ IRC and the project

⌊Section A.1⌉ *The protocol*

IRC is a text based protocol running on top of a TCP stream (eventually secured by TLS but this is not required in the project). The protocol is human readable and so simple that a human can even use it manually, at least when there is not much talk on channels. It was originally fully described in the IETF RFC 1459[1]. That RFC was later updated and superseded by the following set:

- IETF RFC 2810[2] for the general architecture
- IETF RFC 2812[3] for the client/server communication

[1] https://tools.ietf.org/html/rfc1459
[2] https://tools.ietf.org/html/rfc2810
[3] https://tools.ietf.org/html/rfc2812

- IETF RFC 2811[4] and IETF RFC 2813[5] for advanced channel management and server/server communication that you don't have to consider in this coursework

However, the best way to actually get a good first understanding of it is probably to just run locally a simple server such as miniircd[6] and connect a real graphical client to it such as Hexchat[7]. This client allows you to see all the raw traffic from/to the socket (*i.e.* the TCP stream) by opening the 'Raw Log' window in the 'Window' menu. You can then type '/connect 127.0.0.1' and see what is happening while playing with the functionalities of the client. Try to join a channel, talk to it, have a second client connect and do the same,... Note that similar results could also be achieved by watching localhost traffic in Wireshark and following the TCP Flow of the connection between the client and server.

Section A.2 *A simple client (bot)*

The focus of the client you will implement is on network communication and not graphical interfaces, thus you are asked to make a robot able of doing the following task:

- Connecting to a server
- Maintaining the connection to the server
- Joining a channel
- Responding to channel messages starting with a '!' character and more specifically:
  - '!hello' greeting the user sending it
  - '!slap' slapping a random user in the channel with a trout
- Responding to each private message by a random sentence (either nonsense or fun-fact)

As a requirement, the robot program must be running inside the Windows virtual machine provided for the module. It must connect to the server running on the Ubuntu virtual machine provided for the module using IPv6. Addresses used for both virtual machine must be the ones from Week 2 preparation (in the fc00:1337::/96). Miniircd supports IPv6 when started with.

Section A.3 *A simple server*

There is no need to implement a full IRC server to understand how sockets works on the server side, thus it should only provide the functionalities of the protocol which are strictly required to achieve the following task:

- Allowing clients to connect, choosing there username and realname
- Allowing clients to join channels
- Allowing clients to talk to others users in a channel

[4] https://tools.ietf.org/html/rfc2811
[5] https://tools.ietf.org/html/rfc2813
[6] https://github.com/jrosdahl/miniircd
[7] https://hexchat.github.io/

- Allowing clients to talk directly to each other in private

As a requirement, the server must be running on the Ubuntu virtual machine provided for the module. It must listen to IPv6 connection from any address.

⟨Part B⟩ Sockets

Since sockets are provided by operating systems, there specific implementation in low level languages (such as C/C++) are usually dependant on the operating systems although some cross-platform libraries also exist. However, higher level of languages such as Python provide natively in there standard library an interface to sockets which is independent of the operating system. A general introduction to sockets and their relation to the Transport layer can be found in this video[8].

⌊Section B.1⌉ *With Python*

The reference documentation of Python socket API[9] is freely available. A few other good tutorials, sometime better than the reference documentation, can also be found:

- https://medium.com/python-pandemonium/python-socket-communication-e10b39225a4c
- https://realpython.com/python-sockets/

For those prefering videos, http://youtu.be/_FVvlJDQTxk provides a quick overview of sockets in Python and http://youtu.be/wzrGwor2veQ covers them more extensively particularly in Python 3.

⌊Section B.2⌉ *With C/C++*

Things are more tricky with C/C++, depending if you want a cross-platform solution or not. For the Windows operating system, the standard library to access sockets is called WinSock and its reference documentation is available at https://docs.microsoft.com/en-us/windows/win32/winsock/getting-started-with-winsock. Two alternative good tutorial about it can be found at:

- https://www.binarytides.com/winsock-socket-programming-tutorial/
- https://www.codeproject.com/Articles/13071/Programming-Windows-TCP-Sockets-in-C-for

On GNU/Linux, as on many unix like operating system, the socket API is available as a POSIX specification. The following two tutorial gives the basics for using it:

- https://www.binarytides.com/socket-programming-c-linux-tutorial/
- http://tldp.org/LDP/LG/issue74/tougher.html

[8] https://www.youtube.com/watch?v=0s_w8eHu6LQ
[9] https://docs.python.org/3.9/howto/sockets.html

The reference documentation is available as a classical man page, a copy of which is at http://man7.org/linux/man-pages/man2/socket.2.html. It is not recommanded to look at it until you are well familiar with all the concepts involved. Another good reference which is more understandable for beginners is available at https://www.gnu.org/software/libc/manual/pdf/libc.pdf pages 479 and next.

A cross-platform solution exists for C++ through the excellent Boost library where sockets are provided in the Boost.Asio module. Its reference documentation is available at https://www.boost.org/doc/libs/1_71_0/doc/html/boost_asio.html and a very well written tutorial is available at https://www.codeproject.com/Articles/1264257/Socket-Programming-in-Cplusplus-using-boost

⟨Part C⟩ Documentation and Software Development

⌊Question C.1⌉ *Coding good practice*

This assignment is part of your entire course on computing, and thus good practice on coding and software that you have learned in previous modules should be used in this assignment. Your code should be easily readable and you are expected to use well designed data structures that are correctly documented in the code.

You can refer to previous module materials and content for this, otherwise a selection of good advices can be found at:

- https://opensource.com/article/17/5/30-best-practices-software-development-and-test
- https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readabl

Note that you are not expected to follow exactly all advices given in those resources and some are irrelevant to this coursework. They are just references from which a solid basis for this coursework can be built on.

While not mandatory, the use of a version control system such as git is encouraged as well and this will help you fill objectively the student's contributions in the marking sheet by tracing who has contributed to which part of the code.

⌊Question C.2⌉ *One page comment*

In addition to the short comments you can make on the marking sheet on each item, you are offered an opportunity to further comment on 4 items that are incomplete to recover some marks. These should fit within a single page and for each chosen item the problem must be briefly explained (what is happening compared to the expected result) with appropriate references to the protocol and a possible solution (why this is happening and how it could be prevented). Note that at least half of the issues must be related to network and the protocol, *i.e.* they should not be purely coding related. The other half can be coding related (bugs, ...) but should be put in perspective of networks and the protocol.

In the case that you would find yourself short of issues to comment on, you can instead explain how you would extend your implementation to achieve additional features which are not required in this assignment. Again appropriate references to the protocol are expected and the comments should focus more on the network viewpoint than coding.

⟨Part D⟩ Submission and Marking

This coursework uses a combination of self-assessment and peer-review for the marking. You are providing with 3 documents to assist you with this:

- The marking sheet that you must fill together (if you would not agree on it, this can be submitted individually, please drop me an email if so).
- A partial rubric grid, this is a document which specify for each item/functionality of the marking sheet how the marks are distributed.
- A partial test procedure detailing the main steps of how your implementation will be tested with references to the rubric for marks removed in case tests fail.

Note that both of the last two documents are incomplete on purpose as detailing some of the test or criteria too much would give strong hints as to the correct solution to some of the problems you might encounter while implementing the protocol.

Submission will be done through the assessment area on MyDundee and will follow a two step process:

1. On **Thursday 17 December at 17:00**, you must have submitted your code,
2. On **Friday 18 December at 17:00**, you must have submitted the marking sheet and the one page comment on existing issues or additional features.

Academic Misconduct

While discussing this assignment between groups is acceptable, simple copy and paste or paraphrasing the solution from one to another is an academic misconduct as per the Code of Practice on Academic Misconduct by Students available at `https://www.dundee.ac.uk/qf/documents/details/academic-misconduct.php`. Proper references to resources used for preparing your submission are also mandatory, especially if copying or para-phrasing those materials. **That includes any pieces of code gathered online used in your own code.**