IPC Lab 01: Creating a Simple Chat Application

Vassilis Markos

Week 11

Abstract

This lab series is intended to help us revise on any IPC relevant concepts within the context of a realistic application: designing a simple single chat room application utilising UDP sockets, shared memory segments and semaphores.

1 Setting Up A Simple Client–Server

As a fist step for our (and, virtually, any) networked application is to set up a client and a server. In our case, the setup will be quite simple:

- The server, defined in `server.c`, will handle new connections and distribute content appropriately. The server will also be responsible for keeping logs of any communications within the application.
- The client, defined in `client.c`, will create new connections to the server, send messages, and receive any messages sent by other clients to the shared chat room.

Exercise 1. *Create a client and a server process that communicate through UDP sockets and can exchange hard–coded string messages. That is, the client should be able to:*

- *Send a string message to the server, which should be hard–coded into the implementation code of the client.*
- *Receive a response from the server, which it should print on screen.*

In a similar fashion, the server should be capable of:

- *receiving client messages, printing them to screen;*
- *echoing any received client messages ack to the client as a response.*

Test your implementations by first starting your server and then your client in two separate terminals.

Exercise 2. *Implement a simple logging utility for your server. That is, whenever a message is received from a client and whenever a response is sent back you should write into a log file the following information:*

- *The message timestamp;*
- *The message sender;*
- *The message receiver;*
- *The message contents.*

You can look on relevant online resources / books about how to get access to the system's clock, to get a message's timestamp.

Exercise 3. *At last, implement a simple Command Line Interface (CLI) for your client (and, maybe, server, if you want to). Your CLI should allow users to write a message on the CLI and this message should be then sent to the server, which will echo it back to the client. Upon receipt, the message should be displayed on screen. For your implementation, you can either utilise Bash–emulating functions, or just use common C IO functions.*

2 Handling Multiple Connections

So far, we have implemented a simple server that can handle a single client. In practice, however, more than one clients might want to connect to the same server at the same time. For this to work smoothly, we might utilise the following IPC–related constructs:

- The server process will create a separate thread for each client connected to it, to ensure autonomous handling of different clients.
- All threads will use a shared buffer to "post" messages onto; this is where the server reads from.
- All threads will use another shared buffer to "read" messages from; this is where the server posts to.
- All processes should be properly synchronised using semaphores.

Exercise 4. *Modify your server logic so as to accommodate for multiple simultaneous client connections. To do so, you will need to create and join a new thread for each newly received client connection. Do not take care of synchronisation at this stage, just make sure that your server can actually handle multiple connections — verify that by creating a few clients (at least two) and send a few messages through each client's CLI.*

Exercise 5 (Skip this if you have already implemented one in previous labs)**.** *Implement a Cyclic Buffer as a C structure. Your cyclic buffer should:*

- *have fixed maximum size;*
- *utilise a C array (statically or dynamically allocated) to store its contents;*
- *utilise two pointers, one pointing to the head and one to the tail of the buffer;*
- *implement at least a* `push()` *and a* `pop()` *operation, and two logical checks,* `is_empty()` *for underflows and* `is_full()` *for overflows.*

Use a separate source file for your implementation and a separate header file to facilitate inclusion in other files.

Exercise 6. *Finalise multiple connection handling by utilising semaphores to prevent different client threads from accessing their critical sections simultaneously. That is, each thread as implemented server–side should access its own critical section uninterrupted by others. Typically, each thread's critical section in this case should comprise of just the part accessing the shared buffers for reading from and writing contents to them, respectively.*