Practical assignment on Concurrent Programming 2024/2025

Nelma Moreira José Proença

Department of Computer Sciences, Faculty of Sciences, University of Porto

Groups

This is a group assignment. Groups should have at most 2 members – if you fail to find a partner please let us know. You can join a group using Moodle: `https://moodle2425.up.pt/mod/choicegroup/view.php?id=186186`

What to submit

A single ZIP file with the *PDF report* that answers the proposed questions **and** the *source code* of the developed *implementations* in CAAL/PseuCo and in Scala. Use different folders with the implementations of different questions, include a `readme.txt` explaining how to compile and run each implementation, and **do not include compiled code** (e.g., the `target` folders for Scala).

Presentation

Present your encodings and implementations in a 8 minute presentation, to be scheduled for **June 4** - **June 5**, 2025.

Deadline

23h59m of **May 30**, 2025 (Friday)

The Alternating Bit Protocol (ABP)

Exercise 1. [5 pt] *Alternating-Bit Protocol* (ABP) is a simple protocol for reliably transmitting data from a sender (`Sender`) to a receiver (`Receiver`) over unreliable communication channels. This is accomplished by retransmitting lost messages if required. `Sender` transfers data to a component `Trans`, and `Receiver` confirms the reception to a component `Ack`. `Trans` and `Ack` simulate the network behaviour; these can transfer a single message at a time and can either lose or duplicate messages. The overall behaviour is summarised in Fig. 1 and explained in detail below.

- `Sender`: In order to send a message the `Sender`:

- Accepts a message to send.
- Sends it to `Trans` with the control bit b (initialy set to 0).
- After send ing it behaves non-deterministically. It can either

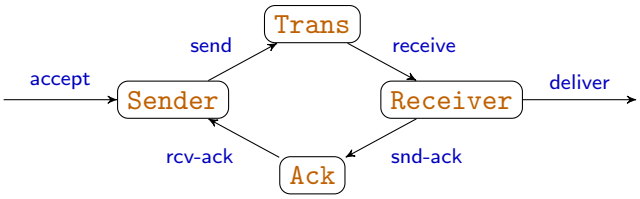

Figure 1: Overall behaviour of the communication between a Sender and a Receiver

  - resend the message with the same control bit b; or
  - receive an acknowledgment b from Ack and restart its behaviour using the control bit $1 - b$.
- Unexpected confirmations from Ack are ignored.

- Receiver: Receiving a message is similar. The Receiver:

- Receives and delivers the message.
- Sends an acknowledgment via Ack with bit b.
- After delivering it behaves non-deterministically. It can either
  - Resend b via Ack; or
  - Receive a message with $1 - b$ from Trans, deliver it, and continues with $1 - b$ instead of b.
- Ignores any message from Trans with bit b.

- Trans and Ack: These abstract away the behaviour of the unreliable network. More specifically:

- Communicate the messages and the control bit send by the Sender to the Receiver and the acknowledgments from the Receiver to the Sender.
- They can lose or duplicate messages.

There is no direct communication between Sender and Receiver. All messages must travel through the network. The only actions visible to the environment are accept and deliver on the Sender and Receiver side; everything else is considered to be internal communication. The goal of this work is to implement two variants in CCS of the ABP protocol.

1.1. Implement ABP in CCS as the composition of the four components above. Abstract away from the content of the messages and focus on the additional control bit.

1.2. Implement an alternative specification SPEC of the protocol in CCS as a single process without parallel composition. Check whether or not it is equivalent to your previous implementation of ABP by using a suitable form of bisimulation. Use CAAL/PseuCo and submit your code (properly commented) in your submission zip file.

1.3. Reimplement the ABP in *value passing CCS*, including the CAAL/PseuCo code (properly commented) in your submission zip file, such that:

1. the value sent is the number of the message being sent, starting with 0; and
2. the control bit b must also be sent.

Run processes in a server

Exercise 2. [7 pt] It is not possible to execute an OS process in a browser via JavaScript. A possible workaround is to implement a server that receives requests from a browser, with the instructions to be executed, and having the browser delegating these instructions to the server.

We provide you the JavaScript client that runs in a browser and a working skeleton of a server in Scala. In this exercise you will implement the rest of this a server. The provided files can be found in `https://fm-dcc.github.io/cp2425/exercises/process-runner.zip`, which include the files:

- `client/index.html` – where you can open a website running JavaScript that can request bash commands to a given server;
- `server/src` – where you can find the source-code to a basic server with placeholders that you need to fill;
- `server/build.sbt` – where you have a configuration that you can use to run the server.

Running and using the server. In a Linux system, open a terminal in the `server` folder and run it using the command `sbt run`. Then using an internet browser (e.g., Chrome) open the website "`client/index.html`".

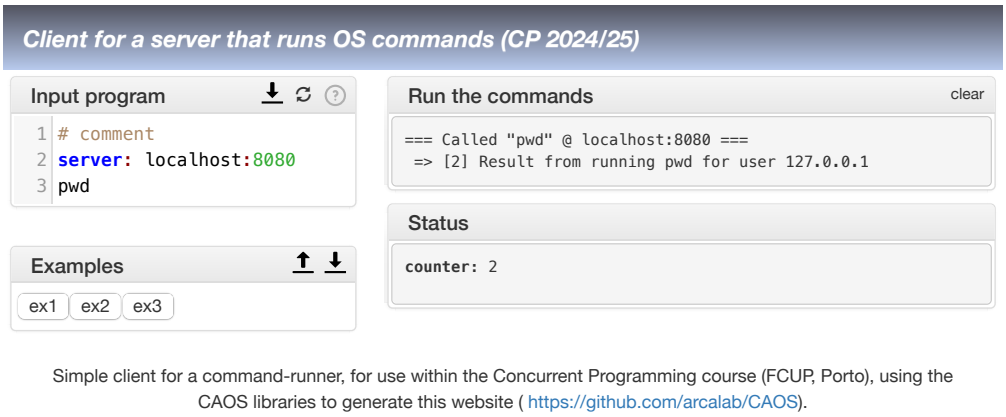

Figure 2: Screenshot of the client's view in a web browser

A screenshot of the website from `client/index.html` is in Fig. 2. You can provide a list of commands in the left window, called "Input program". Click the header "Run the commands" to expand that window and to send requests to the server mentioned in the input program, one request for each command below a `server` reference. The reply from the server will be appended to that window.

After expanding this window once, every time you press the refresh button next to the "Input program" header you will resend the list of requests, and append the replies to the window. You can press the "clear" button to delete the content of this window.

If you click on the "Status" header, you will trigger this window to expand and will send a request for the state of one of the servers, and the reply will eventually be displayed in this window.

What to implement. This assignment is very flexible. You will have to support the execution of processes in a single server assuming many requests can be made at the same time, by different clients. The requirements of this server are listed below.

1. More than one request can be received and executed without waiting for the previous one to terminate.
2. There should be a maximum number of requests that can be executed simultaneously – after that processes should be queued.
3. Requests should be processed by order of arrival.
4. Shared data structures should be used to model the state of the server.
5. Two different kinds of requests can be made to the server: **run a given terminal instruction**, and **ask for the state** of the server.

You should include in your zip file all the source code, using different folders for different questions if you want to use variations of the server. The main files that you need to modify are:

- `server/src/main/scala/cp/serverPr/Routes.scala` (to specify how to react to requests) and
- `server/src/main/scala/cp/serverPr/ServerState.scala` (to specify the internal state of the server).

If you want to run more than one server listening over different ports, you can change the port number in the file `server/src/main/scala/cp/serverPr/Server.scala`.

2.1. Implement the following mechanisms in some part of your server, **using the approaches described in lessons**. If you prefer, you can have multiple versions of the server, each to illustrate different mechanisms, placed in different folders in your submission zip file. You can also add new requirements or functionalities – if so explain these. For each of the mechanisms below you should explain: (1) **how to run and test** your mechanism, (2) **why it makes sense**, and (3) **how it is implemented**, including which shared data structures are used.

1. Synchronised blocks;
2. Lock-free programming;
3. Volatile variables;

2.2. Implement a *bad* variation of one of your versions with a **data race**, and:

1. explain the data race,
2. explain how to replicate the data race, and
3. show how the problem is displayed and how it can be interpreted.

Simulating ABP with actors

Exercise 3. [5 pt] Recall the ABP from Exercise 1. In this exercise we will ask you to simulate the ABP problem using actors.

3.1. Describe the structure and the behaviour of your actors, and include a diagram with the actor hierarchy.

3.2. Describe how to compile, execute, and test your code, illustrating a successful scenario where no message is neither lost nor duplicated.

3.3. Describe how to compile, execute, and test your code, illustrating two scenarios: (1) a successful one where messages are lost and duplicated, and (2) a failure one where two many messages are lost or duplicated.

Presentation

Exercise 4. [3 pt] Give a short 8-minute presentation summarising your CCS encodings and your Scala implementations. You do not need to submit slides and all members of the group should talk when giving the presentation. The presentation will be scheduled for June 2 - June 3, 2025.