# Concurrent Programming

## Exercise Booklet 8: Message Passing

1. Implement the turnstile example given at the beginning of the course. You should have a counter thread and two turnstile threads. All reads and writes to the counter are managed through its dedicated thread.

2. Implement a server that concatenates strings. It should follow the protocol given by the following atoms (which are sent to the server):

   - `{start}`: client wishes to send a number of strings to be concatenated;
   - `{add,S}`: concatenate string `S` to the current result;
   - `{done}`: done sending strings, send back result.

3. We wish to define a server that receives two types of messages from a client: "continue" and "counter".

   (a) Each time the server receives the message "counter" it should display on the screen the number of times that it received the "continue" message since the last time it received the "counter" message.

   (b) Modify your solution so that the server, instead of printing the number on the screen, sends it back to the client which then prints the number of the screen himself.

4. Write a server that receives a number `n` and returns a boolean indicating whether it is prime or not. Provide two solutions, one where you define an ad-hoc client and server and another using the generic server seen in class.

5. Assume we have one server and one client. The server should behave as follows: generate a pseudorandom number in the range [0,10]. The client should try to guess the number the server has generated. In order to do so, it sends it successive messages containing a number. Each message is answered by the server, indicating whether the client has guessed or not. Once the client has guessed, the server is ready to accept a new client.

6. Modify the solution to the previous exercise so that the server delegates the task of checking (and answering) whether a client's guess is correct or not to a new thread.

7. We wish to model a *Timer* that, when spawned, received the frequency with which it should generate *ticks* and the set of pids which should receive ticks (a message represented as the atom `tick`). For example, `spawn(timer,timer,[100, [pid1,pid2,pid3,pid4]])` will initiate the `timer` process that will send tick messages every 100 milliseconds and send them to pids `pid1`, `pid2`, `pid3` and `pid4`.

8. Modify your solution so that the *Timer* is initially created with an empty list of pids and that clients have to register in order to receive ticks. For that they need to communicate with the *Timer* sending it a `register` message.

9. We wish to model a network of temperature sensors. Each sensor maintains a value (representing a temperature). This value may be modified in two ways: it can receive a direct reading (that updates the value of that sensor) or it can update its value based on the values of its neighboring sensors following the protocol which we describe next. Each sensor has a list of its neighboring sensors (you may assume this list is fixed when the sensor is created). All sensors are synchronized by means of the *Timer*: every time the timer ticks, each sensor sends its value to all its neighbors each of which update their own value by taking the average of all received values.