COM3026 – Lab 3

Goals

You will practice distributed programming in Elixir and writing event-driven components.

Before you start

- ☞ You can work individually or in groups of two.
- ☞ It is recommended that you store your programs in a folder hierarchy comprising of a single high-level folder, e.g., `COM3026`, and one sub-folder for each lab session, e.g., `lab1`, `lab2`, etc.
- ☞ Read carefully the instructions, and ask for help if you feel lost.

Exercise 1. Examine the code of the `ExcludeOnTimeout` module, which implements the ”Exclude on Timeout” algorithm from the textbook (Algorithm 2.5) as an event-driven component. The code is available in the `exclude_on_timeout.ex` file, and can be downloaded from the class website. Run the code in multiple processes as per the instructions below. Try to kill some processes, and observe the crashes being detected by the remaining processes.

Follow the steps below to run the code. For the sake of an example, we will create *three* concurrent instances of the ”Exclude on Timeout” algorithm, identified through the atoms `:p1`, `:p2`, and `:p3`. The algorithm can be instantiated in a similar manner for any number of concurrent processes.

1. Compile the code by running `c "exclude_on_timeout.ex"` from the Elixir shell.

2. Run the following code from the IEX prompt:
   ```
   procs = [:p1, :p2, :p3]
   pids = Enum.map(procs, fn p -> ExcludeOnTimeout.start(p, procs) end)
   ```
   This will create three Elixer processes each running an instance of the `ExcludeOnTimeout` module. The PIDs of these processes will be stored in the list `pids` returned by `Enum.map`.

3. Use `Process.exit` to kill one of the processes started in 2. For example, to kill `:p2`, run the following from the IEX prompt: `Process.exit(Enum.at[1], :kill)`. Observe that the remaining processes are able to detect the crash.

4. To visualise the concurrency, uncomment the `IO.puts` statements at the beginning of each event handler, and rerun the code.

5. You can also try to see if you can force the failure detector to make a mistake (i.e., to falsely detect a healthy process as crashed) by reducing the timeout duration (`state.delta`).

Exercise 2. Make the implementation from Exercise 1 more efficient by adding a logic to prevent the processes from sending heartbeat requests to themselves. Make sure that no process can ever suspects itself as crashed as a result of this optimisation.

Exercise 3. Implement the ”Increasing Timeout” algorithm for Eventually Perfect Failure Detector ($\diamond P$) that was studied in class (see also Algorithm 2.7 in the textbook).

1. Start by creating a new module `IncreasingTimeout` and store it in a file named `increasing_timeout.ex`. Then, copy the code from `exclude_on_timeout.ex` into it, and revise it appropriately to support the new logic and the interface.

2. Test that your implementation of Increasing Timeout is capable of tolerating intermittent periods of asynchrony, as implied by the $\diamond P$ specification.

   To simulate asynchrony, you can instrument your code to either delay replies to the heartbeats, or just omit sending replies altogether. Note that the asynchronous periods must be of limited duration as otherwise, the algorithm is not guaranteed to ever stop making mistakes (i.e., output false detections).

   Use `Process.sleep` to suspend a process for the specified time interval (in milliseconds). For example, the following code will suspend process `:p1` for 10 seconds:
   ```
   if state.name == :p1, do: Process.sleep(10000).
   ```