Distributed Algorithms 347

Coursework 1

01 The aim of this coursework is to explore much of the work done in the previous labs and in the lectures.

02 We highly recommend that you finish lab 2 before starting on this coursework.

03 You can work and submit either individually or jointly with one classmate. You will probably be spending more time evaluating your results than coding. So start early and give yourself enough time for testing and evaluating.

04 Don't assume that the Elixir examples in the lab or given in the lectures are suitable for this coursework in the form given, you will need to adapt them to get them to work.

05 Write comments to explain what less obvious code is doing - but don't over-comment. Don't use long lines. Don't use super long names for atoms, variables, functions etc. - unless super long names help **you** to understand what's going on. *Write comments to help you to revise the material in the future*. **If your code does not work or only partially works you must explain what's not working**. If you don't document what not's working, the assumption is that you have tested it.

06 All your source files must include a comment at the top of the files with your name(s) and login(s) e.g.,

```
# Mary Jones (mj16) and Peter Smith (ps16)
```

07 Source files without your name(s) and login(s) **will not be marked**.

08 Also submit a **pdf** report with your outputs and summaries on the tests you conducted. State at the beginning what your local and Docker setups were - e.g. computer/VM, processor, ram, cores. Again ensure that your name(s) are on your report. **We are interested in comments that demonstrate your understanding and critical thinking, including your rationale when designing new tests.**

09 Submit the directory of your report, code, build files, (e.g. Makefile, docker-compose etc) as a single *zip* file on Cate. Include a Readme file with instructions on how to run your tests. It should be possible for us to run your systems on using your instructions.

10 You can organise your code either in a single mix directory (e.g. parameterise the Makefile with a VERSION number) or in several. Simplest is to adapt the Flooding Makefile from Lab2.

11 Your submitted files must also print correctly on DoC printers so check first - you will not be given an opportunity to resubmit if a file does not print correctly.

12 The deadline for submission is **Wednesday 7th February 2018.**

13 Use Piazza if you have questions about the specification or general questions. **Do not post your solutions or share them with others.** Email me directly if you have a question about your solution.

14 If there are any corrections or clarifications, the most up-to-date version of the coursework will be provided on the course webpage not on CATE.

15 There are 20 marks in total for the coursework. There are also upto 5 marks for doing the bonus parts. You will be given credit for doing bonus parts if your mark is below 20. For example if you score 14 for the non-bonus parts and 3 for the bonus parts, your total mark will be 17. If you score 19 for the non-bonus parts and 3 for the bonus parts your total will be 20 (not 22), i.e. the total mark is capped.

System1 – Elixir Broadcast

16 Create a system with N=5 fully connected peers. The structure is

```
System1
  Peer0, .., PeerN-1
```

17 Each peer should **broadcast** `max_messages` or until a `timeout` (in milliseconds) is reached.

18 After creating and binding the peers, `System1` should sent a message `{:broadcast, max_messages, timeout}` to each peer to instruct them to start the broadcast.

19 For the message `{:broadcast, 1000, 3000}` the output should be similar to :

```
 1: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
 4: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
 3: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
 0: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
 2: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
```

20 This formatting isn't strict, you can output something similar, but please use 1 line per peer

21 In the example each peer outputs one line after the timeout is reached. The first number on an output line is the peer number P (from 0 to N-1), the rest are pairs of values, one for each peer Q (0 to N-1) where the first number of the pair is the number of messages sent from peer P to peer Q, and the second number is the number of messages received by peer P from peer Q. The order of the lines does not matter, since the system is non-deterministic. The order of the pairs is Q=0..N-1

22 Your peers shouldn't do all the broadcasts first and then all the receives. Rather interleave sending of a broadcast message with receiving messages from other peers. This doesn't need to be at a high level of granularity.

23 Note that the value 0 and the atom `:infinity` have special meaning in the `after` clause of a `receive`.

24 **Submission**: show and **comment** on the output of the system when run (a) locally and (b) on a Docker network with 1 container for each peer and one for the system for the following requests:
(i) `{:broadcast, 1000, 3000}`
(ii) `{:broadcast, 10_000_000, 3000}`
(iii) your own interesting requests - please only show the output of one request and summarise the results of the others you tested with.

25 You will get 'odd' results doing this coursework - your aim should be to come up with a possible explanation of why they happen and if 'bad' how they might be improved.

26 (**Bonus**) Distribute, run and comment on the results of running `System1` on a non-virtualised network of lab computers. Similarly, for `System2` to `System6` below. You can do this by manually *ssh*-ing into each computer or running *ssh* commands via a shell script, (or using Ansible, Fabric etc if available)

System2 – PL Broadcast

27 The aim of `System2` is to adapt `System1` to use Perfect_p2p_links components (PL) giving the following hierarchy:

```
System2
  Peer0
    PL, App
  ...
  PeerN-1
    PL, App
```

28 One way to connect PL components to each other is for each `Peer` to send the Elixir process-id of its PL component to `System2` and for `System2` to send a suitable `:bind` message to each PL component. `Peer` can then return once its created and bound its components (`PL` and `App`).

29 Once created and bound to its `PL` component `App` takes on the role of `Peer` in the system. However, `App` communicates with other peers using its PL component.

30 PL components can communicate with each other using any available communication mechanism, however for this coursework it is recommended to use Elixir's message passing!

31 **Submission**. Repeat as for System1. You can vary your own requests from previous submissions.

System3 – Best Effort Broadcast

32 The aim of `System3` is to adapt `System2` to Best_effort_broadcast (BEB) components. The hierarchy of `System3` is now:

```
System3
  Peer0
    PL, BEB, App
  ...
  PeerN-1
    PL, BEB, App
```

33 **Submission**. Repeat as for System1. You can vary your own requests from previous submissions.

System4 – Unreliable Message Sending

34 The aim of `System4` is to adapt System4 to add a reliability parameter to the PL components to simulate unreliable message passing.

35 Write and use `Lossy_P2P_links` (LPL) instead of PL with an integer reliability percentage from 0 to 100. 100 sends all messages (100% reliable). 0 drops all messages (0% reliable). 20 sends ~20% of the messages - use a suitable random number test to decide whether to send a message or not.

36 **Submission**: using reliability values of 100, 50 and 0, repeat as for System1. You can vary your own requests from previous submissions.

System5 – Faulty Process

37 The aim of `System5` is to adapt `System4` and make `Peer3` terminate itself after 5 milliseconds (choose a different termination timeout if it helps).

38 The Elixir function `Process.exit()` can be used to terminate an Elixir process.

39 **Submission.** Repeat as for System1. You can vary your own requests from previous submissions.

System6 – Eager Reliable Broadcast

40 The aim of `System6` is to adapt `System5` to use Eager reliable broadcast (BEB) components. The hierarchy of `system6` should be:

```
System6
  Peer0
    PL, BEB, RB, App
  ...
  PeerN-1
    PL, BEB, RB, App
```

41 **Submission.** Repeat as for System1. You can vary your requests from previous submissions. Test with with and without a faulty Peer.

42 (**Bonus**): Implement and evaluate lazy reliable broadcast.

43 (**Bonus**): Add support for routing over a multi-hop network and test for lab02 example net (hard).

44 -- Good luck ! --