Distributed Algorithms 60009

Lab 3 – Broadcast Algorithms

https://www.doc.ic.ac.uk/~nd/dac/lab3.pdf

01 The aim of this lab is to gain a better understanding of broadcast algorithms and of various issues that arise when developing and analysing them.

02 This week's lab is more substantial than the previous ones. Use Piazza if you have questions.

03 Read the whole of this worksheet before starting. If you've not completed lab 2, you should consider completing lab2 before starting on this lab.

04 Elixir solutions WILL NOT BE provided for the exercises, so you'll need to do the exercises.

05 You can work on the exercises with a classmate.

06 You will get *strange* results doing the exercises. Your aim is to come up with explanations of why they happen and if *bad* how you would adapt your implementation. Move on to the next exercise as soon as you can.

07 There is often more than one way of completing an exercise. You need to make you own choices. The important thing is to understand the effects of your choices. Focus your efforts on completing the exercises, not on learning "advanced" Elixir features or elegant Elixir idioms.

08 A **`Makefile`**, a **`Helper`** module, and a skeleton **`Broadcast`** module are available in http://www.doc.ic.ac.uk/~nd/dac_src/lab3_files.tgz

BROADCAST 1 – Elixir Broadcast

09 Create a broadcast test system with N=5[1] fully connected peers:

```
1  Broadcast
2    Peer0, .., PeerN-1
```

10 After creating and binding **`Peer`** modules, **`Broadcast`** should send a message `{:broadcast, max_broadcasts, timeout}` to each peer to instruct it to start.

11 Each peer should *try* to broadcast **`max_broadcasts`** messages. However, when `timeout` (in milliseconds) is reached each peer should printout its status (its counts - see next step) and stop.

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

```
                Peer0       Peer1       Peer2       Peer3       Peer4
3   Peer1: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
4   Peer4: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
5   Peer3: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
6   Peer0: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
7   Peer2: {1000,1000} {1000,1000} {1000,1000} {1000,1000} {1000,1000}
```

13 This formatting isn't strict, you can output something else, but try to use 1 line per peer.

14 The counts (in curly brackets) are the number of messages broadcast to and received by each **`Peer`** module (the **`Client`** module in later systems) with other Peers (including itself!), not the

[1] The Makefile creates 10 peer nodes. Use fewer peers for debugging/testing.

number of messages exchanged by modules like `PL`, `LPL`, `BEB`, `ERB`. Note: lines will be output in some random order since Elixir processes run non-deterministically. Counts across one line are in Peer number order, 0 to 4 above.

15 **Your peers MUST NOT do all the broadcasts first and then do all the receives**. Rather Peers must *interleave* the broadcasting of messages with the receiving of broadcast messages from other peers, which is a more realistic scenario. The implementation of how to do interleaving is left for you to design and implement.

16 Note: the value **0** and the atom **`:infinity`** have special meaning in the **`after`** clause of a `receive`. Read about Elixir's **`Process.send_after`** function also.

17 Run your system for the following message requests:
(i) `{:broadcast, 1000, 3000}`
(ii) `{:broadcast, 10_000_000, 3000}`
(iii) one request of your own that you think is informative/interesting.

18 *When broadcasting we often try to maximise throughput - the rate of messages are received by peers and the throughput for each peer* – however your goal here is to describe the rationale for some (any) design and provide explanations for its behaviour (suggesting improvements), not to develop an 'optimal' solution. Don't spend too long on this version!

BROADCAST 2 – PL Broadcast

19 Copy your solution for version 1 to a new directory for version 2. You may find it useful to draw a diagram of the hierarchy/configuration for your versions 2 onwards.

20 Now adapt broadcast 1 to use **`Perfect_p2p_links`** modules (**`PL`**). The hierarchy of broadcast 2 should be:

```
8   Broadcast           // top level node
9     Peer0             // node 0 with sub-modules Peer, PL, Client
10      PL, Client      // modules in node 0 - created by Peer
11    ...
12    PeerN-1           // node N-1 with sub-modules Peer, PL, Client
13      PL, Client      // modules in node N-1 - created by Peer
14
```

21 The `Peer` module is now responsible for spawning local Elixir processes for the PL and Client module. You may also wish `Peer` to communicate with its parent process (i.e. Broadcast). To spawn a process locally call **spawn**, not **Node.spawn** omitting the node parameter.

22 *One way* to connect PL modules to each other is for each `Peer` to send the Elixir process-id of its PL module to `Broadcast` and for `Broadcast` to send a *suitable* `:bind` message to each PL module. You are welcome to use a different design.

23 Once `Client` is created and bound to its `PL` module, `Client` takes on the role of the testing module in the system (broadcasting messages and receiving them in an interleaved way). `Client` communicates with other `Client` modules using its `PL` module (with `BEB` or with `ERB` in later systems)

24 PL modules should communicate with other PL modules using Elixir's `send` and `receive`.

25 Run your system as for broadcast 1, adding an interesting request of your own. Compare your results with broadcast 1 and explain them.

BROADCAST 3 – Best Effort Broadcast

26 Copy your solution for broadcast 2 into a new directory for broadcast 3.

27 For this version use **`Best_effort_broadcast`** (**`BEB`**) modules for the broadcasting. The hierarchy for broadcast 3 system should be:

```
1   Broadcast
2     Peer0
3       PL, BEB, Client
4     ...
5     PeerN-1
6       PL, BEB, Client
```

28 Run your system as for broadcast 1, adding an interesting request of your own. Compare your results with previous versions and explain them.

BROADCAST 4 – Unreliable Message Sending

29 Copy your solution for broadcast 3 into a new directory for broadcast 4.

30 For this version add a reliability parameter to the **`PL`** modules to simulate unreliable message passing.

31 Use **`Lossy_P2P_links`** (**`LPL`**) modules instead of **`PL`** modules. **`LPL`** should be like **`PL`** but parameterised with an integer reliability percentage from 0 to 100. 100 sends all messages (100% reliable). 0 drops all messages (100% unreliable - broken link). 20 sends *approximately* 20% of the messages. The hierarchy for this version should be:

```
7   Broadcast
8     Peer0
9       LPL, BEB, Client
10    ...
11    PeerN-1
12      LPL, BEB, Client
```

32 Run your system with suitable requests and reliability values of your own.

BROADCAST 5 – Faulty Process

33 Copy your solution for broadcast 4 into a new directory for broadcast 5.

34 For this version make `Peer3` terminate itself after 5 milliseconds. You can use a different timeout value if it helps. You can also manually perform a `kill -9` on the Peer3 Linux process if you wish.

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

36 Run your system with suitable requests and reliability values and fault-injections and understand your results. You can also experiment with terminating several processes.

BROADCAST 6 – Eager Reliable Broadcast

37 Copy your solution for broadcast 5 into a new directory for broadcast 6.

38 For this version use **`Eager_reliable_broadcast`** (**`ERB`**) modules. The hierarchy for this version should be:

```
13 Broadcast
14   Peer0
15     LPL, BEB, ERB, Client
16   ...
17   PeerN-1
18     LPL, BEB, ERB, Client
```

39 Run your system with suitable requests and reliability values and fault-injections and understand your results.

Well done on completing Lab3!

Additional Optional Exercises

40 (**Optional**): Implement and evaluate lazy reliable broadcast.

41 (**Optional**): Add support for routing over a multi-hop network and test for the example network in lab2.

42 (**Optional**) Distribute analyse the results of running **`Broadcast`** on a network of lab computers. You can do this (i) manually by *ssh*-ing into each computer (possibly using **`screen`** if available) and running a suitably named elixir node or (ii) running **`ssh`** commands (plus *ssh public-keys*) with your own shell scripts (including using *ssh public-keys*) or (iii) by using software like **Ansible** if available. You will need to adapt your node creation code. You might also need to open ports if a firewall is in place.

43 (**Optional**) Distribute, run and analyse the results of running one or more **`Broadcast`** solutions on one or more VMs. If you have access to a global VM service then distribute your nodes on VMs in different parts of the world. You will probably need to open ports in firewalls.

44 (**Optional**) Use Docker Swarm or Kubernetes for distributed execution.