Distributed Algorithms 60009

Lab 2 – Flooding Algorithms

https://www.doc.ic.ac.uk/~nd/dac/lab2.pdf

01 Lab 2 aims to develop your skills in Elixir, particularly message passing and functional programming.

02 The exercises are somewhat *vague*. This is intentional - you are required to make you own choices on how to refine and implement various aspects of the exercises. Have fun!

03 Note also that Elixir solutions WILL NOT BE provided for the exercises, so you will need to do the exercises.

04 You can work on the exercises with a classmate.

05 You'll need to start to use the online Elixir documentation and other guides to learn Elixir, look up suitable functions and build on your knowledge and skills from lab1.

06 Flooding is a simple algorithm to send a message to all nodes on a multi-hop network, like a sensor network. More usefully, it can be used to build a rooted spanning tree. For this exercise develop 5 versions, copying and then changing each version as you go. Use mix to create your project.

07 A **Makefile**, a **Helper** module, and skeleton **Flooding** and **Peer** modules are available in http://www.doc.ic.ac.uk/~nd/dac_src/lab2_files.tgz

FLOODING 1 – Single-hop

08 Write version 1 for a single-hop network - a fully connected network.

09 Write a module **Flooding** to create 10 peer processes (each running on its own node) that passes to every peer process, a list of all the other peers that the peer can send to (its neighbours). Peers should include themselves as a neighbour.

10 To start, Flooding should send a **:hello** message to the first peer.

11 Write a module **Peer** to forward the first **:hello** message that it receives to all its neighbours. **Peer** should count all **:hello** messages that it receives.

12 To see what's happening, **Peer** should print out the count of **:hello** messages that it received after a 1 second message timeout[1], for example:

```
Peer <peer-id> Messages seen = <int>
```

13 To identify a peer, you can either print the peer's Elixir process-id or use your own integer numbering scheme for peers (or print both!)

14 Run your system using the supplied **Makefile**.

[1] The **receive** statement can have an optional **after** timeout clause.

15 Question. The algorithm you've written is once-only, you can't broadcast a second message to all peers. How would you fix this? You don't need to implement this.

16 Question. How can you reduce the number of messages a peer sends? Again, you don't need to implement this.

FLOODING 2 – Multi-hop

17 Copy your solution for flooding 1 into a new directory for flooding 2.

18 For version 2 change module **Flooding** to create a multi-hop network with the following configuration. You shouldn't need to change module **Peer**.

```
bind(peers, 0, [1, 6])     # peer 0's neighbours are peers 1 and 6
bind(peers, 1, [0, 2, 3])
bind(peers, 2, [1, 3, 4])
bind(peers, 3, [1, 2, 5])
bind(peers, 4, [2])
bind(peers, 5, [3])
bind(peers, 6, [0, 7])
bind(peers, 7, [6, 8, 9])
bind(peers, 8, [7, 9])
bind(peers, 9, [7, 8])     # peer 9's neighbours are peers 7 and 8
```

19 Here **bind** is a function (that you need to write) that binds a peer (2nd parameter) to some of its neighbours (3rd parameter). **peers** is the list of all peer processes.

20 Draw the multi-hop network that this configuration describes. All hops above are bi-directional, i.e. if Peer1 is a neighbour, 1 hop away from Peer 0, Peer 0 is a neighbour of Peer 1. Check that there are 6 hops between Peer 4 and Peer 8.

21 Run your system and check the results.

FLOODING 3 – Spanning tree

22 One use of flooding is to dynamically construct a rooted spanning tree, where the initiating process acts as the root of the tree.

23 Copy your solution for flooding 2 into a new directory for flooding 3.

24 For this version change **Peer** to remember from whom it receives its first (:**hello**) message[2] i.e. the peer's parent. Change **Flooding** if required also.

25 Extend your output to print the parent, for example:

```
Peer <peer-id> Parent <peer-id of parent> Messages seen = <int>
```

26 Run your system and check your results.

27 (**Optional**) Adapt the network configuration to further check your algorithm, e.g. connect peers in a pipeline.

28 Although this algorithm builds a spanning tree, it may not be the best possible for a particular configuration - there are algorithms that build the shallowest possible spanning tree.

[2] Just include the parent's id in the message.

FLOODING 4 – Data Collection - store child counts

29 The inverse of flooding is data collection where the root peer collects data from other peers, sometimes called *converse cast*.

30 Copy your solution for flooding 3 into a new directory for flooding 4.

31 For this version change **`Peer`** to count the number of child peers a peer has[3]. Change the output to print the number of child peers the peer has, for example:

```
13  Peer <peer-id> Parent <peer-id of parent> Children = <int>
```

FLOODING 5 – Data Collection - summary function (sum)

32 Now extend the system to collect data from the peer network. Assume each peer has a single value, for example, a sensor value.

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

34 Rather than forward all child values, each parent peer should apply a function to the values it receives from its child peers as well as its own value (a *reduce* operation). For this exercise just apply a *sum* function. Effectively you'll be producing the sum of values held in the network.

35 If the peer is a *leaf peer* it should just forward its value (e.g. a small random value) to its parent. Non-leaf peers should collect all child peer values and then forward the result of applying the summary function (sum) to their parent.

36 Change **`Peer`** to print the random values chosen after the number of children. Change **`Flooding`** to receive the final result and print it out.

37 Run your system and check your result.

Well done for completing Lab 2!

[3] Send to the parent peer, a new **`:child`** message and count the number of **`:child`** messages received.