LSINF2345 Report Project

COLETTE Sacha [17841600] COLIN Louis [22541600]

December 2020

1 Global explanation of the architecture

The figure 1.1 is a visual representation of our architecture. Four kinds of threads run in our architecture : the main thread, the node threads, the sender threads and the receiver threads. When a node is created, three threads are created (the node thread, the sender and the receiver).

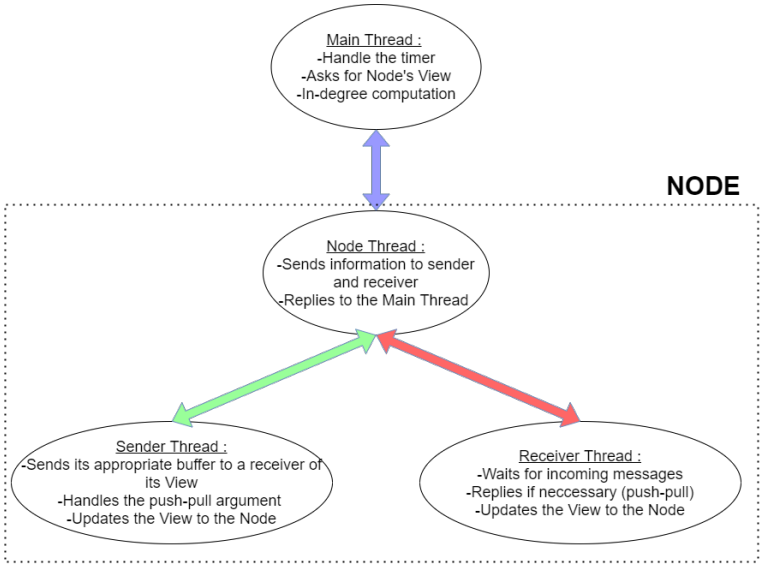


Figure 1.1: Global representation of our architecture

Here is an explanation of the role of the different threads :

1.1 The main thread

The role of the main thread is to manage the timeline of the scenario. To do that, a counter is used in order to know the current cycle number. Every 20 cycles, the main thread sends an "ask view" message to every alive node threads. All the node threads will reply to the main thread with their current view. The main thread will then be able to compute the average of the in-degree and its standard deviation (see section 3).
During the "growing" phase, the main thread initializes the new nodes (creation of a node thread, a sender thread and a receiver thread).
During the "Time to crash" phase, the main thread sends a "dead" message to the chosen nodes. Those nodes will end their process.
During the "recovery" phase, the main thread increases the network and it initializes new nodes.
At each cycle, the main thread sends a "time" message to the node thread. This message means that the node has to send its view to a neighbor.
At the end of every cycle clock, the main thread sends a message "time" to every node thread. When a node receive this message, it knows that it has to share its view with a neighboring node.

1.2 The node thread

The node thread is the junction between the main thread, the sender thread and the receiver thread. Its role is multiple :

- Relay the messages sent by the main thread : When a "time" message is received by a node thread, it transfers it to the sender thread with a copy of the current view. Thanks to this mechanism, the sender knows that it has to send the received view to a neighbor.
  When a "dead" message is received by the node thread, it will transfer it to the sender thread and the receiver thread and finish its current execution.

- Maintain the current view of the node. When one of the other thread type (main, sender or receiver) needs to have the current view of the node, they send a message to the node thread and will receive a response containing its current view. This current view is updated when the sender or the receiver thread sends a message to the node thread to update the view.

1.3 The sender thread (active thread)

When the sender thread receives the current view from the node thread, it knows that it has to send its current view to one of these neighbors (more especially, it will send its current view to the receiver thread of one of its neighbor). If the "push-pull" parameter is set to true, then the sender will wait 1.5 seconds for a response of its neighbor. After receiving the view of its neighbor, it will update its current view and send a copy of it to the node thread. In this way, the view is updated.
If the sender thread receives a "dead message", it finishes its execution.

1.4 The receiver thread (passive thread)

The role of the receiver thread is to receive the current view of one of its neighbor. If the "push-pull" parameter is set to true, it will ask its current view to the node thread and send it to its neighbor. After that, the current view is updated with the new informations sent by its neighbor. This new view is then sent to the node thread. In this way, the view is updated.
If the receiver thread receives a "dead message", it finishes its execution.

1.5 Description of a sending-receive process

The figure 1.2 is a visual representation of the sending-receive process.

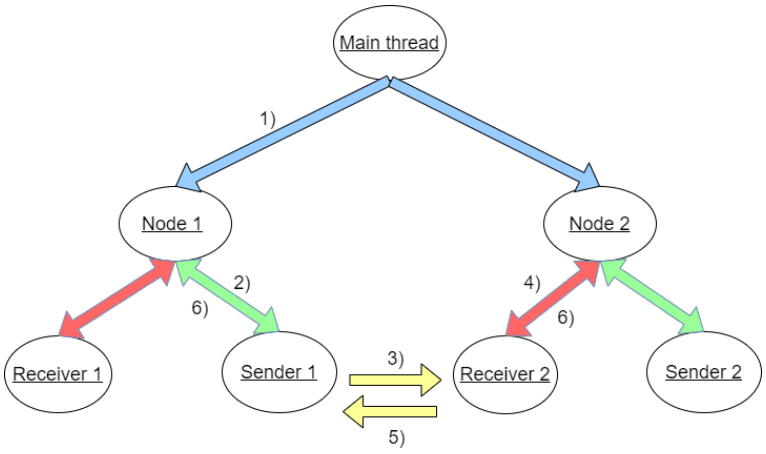


Figure 1.2: Visual representation of a sending-receive process

Here is an explanation of the figure 1.2:

1. The main thread sends a "time" message to the node thread 1

2. The node thread 1 sends its current view to the sender thread 1

3. While receiving a View, the sender 1 knows that it has to send it to one of its neighbor (receiver 2).

4. Because we are in a push-pull view propagation, the receiver 2 asks for the current view of the node 2 in order to reply to the sender 1.

5. The receiver 2 replies to the sender 1 with its current view.

6. The sender 1 updates its current view with the received view and sends it to the Node 1. The receiver 2 also updates its current view with the view sent by the sender 1. The updated view is sent back to the node 2

2 Nodes joining the network for the first time

When nodes will join the network for the first time, they will be put in a double linked list as represented on the figure 2.1. In this example, the two nodes at the extremity have one neighbor while the node in the middle has two neighbors. On this representation, a node is represented as : $Node = \#\{id := ID, list\_neighbors := List\_neigh\}$ and the list of the neighbor is represented as : $List\ Neighbors = [\#\{id\_neighbors := ID1, age\_neighbors = Age1\}, \#\{id\_neighbors := ID2, age\_neighbors = Age2\}, ...]$. Thanks to the list of nodes, the main thread will be able to initialize the node thread, the sender thread and the receiver thread for every nodes. Note that the ID (in green) will be the process ID of the receiver.
In our implementation, the linkedlist with the 128 nodes is created at the beginning but all the nodes are not added in the network. We divide the linkedlist in two sublists and add only the first 40% of the nodes at time clock 0. The others nodes are added during the growing phase.

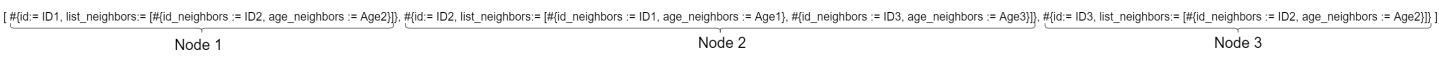


Figure 2.1: Example of a linked list of three nodes

Note : There is a unit test to confirm the correctness of the add function in the linked list on the Erlang file "test_structure.erl"(on the github).

3 In-degree computation

The in-degree computation is made by the main thread. After requesting the current view of all the nodes, the main thread will browse the views of all the alive nodes and compute the in-degree of them. The in-degree of an alive node P is is the number of alive nodes in the whole network that contains P in their views. The average is then calculate as:

$$Indegree\ average = \mu = \frac{\sum_{i=1}^{N} Indegree_i}{N} \tag{3.1}$$

With N, the number of alive nodes and $Indegree_i$, the in-degree of the alive node 'i'. The standard deviation is calculated as :

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(Indegree_i - \mu)^2}{N}} \tag{3.2}$$

4 Discussion of the results

The figure 4.1 is the boxplot of the in-degree in the healer and the swapper deployment. Note that those results has been obtain with a pushpull view propagation and that the peer selection is performed randomly among the alive neighbors (View propagation = pushpull and Peer selection = random).

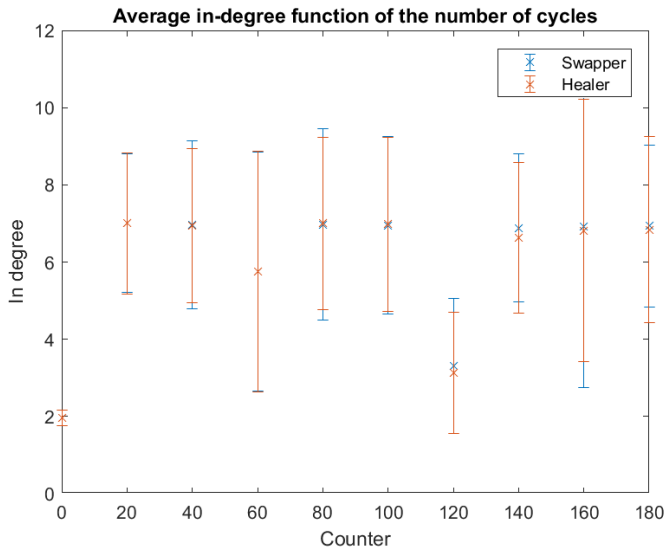
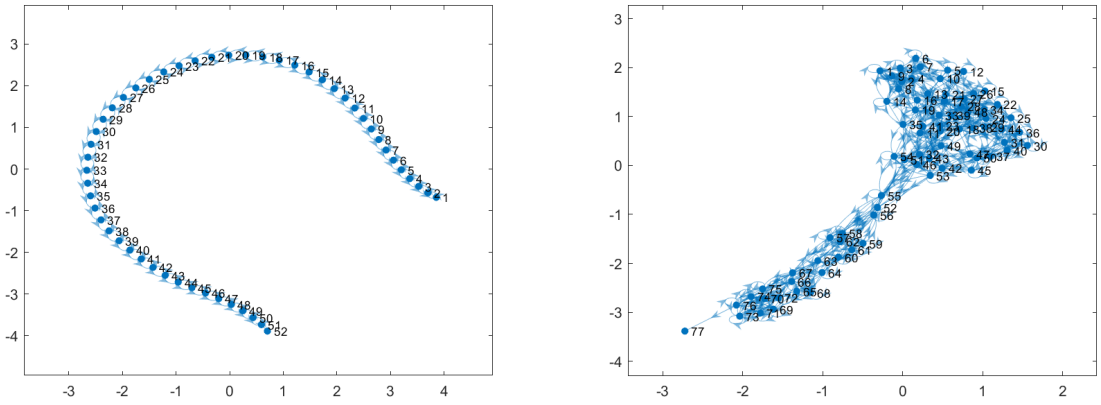


Figure 4.1: Boxplot of the in-degree of the healer and the swapper deployment

Let's comment our results : At the cycle 0, the mean of the indegree is close to 2 and the standard deviation is really narrow. Indeed, at the cycle zero, all the nodes in the network have two neighbors (the one at the right and the one at the left in the double linked list). Only the node at the first index and the last index have a single neighbor (cf. figure 4.2a). This explain why the mean is not exactly 2 and why the standard deviation is not exactly 0. From the cycle 20 up to the cycle 100, the average indegree and the standard deviation remain more or less constant. During this part, the average indegree tends to 7 because all the node have a view size that tends to 7 that is the maximal view size (cf. figure 4.2b). At the cycle 120 (crash of the nodes), the average indegree drops to 3. This can be understood. Indeed, the in-degree of an alive node P is is the number of alive nodes in the whole network that contains P in their views. Because 60% of the nodes crashed, the remaining nodes will still have the crashed nodes in their view. That is the reason why the average indegree drops. At the cycle 160, we can observe an increase of the standard deviation. This can be explained because the "recovery" phase was made at cycle 150. At this moment, the new nodes only have one neighbor (P) on their view. This will of course increase the standard deviation since there will be a huge difference between the indegree of a random alive node and the indegree of this node P. However, we can observe that at the end of the process (cycle 180), the standard deviation decreases a bit. Globally, the behavior of the swapper and the healer are really close to each other.



(a) Clock cycle = 0

(b) Clock cycle = 40