Distributed Mutual Exclusion: Token Passing on a Tree

Marian Alexandru Diaconu, Valentino Armani

October 2019

This report is meant to explain the main architectural choices of the implementation of Raymond tree based algorithm for distributed mutual exclusion [1] using AKKA framework and its testing. Details about the main procedures and requirements of the algorithm are omitted because well described in the original paper and can also be investigated in the source code. The implementation is based on a network of N nodes which interact each other using messages rather than shared memory.

1 Initialization

The first phase is used to create a network and initialize the participants. The initialization consists in assigning to each node a list of neighbors. This is done using an **InitNode** message which contains the required data. The generated tree used to test the implementation can be visualized in figure 1. Subsequently, a **HolderInfo** message is used to signal to a node that it is the initial holder of the token. This node has also to start spread this information to the rest of the network to allow each participant to set its holder variable. To do so the same class message is used.

2 Requesting the token

In order to simulate the fact that a node wants to access the critical section, a **StartRequest** message is used. Given that a node should be present only once in its *request_q*, if it wants to access multiple times the critical section, a variable called *selfRequests* is used to deny excessive usage of the token. Basically, it represents a counter. Each time a node receives a **StartRequest** message the counter increases by 1. When it exits the critical section *selfRequests* decreases by 1 and if it is greater than zero, the node will add itself in its queue, again, allowing neighbors requests to be processed before its future ones.

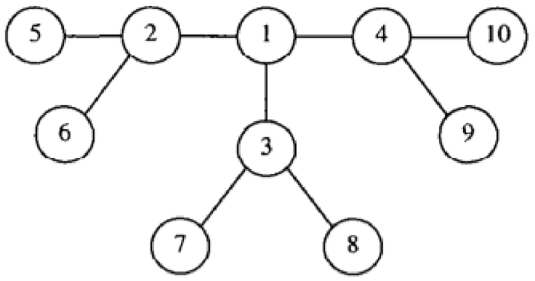

Figure 1: Testing Topology

3 Granting the token and accessing critical section

The token is granted when a node receives the **Privilege** message. The behavior of the nodes is described in the original paper and allows the correct transmission over the underlying network of the token. The access to the critical section instead, is simulated by making the node waiting for a limited amount of time specified by the static local variable *CSDURATION*. Basically, the node loops until the specified time have passed.

4 Node Failure and Recovery

The failure of a node is simulated using a message called **Fail** and a local Boolean flag named *failed*. The message is used to signal, from the outside, that the node has to fail, that is, forget all the information about the algorithm status. When a node fails the flag is turned from False to True. This allows the node to signal to itself that it should ignore **StartRequest, Request, Privilege, Restart** and **Advise** messages. Before the failure, the node schedules a message of type **Recover** to be sent to itself after a limited amount of time defined by the static local variable *FDURATION*. When the **Recover** message is received by the node, the recovery procedure starts, which consists in contacting the neighbors, asking for information about their state (through **Restart** and **Advise** messages) which is finally used to infer the state of the failed node as described in [1]. Moreover, given that the failure of a node is simulated through the reception of a message that starts the scenario, a given node cannot fail during the critical section execution. Another observation for the reader is that the variable *selfRequests* is assumed to be permanent, which mean that even in case of crash, it preserves its value. This is done because, during the recovery phase, neighbors cannot provide information to be used, by the failed node, to infer if, before the crash, it wanted to access the critical section or not. Without the usage of such variable, it is not possible to understand if the failed node has requested the token for itself or on behalf of others.

5 Testing

In order to test the implementation of the algorithm, all the nodes print some data about the actions that they perform. The most important information which is printed and which the reader should be aware of is:

- Send Request
- Send Privilege
- Failure Start (along with the lost information)
- Ignored Messages during Failure
- Receive Restart
- Receive Advise
- Failure End (along with the information inferred)
- Critical Section Enter
- Critical Section Exit

This information allows to test the overall behaviour of the system. More precisely the printed information allows to test if:

- The nodes send and receive messages correctly
- The nodes that have to enter the critical section do so
- The node that has to fail it behaves as specified by the protocol. In details, the node that fails prints the information hold before the failure and also the messages that are ignored while failed. This allows to understand if the inferred information after the recovery is correct or not.

References

[1] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, pages 61–77, 1989.