Akka Implementation of a Tree-Based Algorithm for Distributed Mutual Exclusion

Alessandro Cacco
(mat.203345)
dept. of Information Engineering and Computer Science
University of Trento
alessandro.cacco@studenti.unitn.it

Enrico Saccon
(mat.207531)
dept. of Information Engineering and Computer Science
University of Trento
enrico.saccon@studenti.unitn.it

***Abstract*—An agent based implementation of the algorithm proposed by Kerry Raymond in 1989[1] to solve the mutual exclusion problem exploiting a tree structure.**

I. Introduction

A common problem in distributed systems is distributed mutual exclusion, for instance the problem of accessing consistently a critical section.
The goal is to achieve mutual exclusion on a shared resource in an optimized manner. In the algorithm presented by Kerry Raimond, the distributed mutual exclusion in a network of N nodes can be achieved with $O(\log N)$ message exchanges[1]. This result is possible thanks to the network tree structure, hence the name Tree-Based Distributed Mutual Exclusion (TBDMX).
In order to implement a simulation of the algorithm the Akka framework has been used, in which nodes are represented as concurrent actors.

II. Protocol Overview

The protocol achieves mutual exclusion by means of a regulated token passing. Each node keeps a request queue of the requests received (i.e. saving the respective sender id), which is used in combination with the following state variables and flags:

- `HOLDER`: variable indicating the token holder w.r.t. the the node neighbourhood, that is, which node to send a token request to if needed. If the node is the holder, then a reference to `self` is stored;
- `ASKED`: set if the node has sent out a request message in order to gain token privilege for an enqueued requesting node, which can either be itself or another directly connected node;
- `USING`: if a node needs to access the critical section and obtains the token, then this flag must be set while the actual resource is being used, and unset as soon as the node is done using the token;

During the execution of the protocol a node may experience sudden crashes, but the protocol is able to rebuild the node information in order to continue the normal operations. A node who needs the token, either for itself or for a enqueued requesting neighbour, asks it to the `holderNode`, who enqueues the request and eventually respond with a token after, obtaining it from its token holder if needed. In case of crash, a node asks its neighbours for information about the circumstances, rebuilding its state.

III. Implementation

This section presents the main features of the implemented TBDMX protocol, both from modeling and practical perspectives. Note that all the features make use of Akka message passing features, which automatically manages concurrency between nodes (`ActorRef`) functions. All the controller commands and protocol primitives (explained in Sections III-C and III-D) are implemented by means of Akka messages.

A. TBDMXNode and TBDMXController

The main objects implied in the protocol execution are `TBDMXNode` and `TBDMXController`. The former models the architecture of a node implementing the TBDMX algorithm, shown in the state diagram in Figure 1, with all the message passing and the abstractions of the main protocol management procedures. The latter is a centralized controller entity whose duties are:

- Instantiating the simulation, correctly initializing all the nodes. A custom network structure is loaded from the *tree.conf* file, which can either be a simple list of node neighbours (each line corresponding to a node), or an xml file specifying the nodes and the edges between them.
- Loading all the commands from the *command.conf* file, managing all the nodes by sending them commands described in Section III-C.

B. Protocol assumptions

The version of the protocol presented for this project works on the important assumption that two nodes cannot crash simultaneously, as the complete neighbourhood of a recovering node is needed to rebuild the node information. In the unfortunate case of two neighbour nodes crashing, the correctness of the protocol is not guaranteed.
Furthermore, no duplicate request is made from a single node, meaning that nodes request queues should never contain duplicate id entries and that a node should not make more than a critical section request at a time.
Finally no node may crash during its critical section.

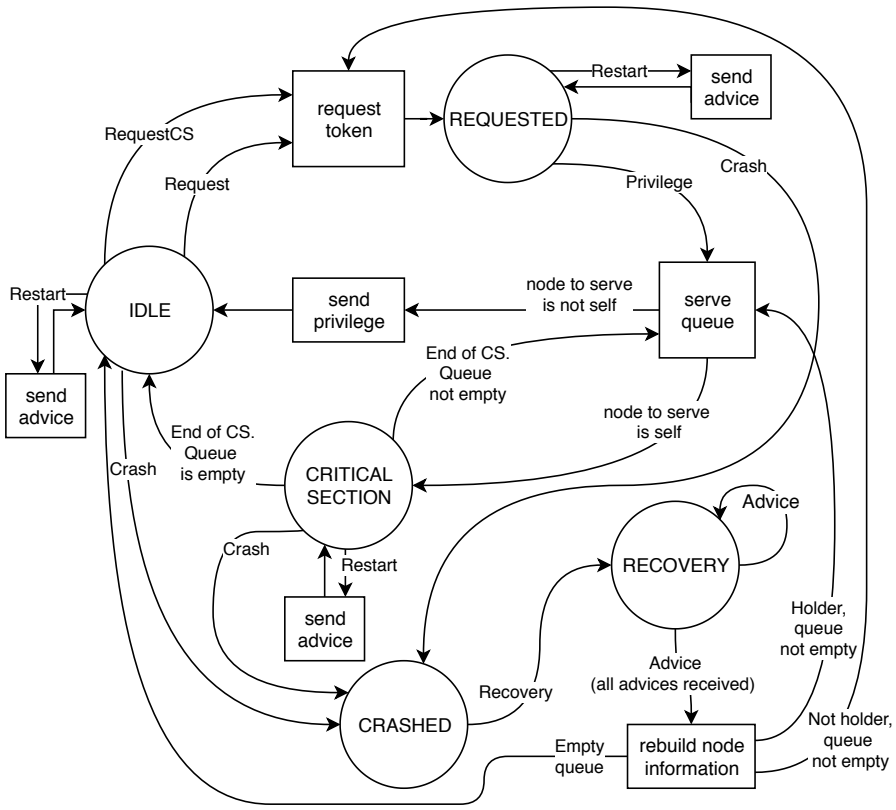

Fig. 1: State diagram of a node. Circles and Rectangles respectively indicate states and procedures. Arcs labels are messages and conditions.

C. Controller commands

- `SetNeighbors`: The controller passes a list of neighbours to each node which saves such information inside a local variable named `group`.
- `ImposeHolder`: The `ImposeHolder` message is sent to the first owner of the token that will then broadcast the information to its neighbours. These nodes will then send a `BroadcastHolder` message to their neighbours recursively.
- `BroadcastHolder`:
  Upon receiving a `BroadcastHolder` message, the node sets the `holderNode` variable to the sender of the message, since it represents the next node a message should be sent to in order to request the token. The `BroadcastHolder` message is then broadcasted to all the neighbours, except for the one the initial `BroadcastHolder` was received from. Each neighbour will therefore performe the same operations recursively, effectively flooding the network.
- `Crash`: Upon receiving a `Crash` messages, a node simulates a crash deleting its request queue and losing knowledge of the token holder. If the node was already down, then no action is taken. To know whether a node is down or not, the boolean variable `crashed` is set to `true` in the first case, to `false` otherwise.
- `Recovery`: Upon receiving a `Recovery` message, a crashed node starts recovering. This implies sending a `Restart` message to all its neighbours, setting the `crashed` flag to `false` and setting the `recovering` flag to `true`.
- `RequestCS`: Upon receiving a `RequestCS` message, the node will attempt to enter the critical section. If it is the holder of the token, then it can enter the CS, otherwise it will send a `Request` message to its `holderNode`.
- `SaveLog`: Upon receiving a `SaveLog` message, the node will close the buffer onto which it was writing the log.

The first three commands are used exclusively for the initialization of the network, informing all the nodes of their respective neighbourhood according to the tree configuration loaded, while the rest of the commands are used to orchestrate the simulation (excluding `SaveLog`). The command interface (i.e. the allowed commands in the *commands.conf* file), effectively abstracting these messages, is illustrated in Table I. Note that the initial holder node is specified in the first row of the *commands.conf* file, while the following lines should contains the commands to execute.

D. Protocol primitives

- `Request`: Upon receiving a `Request` message, if the node is not crashed or recovering and it is the holder of the token, then it calls the function `serveQueue()`,

TABLE I: TBDMXController *commands.conf* allowed commands

Command	Argument 1	Argument 2	Description
`request`	Target node id	Time duration needed in the critical section	Node needs to access critical section for the specified time
`crash`	Target node	n/a	Node crashed
`recovery`	Target node	n/a	Node recovers from crash, starting the recovery procedure
`wait`	Time delay in milliseconds	n/a	Wait the specified time before sending the next commands

otherwise it sends a `Request` to the `holderNode` setting the `asked` flag to `true`. Instead, if the node is recovering, it saves the request in a secondary queue (`recoveryQueue`) which will be merged later when finishing the recovery.

- `Privilege`: The `Privilege` message is used to pass the token. Upon receiving a `Privilege` message, if the node is recovering, then the node memorizes having received the token by setting the flag `recoveryHolder` to `true` and this will be later used to set the `holder` flag. If instead the node is not recovering, then the flags are set as follows: `holderNode` is set to `getSelf()`, `holder` to `true`, `asked` to `false` and finally it calls the `serveQueue()` function.
- `Restart`: Upon receiving a `Restart` message, the node sends back to the recovering node an `Advice` message.
- `Advice`: An `Advice` message contains important information for the recovery procedure such as whether the crashed node was in the queue, if it was the `holderNode` of the node sending the `Advice`, whether the sending node had asked for the privilege and finally a counter which allows to avoid starvation. The actions taken on the receipt of such message are explained in Section Section III-E.

E. Crash and Recovery Procedure

When a node crashes, it loses information such as the `requestQueue` and the `holderNode`, while it maintains knowledge about its neighbours. The goal of the recovery procedure is to actually recover the lost data as consistently as possible w.r.t. to the previous state.
First of all a node stays in a crashed status until a `Recovery` message is received. Upon receipt, it sends a `Restart` message to all its neighbours to gather information. They will respond with an `Advice` message containing the following fundamental information:

- `holder`: a boolean variable that says whether the crashed node was the holder for the sending node;
- `asked`: a boolean variable that states whether the *sending* node asked for the token;
- `inRequestQueue`: a boolean variable that says if the crashed node was in the `requestQueue` of the node sending the `Advice`;
- `AdviceCounter`: a counter used to avoid starvation. During recovery the order of the requests of the queue is not assured to be the same as before crashing. To avoid the possibility of having some nodes answering constantly before others causing the later's starvation, the counter is used with the goal to order the receiving `Advice` messages. Each time an `Advice` message is sent, the counter is incremented and is reset upon receiving a `Privilege` message.

When the recovering node has received an `Advice` from each neighbour, it can start to rebuild its lost knowledge.
If an `Advice` presents both `asked` and `holder` set to `true`, it means that the node sending the `Advice` made a request to its `holder`, which is indeed the crashed node, and for such reason it needs to be added to the `requestQueue`.
If all `Advices` have `holder` set to `true`, then the node was the holder of the token.
If one of the `Advices` states that the crashed node is not the holder, then the node sending such `Advice` is the holder for the crashed node. Moreover, if the flag `inRequestQueue` is set to `true`, then the crashed node had made the request and its `asked` flag should be set to `true`.
While a node was crashed or was recovering, some other nodes might still send `Requests` or `Privileges` messages to it. During the recovery phase, the former are stored in a temporary queue called `recoveryQueue`, while the latter imply that the `recoveryHolder` variable must be set to `true`. The `recoveryQueue` is later merged with the `requestQueue`, checking and removing any possible duplicate entries. The second is dealt with by setting the right flags for the node to become the holder, marking that the recovery is completed and the token is available again.
At the end of the procedure, a check is performed on the node in order to verify its role: if the node is the token holder, then the next element in `requestQueue` is served, otherwise if it is not the holder, its queue is *not* empty and no request has been sent out yet (i.e., `asked=false`), then the node sends the request to the `holderNode`.

IV. CONCLUSION

The protocol implementation for the purpose of this project passed our tests and works as expected w.r.t. the initial assumptions. The graphml input and the automatic command loading allows further testing with ease, and the individual logging of each nodes gives many insights in the execution of the protocol.

REFERENCES

[1] Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. **7**(1), 61–77 (1989). https://doi.org/10.1145/58564.59295, `https://doi.org/10.1145/58564.59295`

APPENDIX A
EXECUTION EXAMPLE

The following is a valid example of `tree.conf` for the framework:

```
1 2
0 3 4
0
1
1
```

This tree configuration corresponds to the simple tree shown in Figure 2.

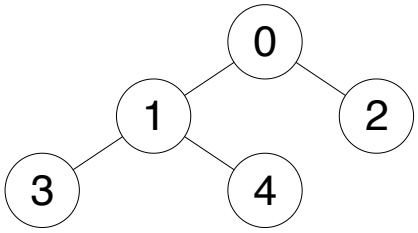

Fig. 2: Sample tree.

A valid `commands.conf` file for this tree may be:

```
0
request 3 1000
wait 100
request 2 2000
request 4 500
crash 0
wait 2000
recovery 0
request 1 100
```