Distributed Mutual Exclusion:

Implementation of Raymond's Tree-Based Algorithm

Andrea Tupini (MAT. 194578)*, Mario Berti (MAT. 207509)†
Master of Science in Computer Science,
University of Trento
Email: *andrea.tupini@studenti.unitn.it, †mario.berti@studenti.unitn.it

I. Introduction

RAYMOND'S algorithm [1] is used to ensure mutual exclusion from a critical section (ie, accessing a resource) in a distributed system. It has an upper average complexity of $O(\log N)$, and the upper bound of messages needed for a node to access the critical section is of $2(D-1)$, where D is the diameter of the topology, and in this case, D can also be taken as the maximum distance between two nodes in the network configuration. For instance, with respect to Figure 1, we need $2(5-1) = 8$ messages to get the token from node *5* to node *10*.

It's $O(\log N)$ complexity makes it a very good algorithm for distributed mutual exclusion. In this report we discuss its implementation using Akka [2], a framework to develop distributed applications using an actor based abstraction.

II. Topology

For the network's topology we chose to use a *radial* one (see Figure 1 for the specific structure).

When creating the *actors* we manually tell them who their neighbors are. As such, the topology of the network is fixed.

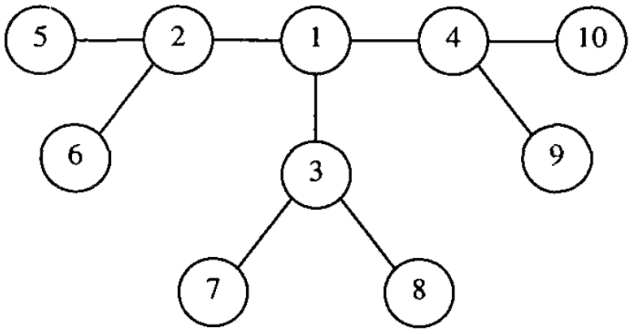

Fig. 1. The network topology used for this project. Image taken from [1, Figure. 7].

III. The *ActorSystem*

When creating the `ActorSystem` we create only one `ResourceActor` (section IV), and a `NodeAct` (section V) for each node in the topology mentioned in section II, passing a reference to the previously created `ResourceActor`. Once all the necessary `NodeAct` have been created, we send a message to each informing them who their neighbors are (by passing a list of `ActorRefs` representing said neighbors). After this, the *Initialize* procedure is invoked: the token is assigned to a random node and a message is flooded to all the nodes, so that they know which of their neighbors is closest to the token.

IV. The *Resource* Actor

Actors will try to access the resource in a mutually exclusive way, and this resource is represented by a special actor called `ResourceActor`. Actors will access this resource by sending a message to `ResourceActor`. Because of this abstraction, it would be easy to extend the message sent to `ResourceActor` to include an arbitrary command to be executed by the resource. Although, currently, when the resource receives the message, it just waits for a while (simulating the execution of a process), and then sends back a message to the actor accessing the resource informing, it that the critical section computation has been completed (again, this message can also be easily extended to contain an arbitrary result from the computation).

V. *NodeAct* Actor

This is the actor implementation for the nodes in the network. It contains all the functionality needed to execute the mutual exclusion procedure, and also to recover from failures.

For this implementation, we tried to stick as much as possible to what is exposed in [1]. All parts presented in the original paper remained the same, but we added some extra functionality to adapt it better to Akka's actors.

We proceed to present a description of each message received by `NodeAct`, and how it is handled. We'll separate these in two sections: those messages used for the mutual exclusion functionality and those used to recover from failures. The explanation of how the messages are processed is given from the point of view of the actor currently receiving the messages.

A. Messages for mutual exclusion

1) RequestToken: This message is sent by an actor when it wants to take possession of the token. When an actor gets this message it adds the sender to its REQUEST_Q, only if it is not already there (no duplicate requests are allowed). It proceeds to ask for the token if needed:

Algorithm 1 Ask for the token if needed

```
1: if (this.holder does not point to this node and
       REQUEST_Q is not empty and
       we have not yet asked for the token) then
2:     this.asked = true
3:     send RequestToken message to this.holder
4: end if
```

The node then checks if it has the token. If it does and is not currently using it, it sends itself a `InvokePriviledgeSend` message to signal itself that the token is ready to be sent to the node at the head of REQUEST_Q.

Finally, the node checks if the node requesting the token is itself and if it already has the token. This is mainly used when the user requests a node who already has the token to enter the critical section (via the `UEnterCS` message).

Algorithm 2 Send token to itself if possible

```
1: if (this.holder points to this node and
       the sender of this message is this node and
       this node is at the top of its REQUEST_Q) then
2:     call the same method that handles SendToken
3: end if
```

2) SendToken: This is the message that stands for the token. Receiving this message means that the recipient is the new holder of the token, and the sender was the previous holder.

When a node receives this message, if it is at the top of its REQUEST_Q it will send itself a `EnterCriticalSection` message. If it is not, then it will send itself a `InvokePriviledgeSend` message to signal that it should relay the token to the node at the head of its REQUEST_Q.

3) InvokePriviledgeSend: This is a message that an actor sends to itself to indicate that the token should be passed on. At the time an actor sends this message to itself, it ensures that it is the holder of the token and that it is not using it.

The processing of this message is split in two: it first sends the token to the node at the head of REQUEST_Q, and then it asks this node to send back the token if there are still nodes in REQUEST_Q.

Algorithm 3 Send the token

```
1: if (this node is the holder of the token and
       this node is not using the token and
       this node is NOT at the head of REQUEST_Q) then
2:     this.holder = this.request_q.pop()
3:     this.asked = false
4:     send SendToken message to new this.holder
5: end if
```

Algorithm 4 Ask to send the token back

```
1: if (REQUEST_Q is not empty and
       this.asked is false and
       this node is not the current token holder) then
2:     send RequestToken message to this.holder
3: end if
```

4) EnterCriticalSection: This is a message that an actor sends to itself to signal that it can enter the critical section. This implies that the current node is the holder of the token.

When processing this message the actor ensures that $this.asked = true$, and then it sends an *AccessResource* message to the *ResourceActor*.

5) ExitCriticalSection: This is a message that the `ResourceActor` sends to the node accessing it once the *computation* of the critical section has finished. This message, informs the current node that it may now stop using the token and send it over to the next node in REQUEST_Q, if any. While processing this message, a node only sets its `this.using = false` and sends itself a `InvokePriviledgeSend` message.

B. Messages for failure recovery

To handle failures, a small extension is also made to the way a couple of the messages mentioned above are handled. To preserve consistency, if the node is currently processing a `RequestToken` or `InvokePriviledgeSend` message and the node is currently performing the *recovery process* then, said message is not processed, it is instead added to a *stash* and once the *recovery procedure* is complete the messages in this stash are prepended to this node's message queue, so that they're processed in the same order as they were originally received.

1) InitializeRecovery: A node sends this message to itself when it restarts after a crash. Receiving this message initiates the recovery procedure.

While processing this message, a node will set a special flag (`this.is_recovering`) signaling that it is currently performing the recovery procedure. It also resets all its internal state variables (who is the holder; if this node has asked for the token or not; if this node is using the token; and the nodes in the REQUEST_Q).

It then proceeds to setup the only data structure used to help with the recovery procedure. This is a set (`this.receivedAdvises`) where we will save which of the neighbors have sent an `Advise` message, and it is originally empty.

Finally, it sends a `Restart` message to all of this node's neighbors.

Note that, as per the assumptions in the project specification, a node may not fail if it is in the critical section nor when it is recovering. So if a `InitializeRecovery` message is received while the node is performing any of these two actions then it is simply ignored.

2) Restart: This is a message that an actor sends to all its neighbors to inform them that it has crashed. When processing this message, the node will simply respond with an `Advise` message.

3) Advise: The `Advise` message is always sent in response to a `Restart` one. The `Advise` contains all the information necessary for the actor who sent `Restart` to reconstruct its internal state. The `Advise` describes the relation between this node and its neighbor who sent it the `Restart`.

An `Advise` message contains the following information (`X` being the node who sent the `Restart` and `Y` being the node who is sending the `Advise`):

- Who is the holder according `Y`
- If `Y` has asked for the token
- Whether `X` is present in `Y`'s REQUEST_Q or not

When receiving an `Advise` message, the actor first adds the sender to its set (`this.receivedAdvises`) to track whether all neighbors have already sent an `Advise` or not.

Once all neighbors have sent an `Advise` then we have all the information we need and we can reconstruct the internal state of the current node. The state reconstruction algorithm is exactly the same as the one described in the original paper [1]:

- if this node has already received the token while performing the recovery procedure, or if all neighbors say that current node is the holder then this node's `this.holder` is set to point to itself
- if on the other hand one of the neighbors does **not** say that this node is the holder then `this.holder` is set to point to that neighbor
- if the current node is not the holder but it is an element of a neighbor's REQUEST_Q then this node's `this.asked` is set to `true`
- if a neighbor says that this node is the holder and that it (the neighbor) has asked for the token then said neighbor is added to this node's REQUEST_Q

The node then proceeds to clear the `this.receivedAdvises` set so that it can be reused for recovering from the next failure. The variable signaling that the node is in recovery (`this.is_recovering`) is then set to `false`.

Finally, we *unstash* all `RequestToken` and `InvokePriviledgeSend` messages that were received while performing the recovery procedure. These are prepended to the *head* of this node's message queue.

C. Notes about the recovery process

If the node which fails was part of its REQUEST_Q before failing, then the current algorithm to reconstruct the state is not able to determine whether this was the case or not. During the reconstruction of the REQUEST_Q, only neighbors are added. It's up to the node who failed to re-request the token if it needs to enter the critical section.

Also, our implementation differs from the one presented by [1] in one matter, when reconstructing the REQUEST_Q we keep a partial ordering of the nodes after the recovery is done, nodes who were in the REQUEST_Q before failing are guaranteed to be above nodes who send token request messages during recovery. This because the latter are added to the queue only after it is reconstructed.

D. Other messages

1) UEnterCS: This is a message that is manually sent by the *user* to signal a node that it should enter the critical section. When a node receives this message it will send itself a `RequestToken` message.

2) USimulateCrash: This is a message that is manually sent by a user to a node and it signals this node that it should fail and start the recovery process. A node processing this message will just send itself an `InitializeRecovery` message.

3) InvokePrintInternalState: Mainly used for debugging. When a node receives this message it will print its internal state: who is the holder, if this node has asked for the token, if it is performing the recovery procedure, and the name of the nodes in its REQUEST_Q.

4) SetNeighbors: This is a special message used during initialization of the nodes to tell them who their neighbors are. Basically, the message contains a set of references to other nodes. When processing this message, the node will set its neighborhood (`this.neighbors`) to be the nodes referenced in the message.

As said in section II the neighbors are defined manually beforehand.

5) Initialize: This is the initialization message that is flooded through the network so that all nodes know which of their neighbors is the nearest to the token. This message has a variable that states if it is the first `Initialize` message sent or not.

Basically, for the first node receiving it, the message will state that it is the first and this node will then proceed to set its `this.holder` to point to itself (meaning that this node is the holder of the token). For further nodes the message will state that this is not the first, so these will then set their `this.holder` to point to the node who originally sent this message to them.

Nodes will then proceed to send a new `Initialize` message to all their neighbors but signaling in the message that this is not the first being sent. In this way, a directed graph is constructed where the *directions* point at the neighbor closest to the token.

Before processing the message, all nodes will check if their `this.holder` has been set or not. If it has been set then it means that this node has already received an `Initialize` message in the past, so the current one will be ignored. In this way we prevent bouncing `Initialize` messages back and forth. If it has not been set then the message is processed as mentioned above.

VI. Results

Our implementation has the same results as those mentioned in [1]. In the worst case scenario we have that the amount of messages needed to access the critical section is $2(D-1)$.

To recover from a crash, the amount of messages needed is exactly $2K$ where K is the amount of neighbors.

References

[1] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 7, no. 1, pp. 61–77, Jan. 1989. [Online]. Available: http://doi.acm.org/10.1145/58564.59295

[2] "Akka: build concurrent, distributed, and resilient message-driven applications for java and scala," 2019, accessed: 2019-07-10. [Online]. Available: https://akka.io/