Distributed Mutual Exclusion: Token Passing on a Tree

Willi Menapace
203778
willi.menapace@studenti.unitn.it

Daniele Giuliani
203508
daniele.giuliani@studenti.unitn.it

1. Introduction

The task addressed by the project is the one of implementing distributed mutual exclusion on a tree structure under the assumptions indicated in the project description document. Our solution uses a modified version of Raymond's algorithm in order to grant FIFO access to the critical section with $\mathcal{O}(\log n)$ messages per request. We provide a range of testing functions that include both interactive sessions and preconfigured execution scenarios. For each of these two functioning modes we provide logging and support for arbitrary message delaying in order to stress uncommon message interleavings.

The next sections describe the structure of the Java program and the implemented messages and handlers.

2. Program Structure

The program is divided into multiple classes:

- `Node.java`, implements all of the protocol specifications.
- `NodeSystem.java`, creates the network by instantiating the different nodes.
- `CommandParser.java`, implements the console interface that can be used to test the system.
- `Configuration.java`, stores global configuration for the program.
- `Logger.java`, helps formatting the output of the console.
- `Tuple.java`, implements a simple data structure.

2.1. Node

The class `Node.java` implements the behavior of a node in order to comply with the protocol specifications. Let's examine the different attribute of this class and their use:

myId integer value that uniquely identifies the node.

holder ID of the node that we need to contact in order to ask for the permission. In case the node owns the Token then holder==myId.

neighbors map containing the IDs and ActorRef of the neighbor of the current node.

request_list queue of integers, represents the nodes from which we have received requests that we have to fulfill.

inside_cs boolean value that represents whether the node is inside the critical section or not.

recovery_info map of messages node IDs and recovery messages, used when the node crashes and must recover it's state.

broker instance of the class MessageBroker, explained in detail later.

logger instance of the class Logger, used to change log output in a controlled manner.

2.2. Message Broker

In order to transparently support message drops during crashes and message buffering during recovery, we introduce a code layer that manages incoming messages. This component receives all incoming messages and dispatches them to message handlers at the correct timeframes. Its behaviour is regulated by a mode which can take the following values

NORMAL_MODE dispatches every message to the handler as soon as it arrives

PREINIT_MODE dispatches only messages related to the initial configuration and buffers the other message types. When the mode will chance to NORMAL_MODE all buffered messages will be delivered in the original order and then normal operation resumes.

SELECTIVE_RECOVERY_MODE in the moment of activation allows only RecoveryInfoResponse and CrashEnd messages and all the others are dropped.

When a RecoveryInfoResponse is received on a channel it starts buffering messages on that channel instead of dropping them. When the mode will chance to NORMAL_MODE all buffered messages will be delivered in the original order and then normal operation resumes.

2.3. Configuration

The class `Configuration.java` stores different global configuration variables used by the other classes. The variable DEBUG is used to enable the debug mode, which enables a more complete logging mode of the behaviour of the nodes. The variable MAX_WAIT is used as an upper bound on the maximum amount of time that a node can wait before sending a message, which is chosen randomly by each node every time they try to send a message. The rest of them contain only special character used to color the output of the console for better visibility.

2.4. Command Parser

The class `CommandParser.java` implements the CLI that can be used to test the system. It accepts command both interactively and via text file, the main commands are:

`source filename` takes as input a filename and executes all commands contained in the file.

`create n` creates and adds *n* nodes to the Actor System.

`connect x y` sets node *x* as neighbor of *y* (and viceversa).

`inject x` injects the token inside network to node *x*. After the execution of this command is not possible to create or change the neighbor of the nodes.

`request x,y,z` takes as input a comma separated list of node IDs and asks each of them to request the privilege.

`crash x` takes as input a single node ID. It crashes/recovers the specified node. This command can be executed only if no other node is crashed!

`help` prints usage instruction of the console.

`exit` terminates the system.

`delay x` suspends the execution of the console for the specified amount of time (in millisecond). Used to delay the execution of commands when reading them from a file.

`rft x y` *request from to*, command use to forge a request message with sender *x* and recipient *y*. Can be used to simulate complex scenarios, if used without caution can easily crash the system.

`force_crash x` sends a CrashBegin message to the node specified regardless of how many other nodes are crashed.

`force_recovery x` sends a CrashEnd message to the node specified regardless if it's actually crashed or not.

Some of this commands are not intended for normal use, but only as a debug tool to simulate complex scenarios without having to rely on random interleaving of the execution.

3. Message Types

The following message types have been included in the protocol:

NeighborInit informs at configuration time the receiving node about a neighboring node

Init informs at configuration time the receiving node about his holder node

TokenInject injects the token in the network at configuration time

Request asks for access to the critical section. Can both be sent from the main thread or between nodes as part of the protocol.

Privilege grants the token to the receiving node. Contains a boolean field indicating whether the token must be returned to the sender after the use.

ExitCS self scheduled message that informs the node that is should exit the critical section

CrashBegin message sent from the main thread that causes the crash of a node

CrashEnd message sent from the main thread that causes a node to recovery from the crash

RecoveryInfoRequest requests a neighboring node for its internal state in order to carry out recovery procedures

RecoveryInfoResponse informs the receiving node about the internal state of the sender in order to perform recovery. Contains the id of the holder and a boolean field indicating whether there are pending requests.

4. Message Handlers

The following are the event handlers that handle the messages in 3 and that define the protocol logic. Only a brief description is given for each one. Refer to the comments in the code for a more detailed description.

onTokenInject starts an Init message flood to inform all nodes about their holder

onNeighborInit records the information of the neighbor into the neighbor nodes list

onInit registers the sender as the holder and continues the flood of the Init message

onRequest checks the state of the request list. If it is non empty it just enqueues the new request, otherwise it checks whether the privilege can immediately be granted or if the request must be forwarded to the holder. Care is taken to ensure no duplicate requests are generated. A filter in the beginning ensures no duplicte requests that may arise due to crash restarts are passed to the method.

onPrivilege forwards the privilege to the first node in the requester list or enters CS directly if the current node is the first. If the privilege must be sent back to the sender it adds the sender to the request list so that it will be served after all the current requests, ensuring FIFO.

onExitCS exits the CS and grants access to the first requester in the list

onCrashBegin erases the internal node state and starts dropping messages from neighboring nodes

onCrashEnd sends neighbors a RecoveryInfoRequest in order to obtain their state and reconstruct our state at the moment of the crash. The moment when a neighbor must see the crashed node become operational again on a link is the moment in which the crashed node receives the corresponding RecoveryInfoResponse. To ensure this, all messages on a link are continued to be dropped until RecoveryInfoResponse, and successive messages are buffered and processed again when all RecoveryInfoResponse are received and the internal state is resumed.

onRecoveryInfoRequest sends the requesting node information about the receiver's state

onRecoveryInfoResponse records the sender internal state information. At this point all further messages from the sender are buffered for processing after the crashed node state is resumed. When RecoveryInfoResponse has been received from all neighbors the holder and the request list is recalculated and message processing restarts normally starting from buffered messages. Note that it is not meaningful for the protocol to recover information whether or not the crashed node wanted to enter the critical section because that decision belongs to the new application logic of the restarted node, so that piece of information is not reconstructed.

5. Protocol safety

The protocol guarantees that there is always at most one node in the network holding the token. At protocol initialization the only node detaining the node is the one in which it is injected and later each time a Privilege message is sent the holder pointer is contextually changed to the destination node, so that never more than a node holds the token. Moreover, after a crash a Token is generated if and only if the crashed node detained it at the moment of the crash or was sent a Privilege message while crashed, so never than a token is present in the network. Permission to enter the critical section can be granted only to the holder of the token, so the protocol guarantees mutual exclusion.

6. Message complexity

Under the assumption of no crashes, in a tree with diameter d, a sequence of m critical section access requests can be satisfied with $\mathcal{O}(dm)$ messages. In the worst case in fact, the token travels through a sequence of requesting nodes distant d links from one another. For each of such exchanges, at most d request messages are forwarded in the path from the requester to the holder and d privilege messages are forwarded in the reverse path, giving a total of $2dm$ messages which gives an amortized $\mathcal{O}(d)$ number of messages per request. If the tree is built as a balanced k-ary tree, then $d <= 2\lceil\log_k((k-1)*n+1)\rceil$, where n is the number of nodes, so the amortized message number is $\mathcal{O}(\frac{\log n}{\log k})$.

If we also consider the possibility of crashes and the same balanced k-ary tree structure, the complexity in terms of Request and Privilege messages remains the same, but in the worst case the holder can crash at each step, generating an additional $2(k+1)$ RecoveryInfoRequest and RecoveryInfoResponse messages for recovery at each movement of the token, so the complexity becomes $\mathcal{O}(k\frac{\log n}{\log k})$.

7. Testing

In order to test the correct behavior of the protocol we created different execution scenarios (e.g. file *scenario_1.txt*) each of which stresses different aspects of the protocol such as concurrent requests, crashes and tree partitioning. These files contain a brief description of the tested behavior and a list of commands: the first part is used to dynamically create an appropriate topology for the current test case, while the second contains the commands that simulate a particular execution of the protocol. Moreover, in order to test uncommon message interleavings, we also add a configurable random delay to each message transmitted as part of the protocol. With the use of this scenarios and random message delays, we were able to observe that the network behaves correctly even under tricky situations. The execution of a scenario can be done done using the *source* command, e.g: `source scenario_3.txt`.