Distributed Systems 1 - Final Project

Alessandro Sartori [215062]
Massimiliano Fronza [216013]

1 Project Structure

1.1 Initialization

At program launch, the `main` method creates a list of clients (instances of `ClientActor)` and a list of replicas (instances of `ReplicaActor`). At this point an `InitializeGroup` message containing the list of replicas is created and sent to all actors. Upon reception of this message, each actor will copy the contests of the list into a local variable, from which clients will choose their target replica and replicas will infer the ring topology of the network. The last step concerns the scheduling of crashes, which are explained in detail in Implementation Choices.

1.2 Election

An election is triggered each time a replica needs to contact the coordinator but finds it unavailable, for instance when receiving the first `WriteRequest` from a client before any election has happened. The Java class `ElectionManager` encapsulates all the related logic: when the `beginElection()` method is invoked, the state is set to "electing" and the first `Election` message is forwarded into the ring. Two helper functions (`createElectionMessage()` and `expandElectionMessage()`) handle the creation and update of the election messages, while the remaining logic is mainly located within the receiver method `onElection()`. If, upon receipt of an `Election` message, a replica finds to be the winner of the election (via the `getWinner()` method), it broadcasts a `Synchronize` message containing all the updates that were provided as "most recent" by the other participating actors. When this message is received, the `coordinatorID` is updated and a callback to the replica is invoked, which will increment the epoch counter and, if any, apply missing updates.

1.2.1 Election Ack Timeouts

Each time an `Election` message is sent, a timer is started and put into a dedicated structure (see Implementation Choices). If the message is not acked by the recipient because of a crash, the timer triggers, the node is marked as unavailable, and the election restarts. If instead the `ElectionAck` is received in time, the timer is canceled and removed from the structure.

1.2.2 Election Timeout

The ring-based election logic rises a particular corner-case scenario, where a crash happening right after the Ack is sent to the previous node cannot be detected, causing the whole election to hang indefinitely. We handle this situation by starting a timer (via the usual data structure) each time an election starts and canceling it as soon as a new coordinator is found. The `TimerList` structure we developed ensures correct behavior even when concurrent elections are running.

1.3 Read Requests

Each client periodically generates Read Requests towards its target replica, which will immediately reply with its local value. Also, the client schedules a timeout to itself if the replica takes too long to answer, for example due to a crash. As for election timers, the structure provides correctness even if more then one request is being generated at a time.

1.4 Write Requests

As for Read Requests, clients periodically generate Write Requests to a replica chosen during their initialization. Upon receiving these requests, replicas check if their role is that of coordinator, and if it's not, the request is forwarded to the leader. When the coordinator receives the message, it generates an `UpdateID` (an epoch/seqNo tuple) and stores it into a specific structure used to track the number of ACKs received. At this point, an `Update` message is sent to all peers, to which they will reply with an `Ack`. For each Ack received, the coordinator increments the counter until it reaches a quorum `Q`. At this point the message is to be considered committed, and therefore the coordinator will propagate a `WriteOk` message. All correct replicas will eventually receive this message and set their value to the new one, after recording the update in their history.

1.5 Heartbeats

Whenever a new coordinator is elected, it schedules a `HeartbeatReminder` message to itself, which, when received, will trigger an iteration that sends a `Heartbeat` message to each replica still alive. Afterwards, the next `HeartbeatReminder` message is scheduled to happen after a predefined interval to restart the iteration.

1.5.1 Heartbeat timeouts

Each time a replica receives a `Heartbeat`, it starts a timer to wait for the next one and, if present, deletes the last active one. If the timer reaches its timeout, the coordinator is considered crashed and a new election is begun.

2 Implementation Choices

2.1 Updates

The class UpdateID contains an epoch value and a sequence number, and it is used to uniquely identify updates. The structure also provides helper functions to verify equality or happens-before relations between updates. `UpdateID`s are then used as keys inside the `Update` class, which associates to each ID the corresponding update value.

2.2 Update History

Update sequences are kept into a dedicated structure called `UpdateList`, which behaves like a list sorted by `UpdateID`. This class provides utility functions that replicate those of lists, such as `add()/remove()`, `size()/isEmpty()`, `contains()`, ..., but also novel methods like `getMostRecent()` or an overridden `toString()` to suitably represent histories in log messages. The ordering is kept by means of the happens-before methods exposed by `Update` and `UpdateID`.

2.3 Timeout management

Except for Heartbeats, which being recurrent are implemented via Akka's dispatcher, timeouts are implemented as Java `Timers` in two different classes: `TimerList` and `TimerMap`. These two classes provide access methods to create and add timers with a delay and a timeout-callback specified during instantiation, and to cancel them based either on their chronological order or a hashable key given to them, respectively.

`TimerList` exposes `addTimer()` and `cancelFirstTimer()`, and is used when timers are ensured to happen in a FIFO order and a key is not available for the identification of events, while `TimerMap` exposes `addTimer()` and `cancelTimer(T key)`, and is used when event order is arbitrary and timers need to be individually identifiable, for example when awaiting ACKs from specific senders.

2.4 Crashes

In order to allow precise control over the instant in which replicas crash, we conceived a `CrashHandler` singleton class which operates in two steps: the first step happens during initialization, where crashes are scheduled by providing an actor identifier and the corresponding `Situation` in which it is supposed to halt (for example `ON_ELECTION_ACK_RCV`). The second phase happens instead internally to each replica during execution, where before or after each important event (such as reception of a particular message), the node queries the `CrashHandler` to know if a crash is due for the occurred situation. If so, the behavior of the actor is changed to ignore every incoming message, and all its active timers are cancelled.