Distributed Systems

Murtas Cristian, 248025, cristian.murtas@unitn.it
Wang Marco, 249368, marco.wang@unitn.it

December 2024

1 Project Structure

1.1 Client (it.unitn.ds1.Client)

Clients represent external entities that interact with the system. Their role is to send requests to the system and to receive responses. Specifically, a client can send a *ReadRequest* to read the latest update serialized by the system. It can also send a *WriteRequest* to update the system with a new value.

1.2 Replica (it.unitn.ds1.Replicas)

Replicas are the core of the system. They are responsible for storing the history of updates and sending the latest one to the clients when requested. Replicas can operate under three different behaviors:

- *inElection*: replicas will switch to this mode when a new coordinator is needed. This behavior allows the group to process only the messages related to the election phase;
- *createReceive*: this is the default behavior, which allows a replica to process all the messages;
- *crashed*: this state is used to mimic a crashed replica, as it will ignore all incoming messages.

1.3 Simulation Controller (it.unitn.ds1.SimulationController)

The simulation controller is a wrapper that allows interaction with the system. It is responsible for the creation of clients and replicas. Thanks to its configurable implementation, it has been used to run the system in different scenarios.

2 Two-Phase Broadcast

The two-phase broadcast protocol is used to ensure that a message is either delivered to all correct replicas or none at all. The design handles partial failures by relying on acknowledgments and timeouts (See section 5). Moreover, the implementation of the protocol prevents messages from being lost due to crashes. If a message is not successfully written because of a crash, the next coordinator will handle it. All the scenarios can be grouped in two cases:

- if the coordinator crashes before sending the update message, the original replica will resend the message to the new coordinator;
- if the coordinator crashes after sending the update message, all replicas will have the message stored in their *temporaryBuffer*. In this case, the new coordinator will complete the serialization process.

Replicas leverage Three different data structures to manage data during the processing of write operations:

- *writeRequestMessageQueue*: contains the *WriteRequest* messages until an *UpdateMessage* for the specific request is received;
- *temporaryBuffer*: stores the *Update* until the replica receives a *WriteOK* message;
- *history*: stores the delivered *Update*.

3 Election Phase

The main role of the leader is to manage the serialization process of the write requests by gathering the consensus from the correct replicas.
The election phase is used to elect a new leader when there is none in the system (e.g., when the system starts) or when the current leader fails. Following a ring topology, every replica proposes itself as a candidate and sends its latest update to the next one.
The system will eventually reach a consensus on the new leader, which is going to be the most up-to-date replica. The flow charts describes the election algorithm when started in *createReceive* (fig. 1) and *inElection* (fig. 2) behaviors.
When the system is started for the first time, all replicas will enter the *inElection* behavior, sending an *ElectionMessage* to the next peer.

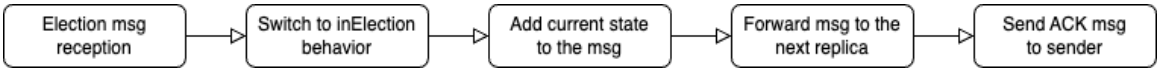

Figure 1: Election algorithm, while in *createReceive* state.

When a replica is in election mode, to prevent network overflow, only messages containing a potential winner are forwarded. If a replica receives a message that does not include a contender stronger than itself, it does not forward the message, knowing that a message containing itself as a potential winner may succeed.

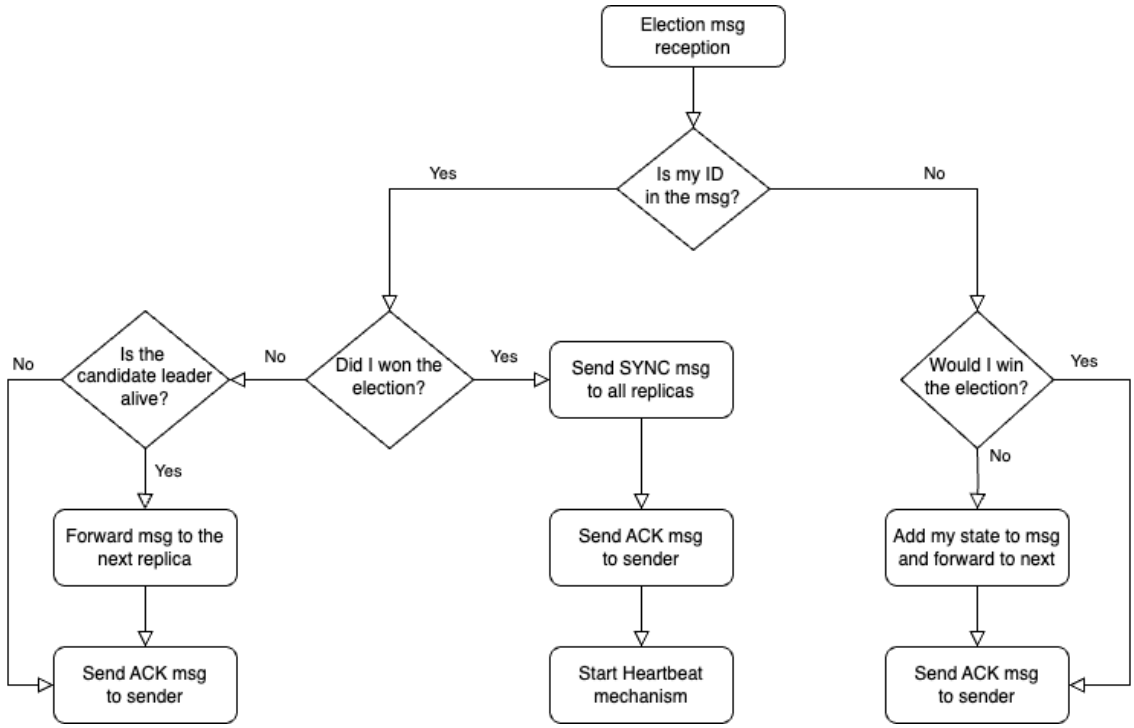

Figure 2: Election Algorithm, while in *ElectionState*.

3.1 Global Election Timeout

When a replica enters the *inElection* behavior it starts a timer related to the election. If the timer expires, the replica is going to restart the election process. This is done to avoid corner cases where the election may not terminate, such as:

- if two consecutive replicas fail in a way that the preceding correct replica is not able to detect;
- if the best candidate crashes.

3.2 Synchronization

Eventually a node will become the coordinator and it is going to broadcast a *SynchronizationMessage* to all replicas, providing them with potentially missing updates. Upon reception of this message, the replicas will: switch to the *createReceive* behavior; set the coordinator ref; start the heartbeat timer and, if needed, update their history and *temporaryBuffer*.
Furthermore, the coordinator is going to handle the pending updates before starting its own epoch. Finally, the system will process the message(s) received during the election before resuming its normal activity.

4 Crashes

The code has been instrumented to test specific crashes. These can be specified before creating the system by using an array of *Crash* objects.
The following list shows all the tested crash:

- New coordinator updates outdated replicas.
- Coordinator crashes after sending heartbeat message.
- Coordinator crashes after sending an *UpdateVariable* message.
- Coordinator crashes before sending an *UpdateVariable* message.
- Coordinator crashes while multicasting an *UpdateVariable* message.
- Multiple coordinator crashes.
- Replica crashes after receiving an *ElectionMessage*.
- Replica crashes before forwarding an *ElectionMessage*.
- Replicas crash before sending an *Ack* message related to an *UpdateVariable* message.
- Two consecutive nodes failure scenario during election.
- Replica crashes after forwarding a *WriteRequest*, and the coordinator crashes after multicasting the related update.

5 Timeout

During the execution, replicas set different timers to ensure that the system behaves as intended. If a timer expires, a specific subroutine will be executed to handle the fault.
The following list provides the implemented timers and their callbacks:

- *afterUpdateTimeout*: set when a replica receives an update from the coordinator. It starts a new election if a *WriteOK* message is not received;
- *afterForwardTimeout*: set when a replica (non-coordinator) forwards a *WriteRequest* to the coordinator. It starts a new election if an *UpdateVariable* message is not received;
- *electionTimeout*: related to the global election duration (See 3.1). It starts a new election if the current one does not end within an interval;
- *ackElectionTimeout*: set after forwarding an *ElectionMessage*. If the next replica does not reply with an ACK, it is marked as crashed and the message is forwarded to the new next;
- *heartbeatTimeout*: set on *SynchronizationMessage* by all replicas (except the coordinator) and it is used to monitor if the coordinator is still alive. This timer is re-set each time a *HeartbeatMessage* is received. Otherwise, a new election is started.
- *readRequestsTimers*: set by the client after sending a read request. If the replica does not reply, it means that it is crashed and it is going to be removed from the *replicas* list.