Course Project: Quorum-based Total Order Broadcast

Distributed Systems 1, 2019 – 2020

In this project, you are requested to implement a protocol for the coordination of a group of replicas, sharing the same data, that tolerates multiple node failures thanks to a quorum-based approach. The system handles update requests coming from external clients, and guarantees that all replicas eventually apply the updates in the same order. Updates are managed by a special replica, the *coordinator*, following a two-phase broadcast procedure. If the coordinator crashes, a new one is elected to ensure continuity of service.

1 General description

The project should be implemented in Akka with each of the N replicas being an Akka actor, identified by a unique ID. For simplicity, we assume that replicas hold a single variable v. External client actors can contact any replica to read its value and request a write operation to update it. Sequential consistency is guaranteed if the client always interacts with the same replica during its operations. We assume no new replica will join the system, however replicas may crash at any time.

Client requests. The client can issue read and write requests to any replica. Both types of request contain the ActorRef of the client. The write request also contains the new proposed value $v*$. For read operations, the replica will reply immediately with its local value. For write operations, instead, the request will be forwarded to the coordinator.

Update protocol. To perform the update request, the coordinator initiates the protocol by broadcasting to all replicas the UPDATE message with the new value. Then, it waits for their ACKs until a quorum Q of replicas is reached. In this implementation, $|Q| = \lfloor N/2 \rfloor + 1$, i.e., a majority of nodes. Once enough ACKs are collected, the coordinator broadcasts the WRITEOK message.

Each UPDATE is identified by a pair $\langle e, i \rangle$, where e is the current epoch and i is a sequence number. The epoch is monotonically increasing; a change of epoch corresponds to a change of coordinator. The sequence number resets to 0 for every new epoch, but identifies uniquely each UPDATE in an epoch. Upon a WRITEOK, replicas apply the update and change their local variable. They also keep the history of updates to ensure none of them is lost in the case of a coordinator crash (see "Coordinator election").

Crash detection. The coordinator and the other replicas may crash at any time. However, we assume that a quorum of $|Q|$ replicas is always available. The system should implement a simple crash detection algorithm based on timeouts. Replicas detect a crashed coordinator if they timeout while waiting for the WRITEOK message. In addition, a replica forwarding a write request should timeout if the coordinator does not initiate the broadcast. Finally, all replicas expect to hear from the coordinator from time to time; the coordinator periodically broadcasts a heartbeat message.

Coordinator election. If the coordinator crashes, the remaining replicas must elect a new coordinator to take its place. Coordinator election must be performed by a simple ring-based protocol, where the ring topology is established based on replica IDs. ELECTION messages forwarded in the ring should contain the information about the known updates for each node. The new coordinator should be a replica that knows the most recent update. IDs are used to disambiguate replicas that are equally up-to-date. Eventually, the best candidate broadcasts a SYNCHRONIZATION message to announce the new leadership and synchronize all other replicas by sending all the updates that they missed. Then, the broadcast protocol can continue.

Properties. The system must respect the following property: if a replica a applies an update w, then all correct (not faulty) replicas will apply w (regardless of whether a is correct). If the property does not hold, a client may read a value that is found nowhere else in the system. When a new coordinator is chosen, it should take care of completing any interrupted broadcast so that the above property is respected.

2 Implementation-related assumptions and requirements

- We assume the unicast message exchanges between actors are reliable.
- To emulate network propagation delays, you are requested to insert small random intervals between the unicast transmissions, as done in the examples seen in class.
- Ensure proper actor encapsulation. Avoid using shared objects unless they are immutable.
- The program should generate a log file (or multiple log files) recording the key steps of the protocol. In addition to arbitrary logging, the program must record the following log messages:
  - `Replica <ReplicaID> update <e>:<i> <value>`

  - `Client <ClientID> read req to <ReplicaID>`
  - `Client <ClientID> read done <value>`

  To automate testing, you can request a series of write operations (that may be affected by crash events), while a client reads from a replica; the read values should respect sequential consistency. The correct ordering can also be checked by inspecting the log of each replica.
- To emulate crashes, a participant should be able to enter the "crashed mode" in which it ignores all incoming messages and stops sending anything.
- We assume crash detection does not give false positives. Use reasonable values for timeouts.
- During the evaluation it should be easy, with a simple instrumentation of the code, to emulate a crash at key points of the protocol, e.g., during the sending of an update, after receiving an update, during the sending of WRITEOKs, during the election, etc.

3 Grading

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness). *Do not* spend time implementing features other than the ones requested — focus on doing the core functionality, and doing it well.

A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit programs implementing a subset of the requested features or systems requiring stronger assumptions. In these cases, lower marks will be awarded.

You are expected to implement this project with exactly one other student, however the marks will be individual, based on your understanding of the program and the concepts used.

4 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (davide.vecchia@unitn.it), well in advance, i.e., at least a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enrol in the next exam session.
- The code must be properly formatted otherwise it will not be accepted (e.g., follow style guidelines).
- Provide a short document (1-3 pages) in English explaining the main architectural choices.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation in a directory called `RossiRusso`, compress it with zip or tar ("`tar -czvf RossiRusso.tgz RossiRusso`") and submit the resulting archive.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.