Quorum-based Total Order Broadcast

Damiano Salvaterra
Nawal Deffou
Department of Information Engineering and Computer science, University of Trento

Abstract—**This report details the implementation of a quorum-based total order broadcast system using Akka actors in Java. The system coordinates a group of replicas that share the same data and guarantees that all replicas eventually apply updates in the same order, ensuring sequential consistency for client operations.**

I. System Architecture

The system consists of the following main components:

- **Replicas**: Akka actors that hold and manage a single variable.
- **Clients**: Akka actors that issue read and write requests to replicas.
- **Coordinator**: A special replica responsible for managing write operations using a two-phase broadcast protocol.

II. Project Structure

The project consists in several source files. The message classes are packaged inside the Messages package, which contains classes of messages divided per type: **ClientMessages.java**, **CoordinatorMessages.java** and **ReplicaMessages.java**. The package contains 2 additional classes, **Data.java** and **Timestamp.java**: the first encapsulates the update and its state (stable or unstable), while the second is the pair *(epoch, sequenceNumber)*. These two classes were included in the Messages package since the messages exchanged in the system are strongly coupled with these data structures. The package is used by the main classes that implement the protocol, which are:

- **Replica.java** is the class that contains most of the logic of the system. This class manages both the behavior of a normal replica and the behavior of the coordinator: the difference between the two is in the Behavior of the Akka Actor.
- **Client.java** implements the client, which only purpose is sending read and write requests to the replica. The requests are triggered by two specific messages sent from the Main to the clients.
- **Main.java** is the class that is used for testing the protocol.
- **TimeoutType.java** is an enumeration that defines the timeouts and their timing.
- **Utils.java** is a class of utilities methods.

III. Design choices

In the following we report the main design choices, regarding on how to manage the updates and the algorithm for the election protocol.

A. Updates Storage

The replicas store the updates in an ordered Map, called *localHistory*, which has as keys a Timestamp object and as values a Data object. The Map is implemented as a LinkedHashMap, which maintains the insertion order of the keys, feature that becomes useful when the new coordinator has to update the replicas with the missing updates. Moreover, the replica holds to pointers to the HashMap, *lastUpdate* and *lastStable*, which point respectively to the last inserted update and to the last stable update. The Data class has a boolean field *stable* that holds the information necessary to distinguish if an update has received the WRITEOK message or if it is still in the UPDATE case (stability): when a replica receives the UPDATE message, it stores the update in the Map with the *stable* set to false. When the replica receives the WRITEOK message for the same update, it sets the flag to true. This way we can store the updates in the same data structure regardless of their state.

B. Timeouts

Each replica has to manage several timeouts, in particular we have:

- **UPDATE**: This is the timeout that a replica sets when it forwards a write request to the coordinator: after forwarding the request, the replica expects to receive an update message. If the UPDATE does not arrive, the replica starts an election;
- **WRITEOK**: This timeout is set by a replica when it receives an UPDATE message. After receiving the UPADTE, the replica expects that after some time the coordinator will send the WRITEOK and if it does not receive it the replica will start an election;
- **ELECTION_PROTOCOL**: When a replica starts an election, it sets this timeout that it be will cancelled once the protocol is converged to a coordinator: the replica expects that the protocol will terminate after a while, otherwise this means that the token is circulating in the ring forever (because the best candidate is crashed during the election) or that two consecutive replicas have crashed in an undetectable way. If the timeout expires, the replica will start a new election.
- **ELECTION_ACK**: This is the timeout for when a replica, during the election, forwards the token to the successor: if it does not receive the acknowledgement from the successor, it will forward the token to the next successor.

- **RECEIVE_HEARTBEAT** This timeout is (re)set when a replica receives the HEARTBEAT message from the coordinator or if it receives a SYNCHRONIZATION message from the new coordinator.
- **SEND_HEARTBEAT**: This is the periodic timeout set by the coordinator for sending the HEARTBEAT message.

The timeouts are defined as an enumeration that brings the timeout value as parameter for each enumeration value. When a timeout expires, a Timeout message is sent from the replica to itself. This timeout message contains the enumeration type of the timeout and based on type the replica will perform the operations concerning each timeout. This is true for all timeout types except for the SEND_HEARTBEAT timeout, which is scheduled once when the replica becomes coordinator and is used to set a scheduler that periodically ping the other replicas.

For keeping the state of the timeouts and the Cancellable jobs we use an HashMap that maps the timeout type to a queue of Cancellables. When a timeout is scheduled, the relative Cancellable is enqueued in the queue returned by the HashMap; when a timeout is cancelled, we poll the queue and remove the timeout in FIFO order. This is necessary especially for the WRITEOK and ELECTION_ACK timeouts, since one replica may wait for multiple responses of these type concurrently[1].

C. Election protocol

Although the election protocol is in principle simple, designing the algorithm that correctly implement was not straightforward. The main issue is, in fact, managing the case when multiple tokens circulates in the ring concurrently. This may happen in two cases: when different replicas detects the same coordinator crash and starts two elections concurrently, or when the election protocol does not terminate (i.e the best candidate crashes during the election or two adjacent nodes in the ring crashes in an "unlucky" way). Since the protocol requires that a replica sees the token two (and only two) times, having multiple tokens circulating in the system breaks the protocol, because with a straightforward implementation the nodes have no way to distinguish one token from another. The algorithm that we propose covers the two sources of multiple tokens just mentioned and solves the two problems separately:

- **Managing concurrent instances of the protocol:** as said earlier, is not hard to realize that when the coordinator crashes, it's likely that several replicas will detect the crash at the same time, each of them initiating an election concurrently. In order to avoid the interleaving of the data brought by the token that will lead to breaking the protocol, we need a sitematic way to discard the less "valuable" token. The solution we thought is the following:
  - Each replica maintains to variables: *candidateID* and *mostRecentUpdate*. The first is the current best candidate that the replica knows and the second is the currently known most recent update **in the system** that the replica knows.
  - When a replica initiates an election it sets *candidateID* to itself and *mostRecentUpdate* to *lastStable*. These two informations are put in the token and forwarded. When a replica receives a token for the first time, it checks if its *lastUpdate* is more recent then the most recent update known in the token. If it is, it updates the token with its most recent update, set itself as candidate (in its local attribute and in the token) and forwards[2]. If it is not, the replica sets its *candidateID* and its *mostRecentUpdate* as the ones in the token and forwards.
  - When a replica receives the token for the second time, if the token brings a more recent update, then the replica updates itself again like the previous case. If the token brings a less recent update then the local one instead, we can safely discard the token: since is the second time that the replica see a token, there is for sure another circulating token that is at least as updated as this replica, so the current token is useless. This way, the system and the tokens converge to the same values and the redundant tokens are discarded. The third case possible in the receipt of a token that is not the first is when the token brings the current replica as candidate: in this case the current replica becomes coordinator.
- **Discard a non-terminating token:** a non-terminating token is a token that holds as best candidate a node that is crashed during the election, i.e. when the token passes through the best candidate C the first time, but when it has to pass the second time the candidate is crashed so the predecessor of C forwards the token to the successor of C. At a certain point some replica (likely the initiator of the protocol) will timeout for ELECTION_PROTOCOL and will restart a new election. With the algorithm for discarding redundant tokens just described, the token with the crashed candidate will always preempt the other tokens (and also the new one) making the system converge to a crashed candidate and making the token be forwarded forever. To avoid this, another condition for discarding the token is added: the replica holds a counter, called *electionInstanceCounter*, which is reset to 0 at every epoch, which is incremented and put in the token each time the replica initiates a new election (within an epoch). Before processing the token as descirbed before, a replica, when receives a token, it compares the token counter with the local one:
  - If the token counter is lower than the local one, then discard the token: it belongs to an old election[3].
  - If the token counter is higher than the local one,

[1]Actually, also for the UPDATE timeouts we could have multiple timeouts enqueued, if a replica forwards multiple write requests in short time, although we did not directly tested this.

[2]If the updates are the same, the replica with higher ID is selected as candidate.

[3]*old* means an election that was started before the best candidate crashed.

then a new election is initiated: the replica resets its information about the candidate and the last update in the system (returns to the init state of the election) and restarts the protocol.
- If the token counter is the same as the local one, then process the token with the algorithm described above

D. Synchronization

When a replica becomes coordinator, it broadcasts a SYNCHRONIZATION message to inform the other replicas. Then, it parses the information in the token in order to know the state of the updates in the system. It repeats the broadcast of the WRITEOK message for the stable updates that have not been set as stable in all the replicas and re-ask for the quorum for all unstable updates. In this way, even if the new coordinator crashes during the synchronization of the other replicas, there is an invariant that is maintained. This invariant is that, due to the channel assumptions, there can be an arbitrary number of unstable messages **that everybody know** waiting for the quorum, plus **at most one unstable** update that is known only to some replicas **OR at most one stable** update that is known only to some replicas. We could have choose to simply broadcast all missing updates as stable after the synchronization, but in case of coordinator crash during this broadcast it would be less immediate to track down the state of the system. With this implementation instead, we loose some performance after the election (since we have to repeat the two phase broadcast for the unstable messafges befor changing the epoch) but the behavior and the state of the system are the same in any moment as the coordinator never crashed.

E. Write requests and quorum

When a replica receives a write request from the client, it forward the request to the coordinator. When the coordinator receives a write request, it assigns a Timestamp, it puts in the quorum and broadcasts an UPDATE message with the value received (as unstable) and the timestamp assigned. When a replica receives the UPDATE it puts the unstable update in its local history, sends the UPDATEACK and set the timout for the WRITEOK. When the coordinator receives an UPDATEACK for a given update, it increments the quorum count for that update and if the quorum is reached, it sets the update as stable and broadcasts the WRITEOK. The quorum count is kept in a HashMap with the Timestamp as key and the quorum count as value. This way the coordinator can easily gather different quorums for different updates simultaneously. When a replica receives the WRITEOK, the relative update is already in the local history (as unstable), so it just sets the *stable* flag as true.

IV. Debugging and testing

During the debug, several debug prints was inserted. The debug prints can be seen on standard output and they keep track of every single step done during the broadcast protocol and the election protocol[4]. For the testing, we set some predefined cases that can be executed by executing the main.

A. Crash simulation

For simulating the crashes at specific times of the protocol, we created a **CrashMode.java** class that contains an enumeration of the relevant crashes and a parameter, used to control when exactly the crash happens during a broadcast[5]. One instance of this class is sent to the replicas through a message (**DebugMessages.CrashMsg**). The replicas keep one instance *crashEvent* of the CrashMode class (set by default to NO_CRASH) and when they receive a CrashMsg they set *crashEvent* to the instance in the message. This object defines the behavior of the replica in specific situations: for example, when the crash event called *DURING_UPDATE_BROADCAST* with parameter 4 is set, the coordinator will crash just before sending the fourth UPDATE message of the broadcast (i.e. only 3 replicas will receive the UPDATE). To use this mechanism, is important to send the message containing the CrashMode before starting the sequence of events that leads to the wanted crash. Furthermore, the CrashMsg has to be broadcasted to all the replicas in the system for two reasons: first, if the crash message is referred to the coordinator, we need to broadcast it because we do not know a priori who the coordinator is; second, there are some crash modes that are not coordinator-specific, for example the crashes happening to the candidate coordinator during the election: in this case, we do not know a priori who the best candidate will be, so we have to set the crash mode to all the replicas and "get them ready to crash"[6]. The crash messages can be broadcasted to all the replicas in any case, the replicas contain the logic for filtering the crash events that are not relevant to their state.

[4]Due to multithreading, it may happen that the prompts and the debug prints are interleaved, just press Enter if the prompt is not visible.

[5]Crashes not related to broadcasts do not use this parameter, so it is set to -1.

[6]For the crash events related to the election, is important, at the end of the election protocol, to reset the crash mode of all replicas to NO_CRASH, otherwise their crash state will remain to the crash mode related to the election.