Introduction to Distributed Systems
WT 17/18

Assignment 6 – Part II (programming)

Submission Deadline: Monday, 29.01.2018, 23:59

- *Submit the solution via ILIAS (only one solution per group).*
- *Respect the submission guidelines (see ILIAS).*

5 Data Replication – Weighted Voting [20 points]

Weighted Voting is one of several data replication methods presented in the lecture to increase the availability of data. In this exercise, you shall implement the Weighted Voting protocol as given in the lecture.

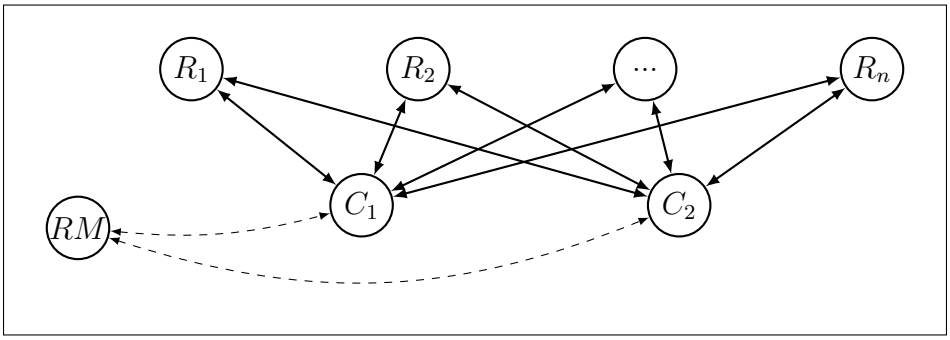

Figure 1: System consisting of multiple replicas $R_1, R_2, \ldots$, two clients C_1 and C_2 and the replication manager RM.

From the course materials website (Ilias), you can download a code stub (`a6q5.zip`) that contains an example of a data replication system as shown in Fig. 1. Therein, the following message classes are provided for implementing the communication between clients and replicas:

`ReadLockRequest` – Requests a read vote/lock for the client on a replica.

`WriteLockRequest` – Requests a write vote/lock for the client on a replica.

`UnlockRequest` – Releases a previously acquired read or write lock for the client on a replica.

`ReadRequest` – Requests the current value from a replica.

`WriteRequest` – Requests that a replica updates its value and version to those contained in the message, given that it is a valid update (i.e., given version number > current version number).

`VoteResponse` – Contains the vote (`YES`/`NO`) and version number of a replica. In addition, this message is used as a general acknowledgment reply to `UnlockRequest` and `WriteRequest` messages. The version number is only valid for `YES` votes sent in response to a `ReadLockRequest`/`WriteLockRequest` message. For `YES` votes, these objects also carry the number of votes held by the corresponding replica.

`ValueResponse` – Sent in response to a `ReadRequest` message. Contains the value stored on this replica.

To implement a communication protocol using these classes, you can use the methods for (de)serializing objects from/to byte arrays provided by the `Util` class to transfer message objects using sockets.

Notes:

- *For the sake of simplicity, clients do not execute concurrently.*
- *UDP (`DatagramSocket`) is used for all communication between clients and replicas. For the sake of simplicity, the methods of the replication manager can be accessed directly by clients.*
- *All requests to a replica require a response. For a `ReadRequest` message, a positive response is a `ValueResponse` message. All other messages are answered by a `VoteResponse` message.*
- *In addition to what is shown on the slides, you have to explicitly release any locks after the corresponding operation has been completed or when a quorum could not be reached.*

Complete the provided code by following these steps:

a) [2 points] The `WeightedVotingReplicationManager` class represents a simple replication manager that maintains a list of replicas, determines appropriate read and write thresholds, and provides `checkRead/WriteQuorum()` methods for clients to check if the corresponding operation can be performed.

Complete the constructor of the `WeightedVotingReplicationManager` class by correctly initializing the `read/writeThreshold` fields!

Also, implement the `checkQuorum()` method!

b) [5 points] The `Replica` class represents the replica nodes, storing copies of a replicated variable.

Replicas lock their copy when they receive a `ReadLockRequest` or `WriteLockRequest` and send a `YES` vote if they are not already locked. Otherwise they respond with a `NO` vote. All other requests are handled according to the current lock status. In particular, requests that require a lock may only be processed if they were sent by a client that holds the corresponding lock.

In accordance with the system model used in the lecture, the `Replica` class shall simulate nodes that may be unavailable (i.e., do not answer to a request). To this end, each `Replica` is configured with an `availability` field, where an availability of 0.7 means that on average 30 % of all vote/lock requests (`ReadLockRequest` and `WriteLockRequest`) are ignored at random.

*Note: To simplify the overall implementation, replicas may **only** fail as long as they are **not locked**. If implemented correctly, you can therefore assume that a replica which, for instance, responded to a `WriteLockRequest` with a `YES` vote will reliably handle a subsequent `WriteRequest`.*

Complete the implementation of the `Replica` class by implementing its `run()` method!

c) [3 points] In previous exercises you have usually used blocking I/O operations. When communicating with replicas that are unavailable, blocking I/O will lead to a deadlock. In this exercise, a non-blocking receive operation shall be implemented by completing the `NonBlockingReceiver` class.

Implement the `receiveMessages()` method to receive (up to) an expected number of messages within a given timeout period! It shall block only until either the timeout has passed or the expected number of messages has been received and return all messages that were received during this time.

Hint: You can use `DatagramSocket.setSoTimeout()` to set a timeout for blocking operations. It may be assumed that if any replica does not reply within 1000 ms, then it has crashed and is not going to respond.

d) [5 points] Finally, the `Client` class is used to read and write replicated values from replicas with the help of the replication manager.

Implement the `get()` method of the client to read replicated values according to the Weighted Voting protocol!

You may use the provided `requestReadLock()`, `readReplica()` and `releaseLock()` methods.

Note: You must not read a value from any replica that did not reply with a `YES` vote!

e) [5 points] Implement the `requestWriteLock()`, `writeReplicas()` and `set()` methods of the client to write replicated values according to the Weighted Voting protocol!

Note: You must not write a value on any replica that did not reply with a `YES` vote!