Introduction to Distributed Systems
WT 19/20

Assignment 5 – Part II (Programming)

Submission Deadline: Monday, 13.01.2020, 08:00

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

5 Majority Consensus [15 points]

Majority Consensus is one of several data replication methods presented in the lecture to increase the availability of data. In this exercise, you will implement the Majority Consensus algorithm as given in the lecture.

From the course materials website (Ilias), you can download a code stub (`a5q5.zip`) that contains an example system as shown in Fig. 1. Furthermore, the following set of classes is provided for implementing the communication:

`RequestReadVote` – Requests a read vote for the client on a replica.

`RequestWriteVote` – Requests a write vote for the client on a replica.

`ReleaseReadLock` – Releases a read lock for the client on a replica.

`ReleaseWriteLock` – Releases a write lock for the client on a replica.

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

`WriteRequestMessage` – 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).

`ValueResponseMessage` – Sent in response to a ReadRequestMessage. Contains the value stored on this replica.

`Vote` – Contains the Vote ("YES"/"NO") and version number of a replica. In addition, this message is used as a general ACK/NACK message for ReleaseReadLock, ReleaseWriteLock and WriteRequestMessage. The version number is only valid for "YES" votes sent in response to a RequestReadVote/RequestWriteVote message.

To implement this communication protocol, you can use the built-in serialization of objects to byte arrays (`ObjectInputStream`/`ObjectOutputStream`) to transfer objects over the network.

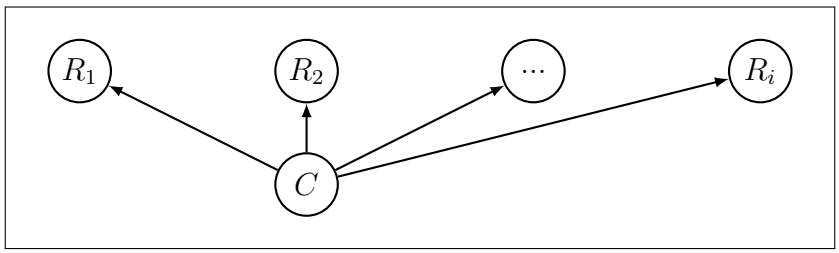

Figure 1: Example system of one client C and multiple replicas R_i

Notes:

- Although this exercise simplifies the problem by introducing only a single client, your replicas should be designed to support access by multiple clients. However, you can assume all locks to be exclusive, i.e., two clients might request a read lock for a replica, but only one of these client will obtain the lock and interact with the replica. On the client-side, you do not need to consider concurrent requests from other clients.
- You can download the stub classes for all processes on the course materials website. The process structure is set up by the `Main` class.
- Use UDP (`DatagramSocket`) for all communication.
- All requests to a replica require a response. For a ReadRequestMessage, a positive response is a ValueResponseMessage. All other messages are responded to by a Vote message.
- In addition to what is shown on the slides, you have to explicitly release locks set by vote requests after an operation or when the quorum could not be reached.

Complete the provided code by following these steps:

a) [3 points] The `Replica` class represents the replica nodes, storing copies of y. To allow for testing of the majority consensus protocol, the `Replica` class needs to simulate nodes that may be unavailable, i.e., do not answer to a request. Each `Replica` is configured with an `availability` value, where an availability of 0.7 means that for 30 % of all requests, no response is sent. *Note: To simplify the overall implementation, Replicas can only fail as long as they are not locked.*

Complete the implementation of the `Replica` class. It must store a copy of y with the availability properties described above. Furthermore, implement the replica-side of the network protocol (message reception and processing) in the `Replica` class.

b) [2 points] In previous exercises you have always used blocking I/O operations. When communicating with replicas that are unavailable, blocking I/O will deadlock your program. Therefore, a non-blocking solution is needed. `NonBlockingReceiver.receiveMessages()` takes a timeout value and an expected number of messages. It will wait 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.

Implement the class `NonBlockingReceiver` so that you can receive replies from all available nodes within a given timeout.

Note: The given timeout is a total value for receiving all replies.

Hint: You can use `Socket.setSoTimeout()` to set a timeout for blocking operations. Be careful when determining the timeout values. For example, implementations which simply divide the given timeout by the number of expected packets will not receive points!

c) [5 points] Implement reading a replicated value in the `MajorityConsensus` class. To do this, you will need to implement the methods `requestReadVote()`, `readReplica()`, `releaseReadLock()` and `checkQuorum()`. Using these methods, implement the `MajorityConsensus.get()` method.

Note: You must not read a value from replicas that did not reply with a "YES" vote!

d) [5 points] Implement updating a replicated value in the `MajorityConsensus` class. To do this, you will need to implement the methods `requestWriteVote()`, `writeReplicas()` and `releaseWriteLock()`. Based on these methods, implement the `MajorityConsensus.set()` method.

Note: You must not write a value on replicas that did not reply with a "YES" vote!