POLITECNICO
MILANO 1863

Distributed Systems Project

Quorum-based Replicated Datastore

Antonino Sposito
Matteo Lukasch

Assignment

We have to implement a distributed key-value store that accepts two operations from client:

- put(k, v), which inserts/updates the value v for key k;
- get(k), which returns the value associated with key k.

The store is internally replicated across N nodes (processes) and offers sequential consistency using a quorum-based protocol, with read and write quora as configuration parameters.

We decided to simulate the network by means of OMNeT++, making use of the knowledge gained from the course and our own understanding of leaderless protocols to implement the requested quorum-based distributed datastore.

Assumption

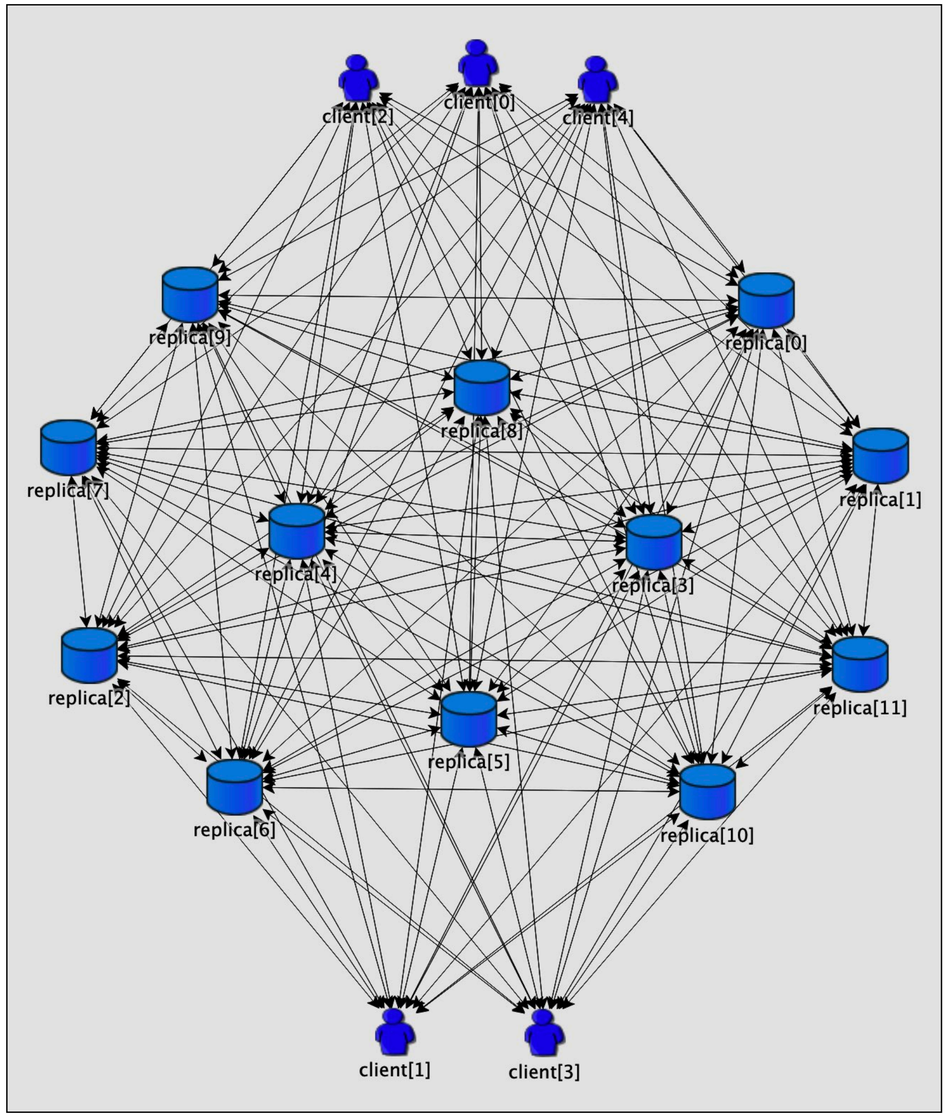
- Each key and value is a positive integer numbers;
- The network topology is structured as a full-mesh between replicas and each client is connected to every replica (clients cannot communicate with each other);
- Each client, upon sending a request, will contact a random replica which will serve as a proxy, contacting a number of random nodes based on the quorum parameters;
- Replicas are initially empty, with key-value pairs added upon put requests performed by client and for which agreement has been reached inside the replicated datastore;
- Put requests are blocking: in the context of two puts for the same key, the last arrived is aborted;
- Get requests are non-blocking: every get is always served with the most recent version of the value for a specific key, if present;
- Processes and links are reliable.

Network elements

The network is composed of two elements:

- Replicas,
- Clients.

The former are the ones composing the replicated datastore, representing processes in charge of managing requests coming from clients. Whereas the latter will be the ones to perform put and get requests to the datastore.

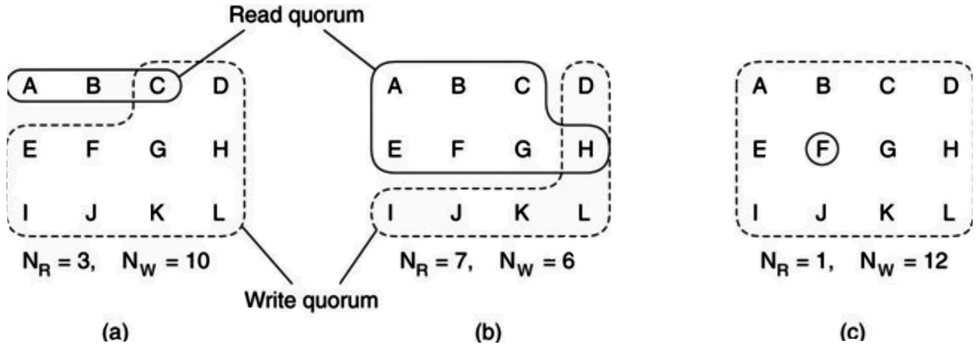


Sequential consistency and Leaderless protocols

“The result is the same as if the operations by all processes were executed in some sequential order, and the operations by each process appear in this sequence in the order specified by its program”

In sequential consistency, operations within a process may not be reordered, and it guarantees that all processes see the same interleaving, while not relying on time.

The leaderless protocol might make use of a quorum-based approach, where clients contact multiple replicas to perform a read or a write operation, and an update occurs only if a quorum of the servers agrees on the new value to be written (assigning also a new version for that value), whereas the read operation requires a quorum to ensure the latest version for the resource is being read.



In this example, straight lines are used for the read quorum, whereas the write quorum is denoted by dotted lines. In (a) we can notice a correct choice for read and write quora, (b) may lead to write-write conflicts and (c) is known as ROWA (read one, write all).

Network parameters

- N, the number of replicas (processes) inside the replicated datastore;
- M, the number of clients making put and get requests.

Quorum parameters

- $N_w > \frac{N}{2}$, number of write operations to perform in order to guarantee agreement on write;
- $N_r > N - N_w$, number of read operations to perform in order to guarantee agreement on read.

Protocol parameters

- Keyspace, number of keys available in in the system;
- Put (Get) time interval, time interval between subsequent put (get) requests.

Working principles of the client

- The client can either send put or get requests:
  - The former is a request to store a new value v for a specific key k;
  - The latter is a request to obtain the latest version of the value v associated to the key k.
- In the case of a put request:
  - The client randomly chooses a replica to submit the request and wait for a response;
  - If the client receives back a commit-message, replicas agreed on the new value proposed by the client;
  - Otherwise, an abort-message is received by the client, meaning that a subset of the replicas, to which the put request has been sent to, was already working on that key.
- In the case of a get request:
  - The client again chooses a replica randomly and submit the request, then wait for a response;
  - If the client receives back a get-response message, a value was present for the given key and the latest version for it is delivered to the client;
  - Otherwise, an empty-get-response message is received by the client, meaning that no values are present for the given key.

Working principles of the replica - 1

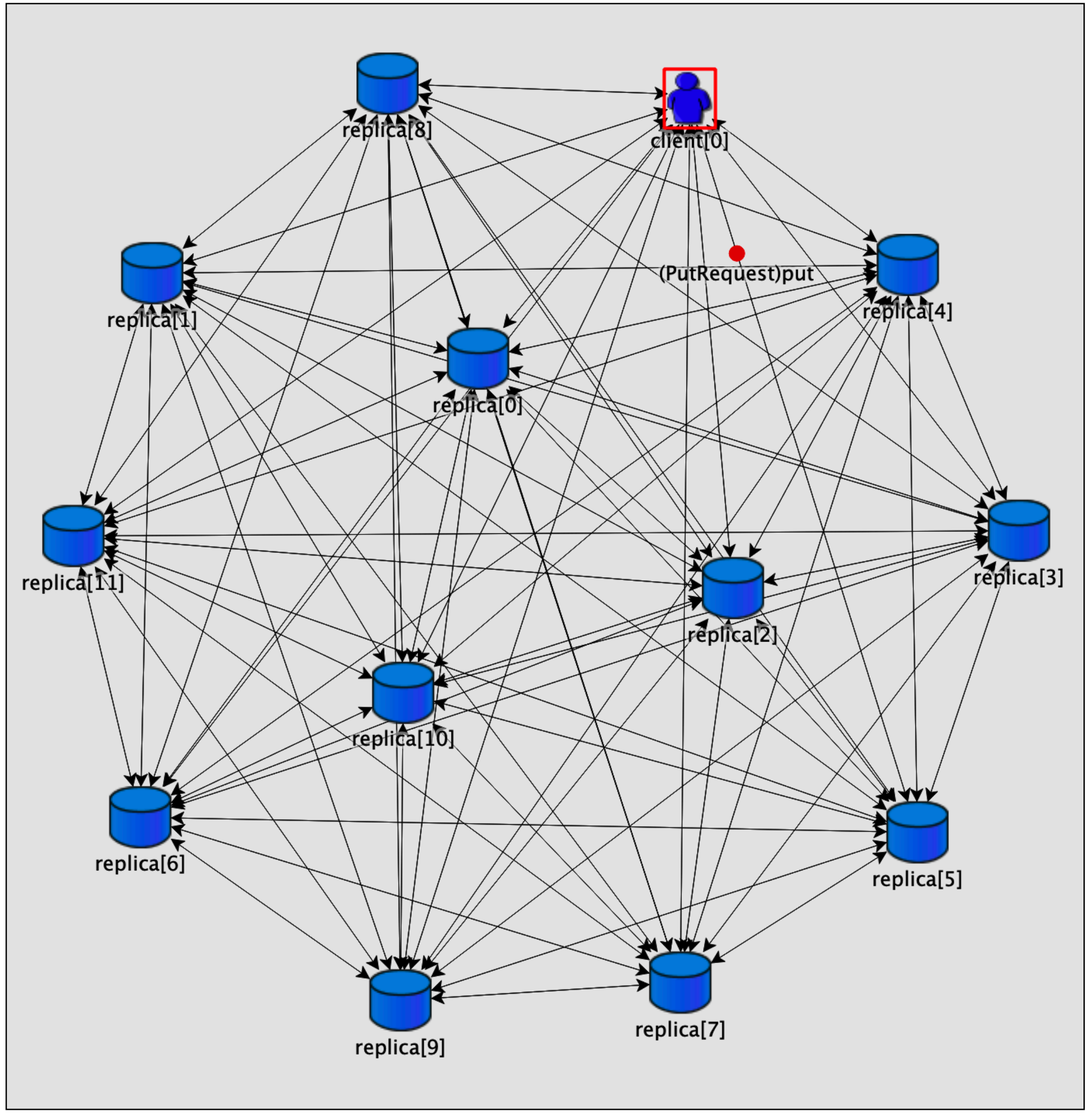
- The replica can either receive put and get requests, or inter-communication messages:
  - The former are requests coming from clients to perform put and get operation;
  - The latter are used by replicas, for example to reach an agreement or to signal an abort.
- In the case of a put request:
  - The chosen replica serves as a proxy, contacts the number of replicas needed to reach the write quorum and waits for all of them to send back their version of the value for the given key, if present;
  - Once all the replicas replied, if no abort message is received by the proxy:
    - The proxy decides for the new version and sends a commit-message with the new value and the next version for that key;
    - Otherwise, an abort-message is forwarded to all replicas and then to the client;
  - In particular, if the key is not present a commit-message is sent to all the other replicas of the quorum and to the client with the new key-value pair.

Working principles of the replica - 2

- In the case of a get request:
  - The chosen replica serves again as a proxy, contacts the number of replicas needed to reach the read quorum and waits for all of the them to send back their version of the value for the given key, if present;
  - Once all the replicas replied:
    - The proxy selects the latest version and sends a get-response with the associated value for the specific key and version;
    - Otherwise, an empty-get-response is forwarded to the client if the key is not present in any replica of the quorum.
- The read and write quorums guarantee that put requests reach the majority of replicas in case of success, and the get ones to return the latest version of a value associated to a key.
- In the protocol implementation we took inspiration from 2PC for the agreement procedure, with the replica selected as proxy somehow acting as a coordinator for the other replicas in the quorum.

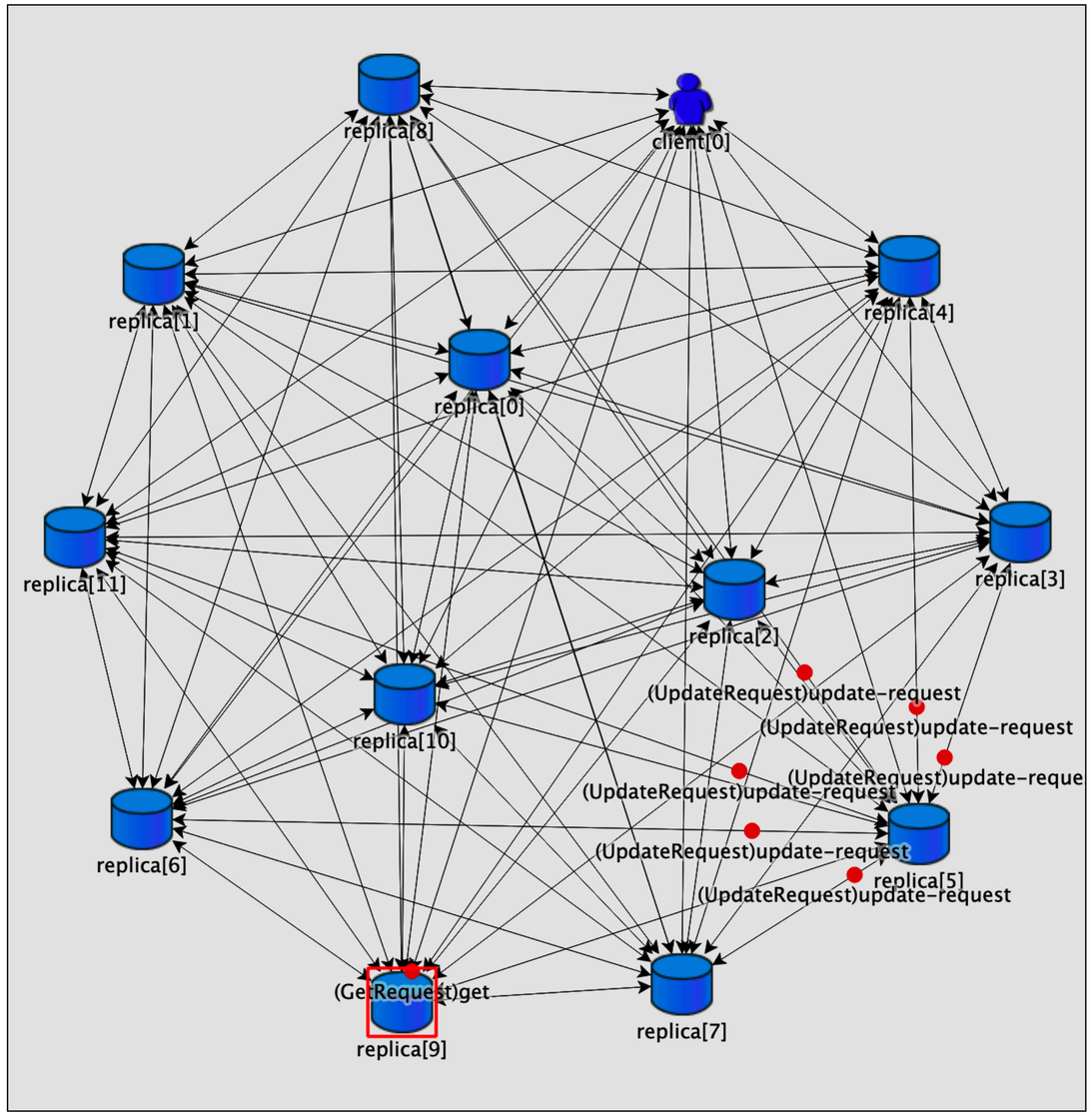
Put request procedure

1. Client sends a put request to a random replica (e.g. 5);



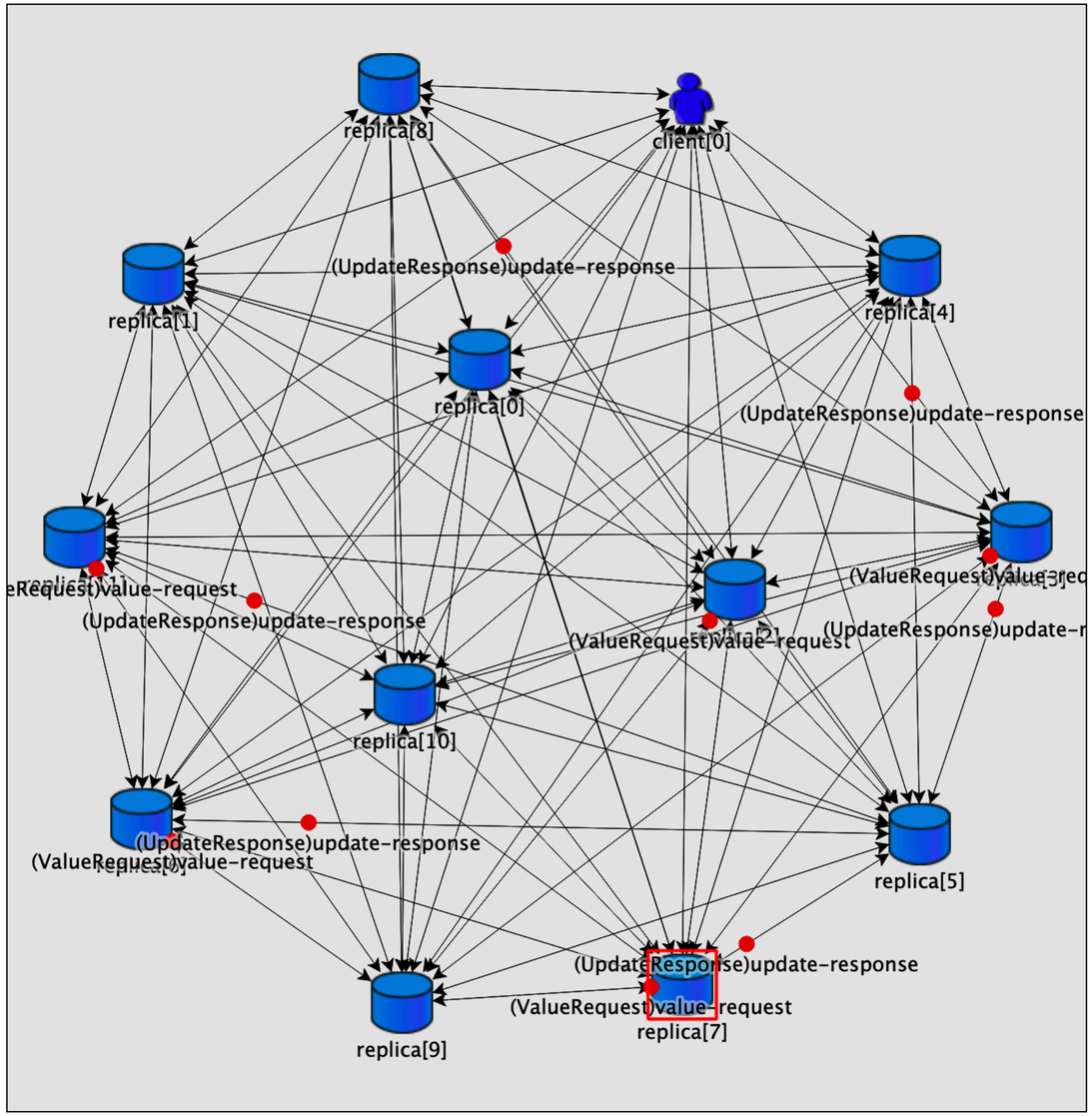
Put request procedure

1. Client sends a put request to a random replica (e.g. 5);
2. Replica 5 forwards the request and waits for responses (update-request);



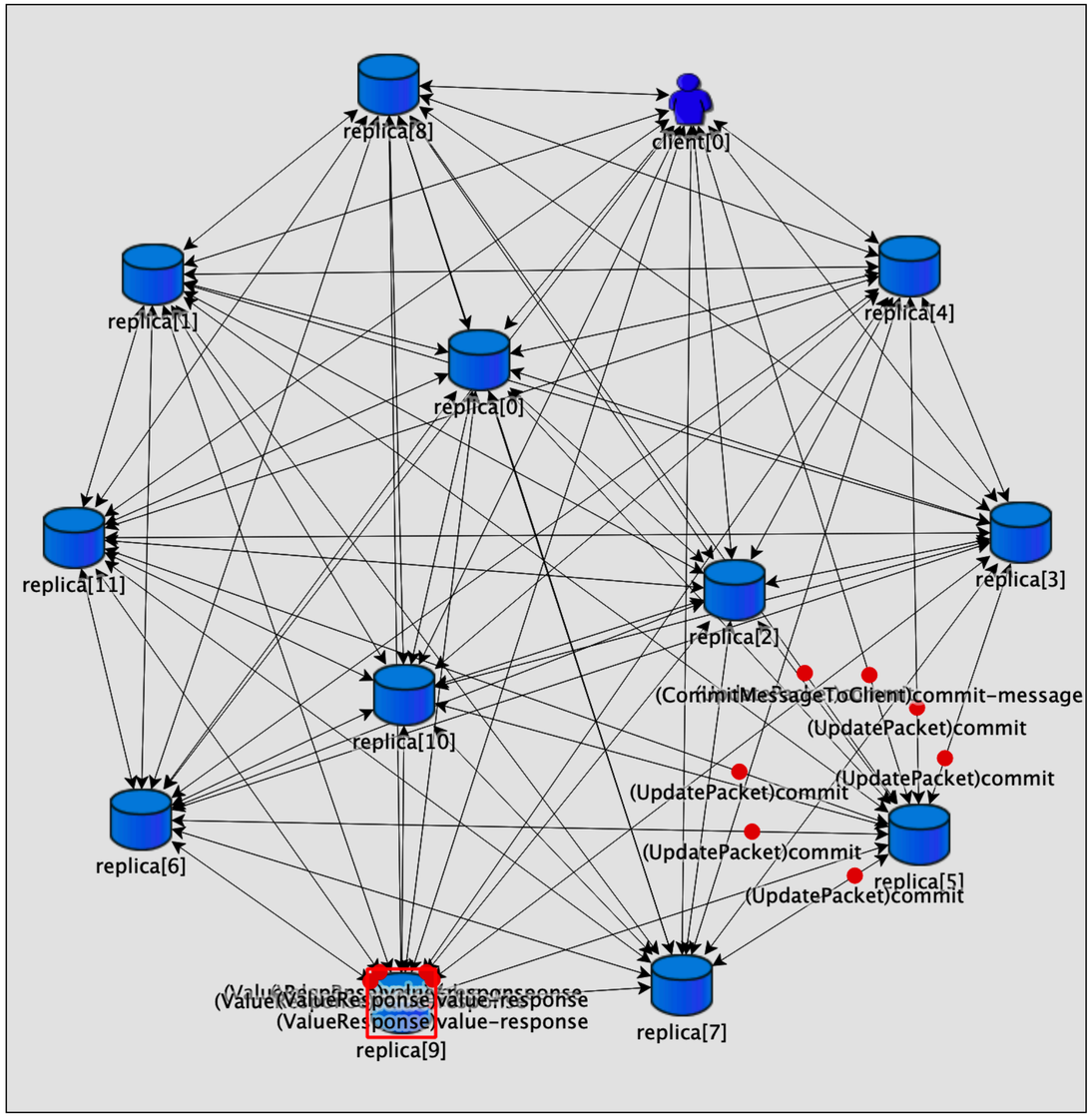
Put request procedure

1. Client sends a put request to a random replica (e.g. 5);
2. Replica 5 forwards the request and waits for responses (update-request);
3. Replicas send back their responses to Replica 5 (update-response);



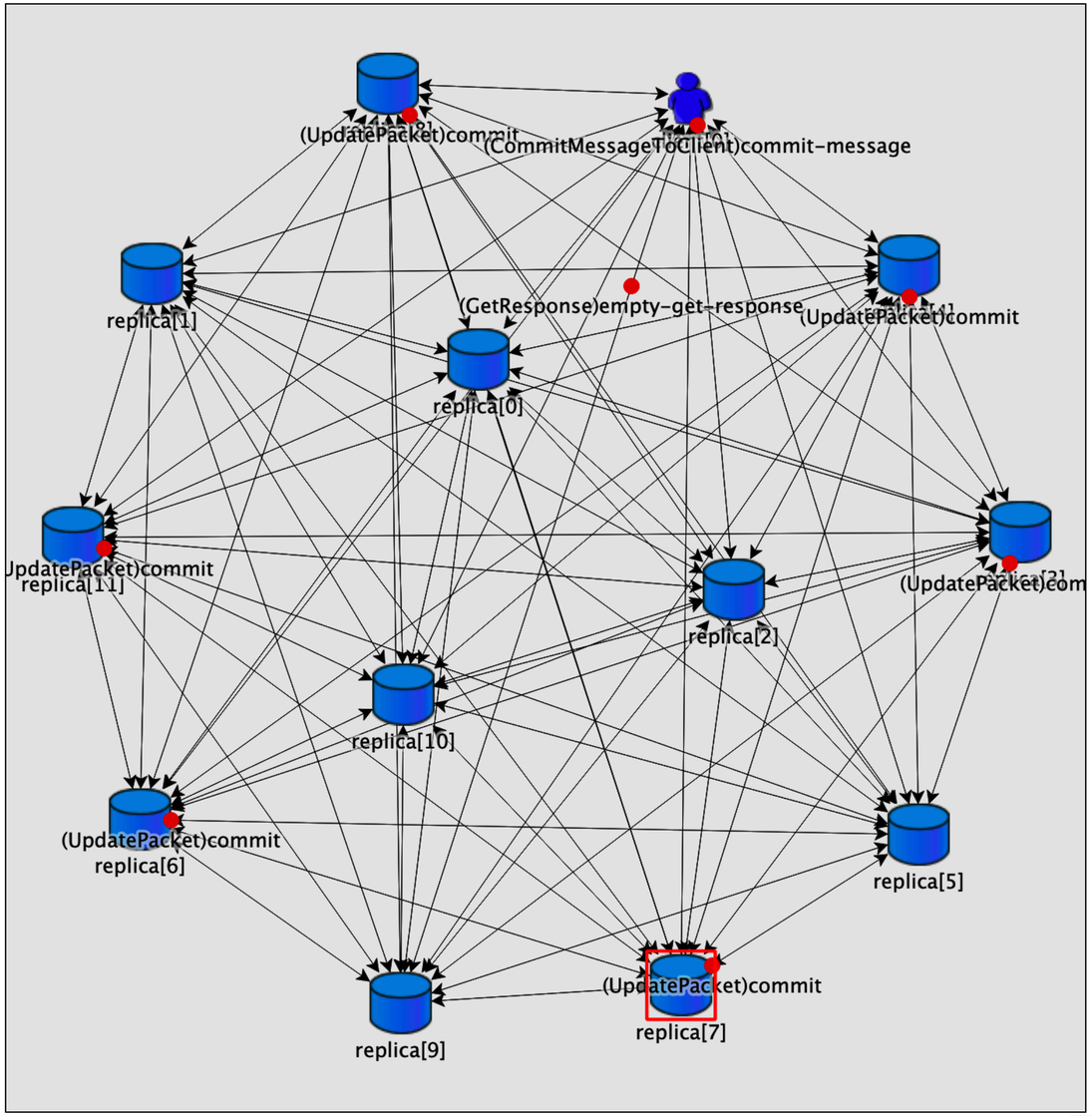
Put request procedure

1. Client sends a put request to a random replica (e.g. 5);
2. Replica 5 forwards the request and waits for responses (update-request);
3. Replicas send back their responses to Replica 5 (update-response);
4. Replica 5 sends a commit-message to all replicas, if no abort-message is received;



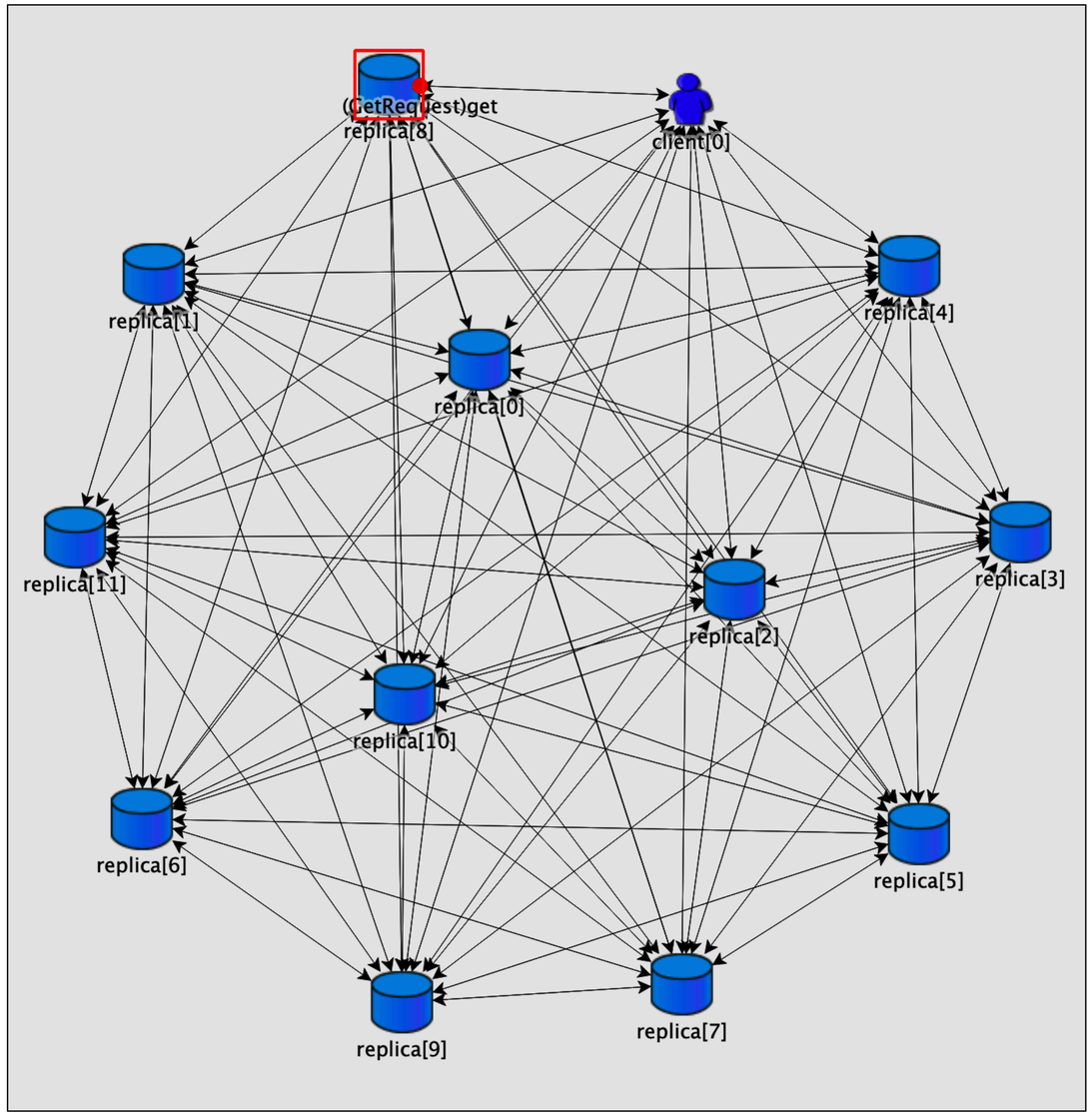
Put request procedure

1. Client sends a put request to a random replica (e.g. 5);
2. Replica 5 forwards the request and waits for responses (update-request);
3. Replicas send back their responses to Replica 5 (update-response);
4. Replica 5 sends a commit-message to all replicas, if no abort-message is received;
5. Replicas and Client receive the commit-message.



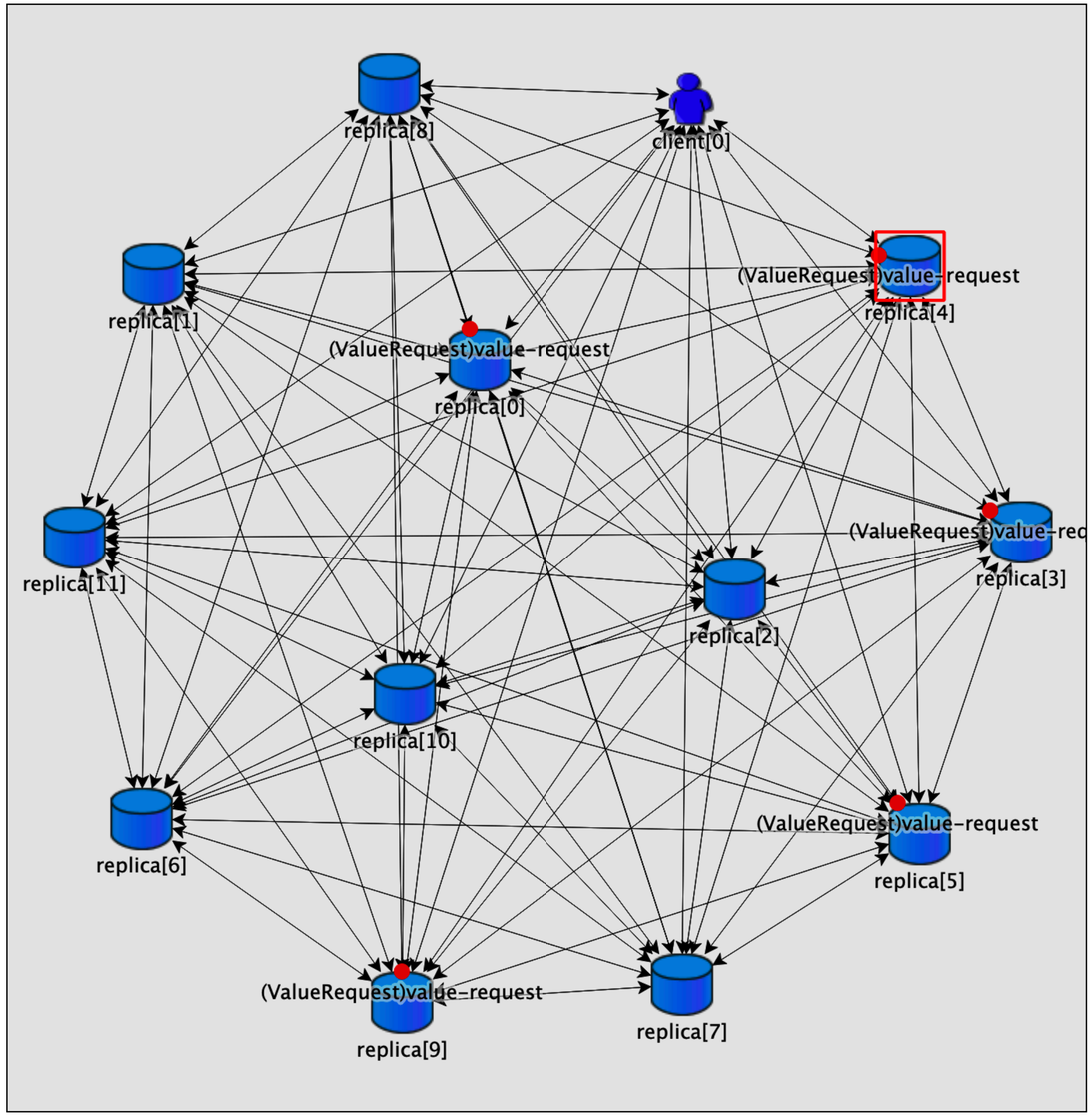
Get request procedure

1. Client sends a get request to a random replica (e.g. 8);



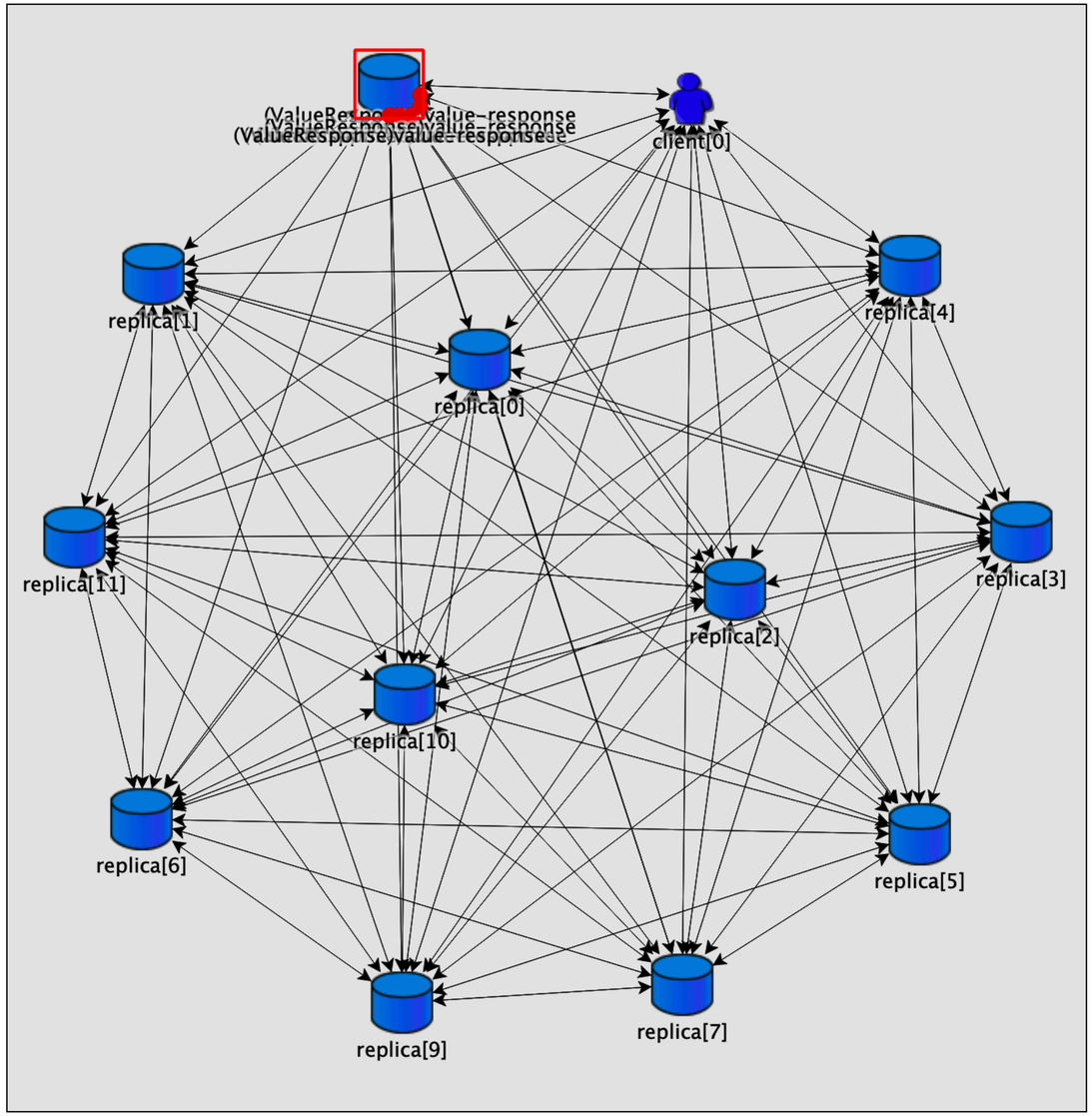
Get request procedure

1. Client sends a get request to a random replica (e.g. 8);
2. Replica 8 forwards the request and waits for responses (value-request);



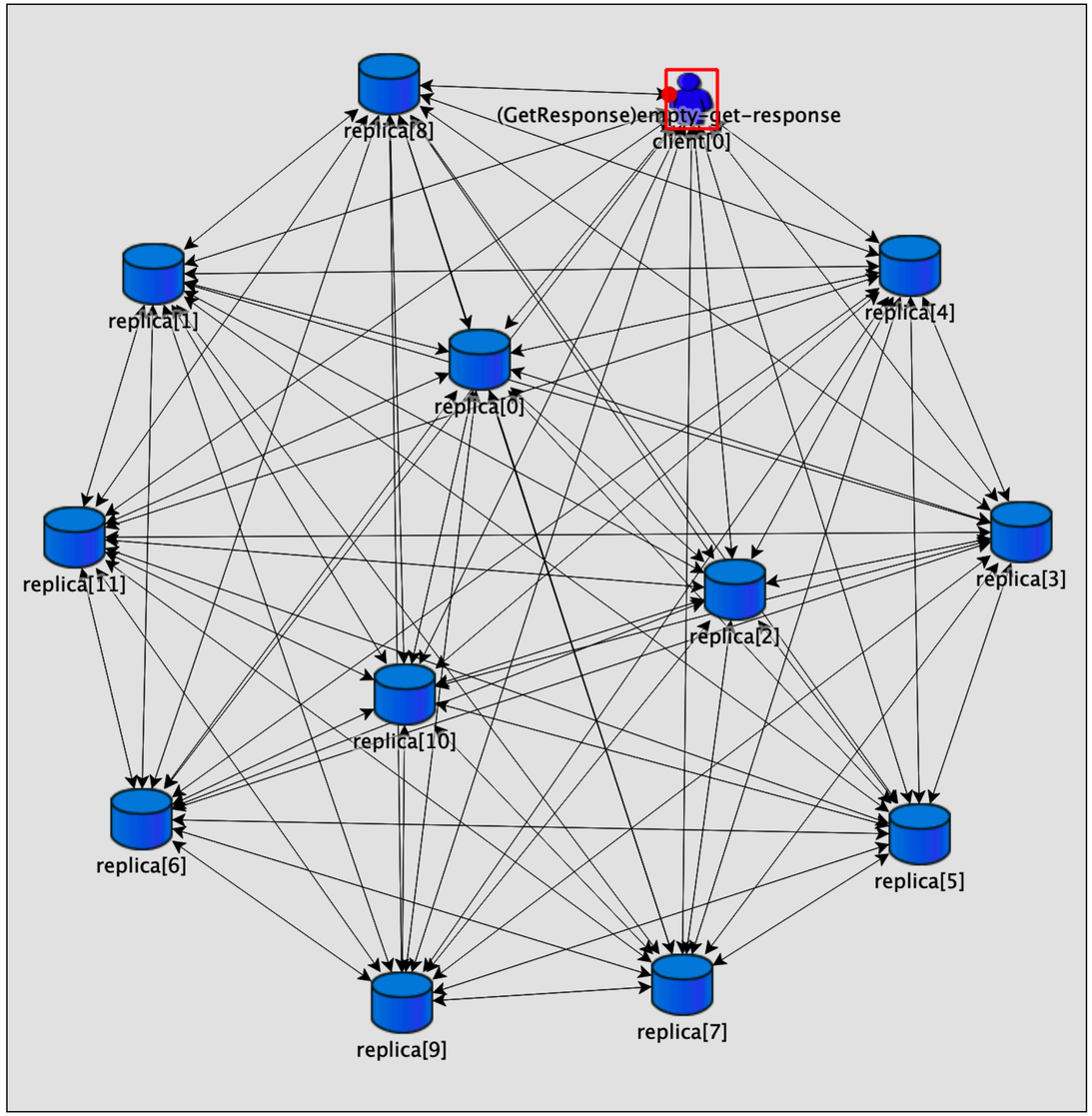
Get request procedure

1. Client sends a get request to a random replica (e.g. 8);
2. Replica 8 forwards the request and waits for responses (value-request);
3. Replicas send back responses to Replica 8 (value-response);



Get request procedure

1. Client sends a get request to a random replica (e.g. 8);
2. Replica 8 forwards the request and waits for responses (value-request);
3. Replicas send back responses to Replica 8 (value-response);
4. Replica 8 sends the get-response to the client (e.g. empty-get-response here).



Performance evaluation

We decided to evaluate the performance by changing the size of the keyspace, the time interval between subsequent put requests of each client and by modifying the quorum parameters (ROWA).

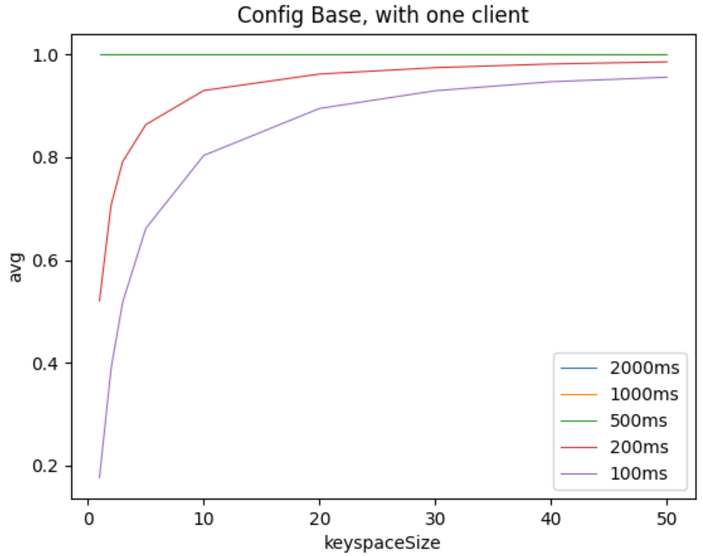
What we expect to notice is:

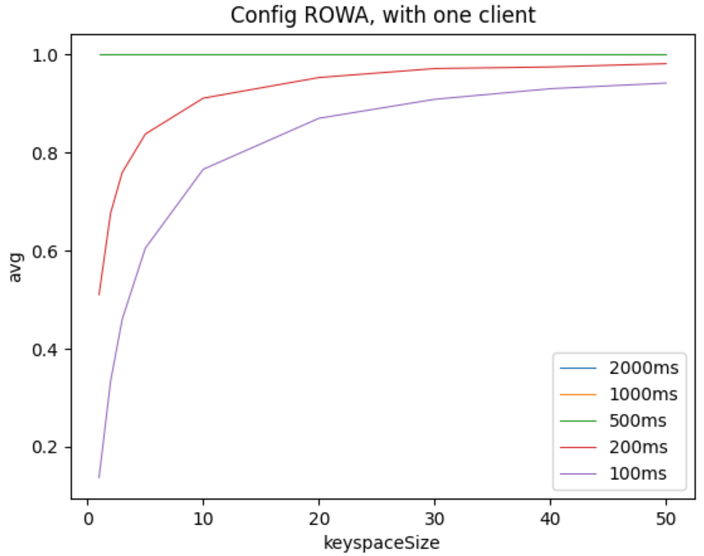
- A decrease in the number of aborted transactions as the size of the keyspace increases;
- A decrease in the number of aborted transactions as the time interval increases;
- An increase in the number of aborted transactions as the number of clients increases;
- An increase in the number of aborted transactions by adopting a Read-One-Write-All policy.

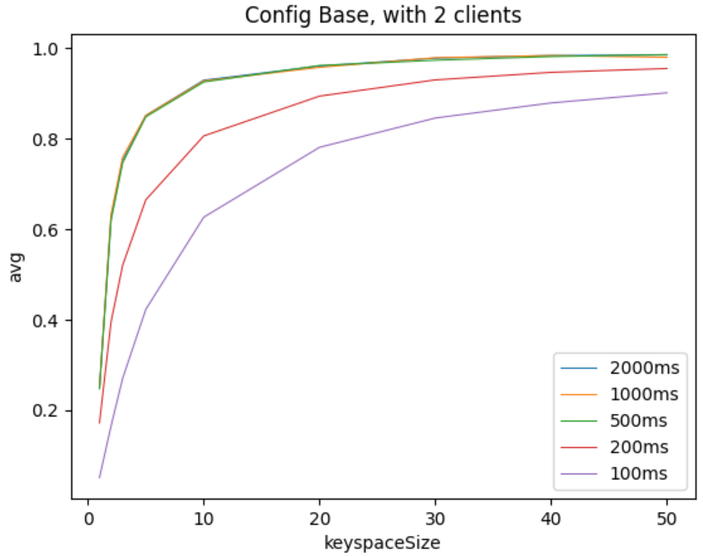
In the next slides we present the result, plotting on the horizontal axis the size of the keyspace, and on the vertical axis the average ratio of committed and total put requests for all clients, using different colors for different time interval between subsequent put requests. In particular, the formula used for the ration in the vertical axis is:

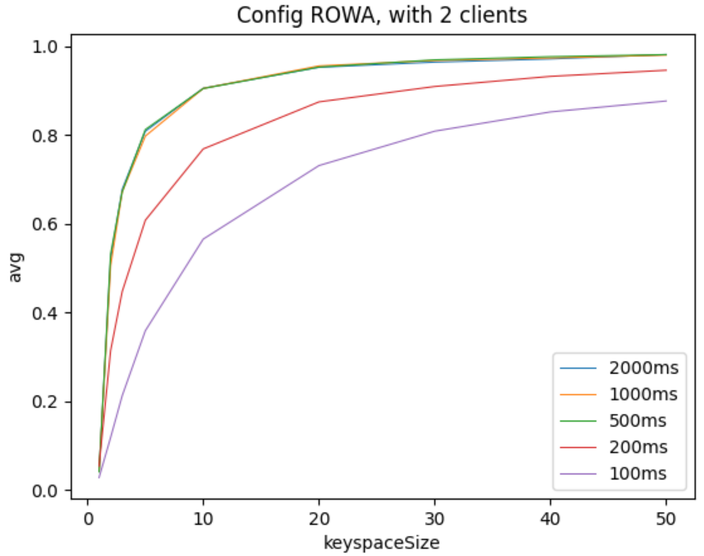
$$avg = \frac{1}{M} \sum_{i=0}^{M-1} \left(\frac{committedPut(i)}{totalPut(i)} \right)$$

Results - 1

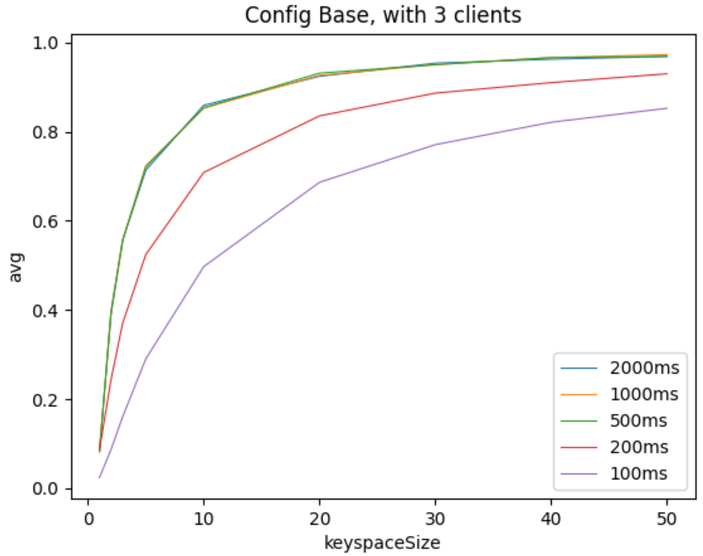

Config Base, with one client
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms

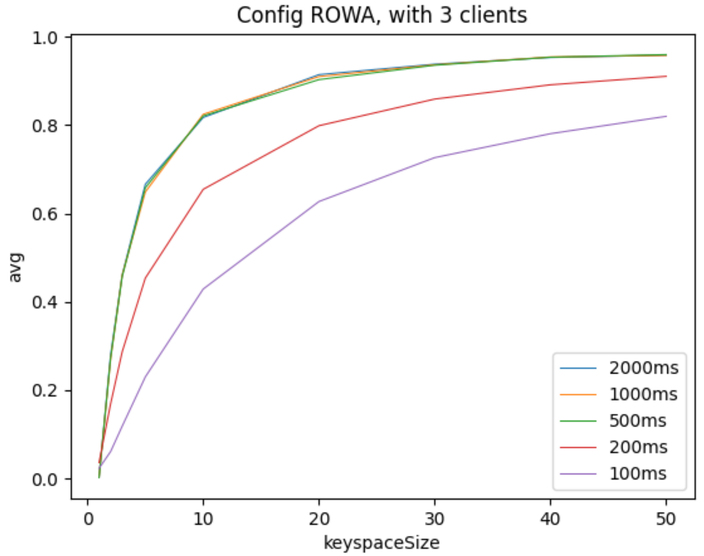

Config ROWA, with one client
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms

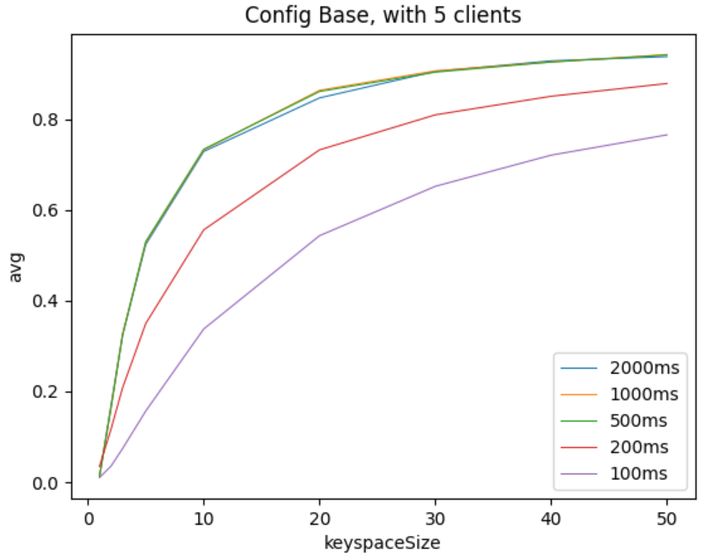

Config Base, with 2 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms

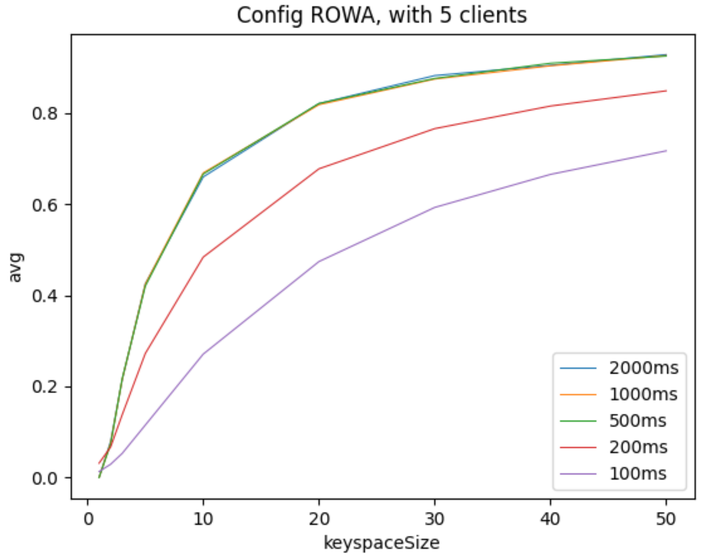

Config ROWA, with 2 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms

Results - 2


Config Base, with 3 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms


Config ROWA, with 3 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms


Config Base, with 5 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms


Config ROWA, with 5 clients
avg
keyspaceSize
2000ms
1000ms
500ms
200ms
100ms

Conclusions

As we observed from the results obtained in the simulation, our expectations have been confirmed and proved to be true.

As a possible solution to the performance decrement, as the number of requests from different clients increases, we may introduce two techniques present in the literature:

- Read-repair, in which clients detect stale responses and send the new value to replicas not up-to-date;
- Anti-entropy, where replicas periodically exchange data in background to remain up-to-date, in addition to read-repair.

Up to now they are not present in our implementation and it might be a good point to start from, if we wish to increase the general throughput of the protocol.