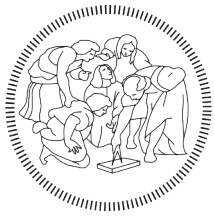

POLITECNICO
MILANO 1863

Distributed Systems Project

Quorum-based replicated datastore

Giovanni Paolino 10696774
Ilaria Paratici 10707097

Academic Year
2022/2023

Contents

1 Project Description

The project consists in the implementation of a distributed key-value store that accepts two operations from clients:

- **put(k, v)** inserts/updates value v for key k;
- **get(k)** returns the value associated with key k (or null if the key is not present).

The store is internally replicated across N nodes (processes) and offers sequential consistency using a quorum-based protocol.
The read and write quorums (NR and NW) are configuration parameters. The project is implemented to be simulated using OmNet++. Assumptions: Processes and links are reliable.

2 Sequential Consistency

Sequential consistency is a consistency model that ensures that the order in which operations are executed by different processes in the system appears to be consistent with a global order of execution.

"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"

Operations within a process may not be re-ordered but all processes see the same interleaving. Sequential Consistency does not rely on time.

If you can find a global sequence of operations that are seen in the same order by all the processes, than there is sequential consistency.

As for sequential consistency to subsist, the order of operations has to be always the same for all processes; two operations are never seen in inverted order by different processes, but some processes may see operations that other processes do not see.

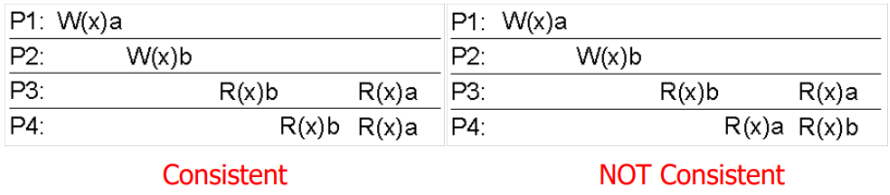

Figure 1: Sequential Consistency

2.1 Leaderless Implementation: Quorum Based

In a leaderless replication protocol, the client contacts multiple replicas to perform the read and write (get and put) operations.
In some implementations (as in our), a coordinator forwards operations to replicas on behalf of the client.

Leaderless implementation uses quorum-based protocols to avoid conflicts: you need a majority of replicas to agree on the write and you also need an agreement on the value to read.

Quorum-based:
About writing (put), an update of resources on replicas occurs only if a quorum of the servers (replicas) agrees on the version number to be assigned.
Reading (get) requires a quorum to ensure the latest version is being read.

Giving:
N: total number of replicas
NR: number of replicas that you need to agree on reading
NW: number of replicas that you need to agree on writing

These parameters are tipically set according to the following rules:

- $\boldsymbol{NR + NW > N}$, which avoids read-write conflicts.
- $\boldsymbol{NW > \frac{N}{2}}$, which avoids write-write conflicts.

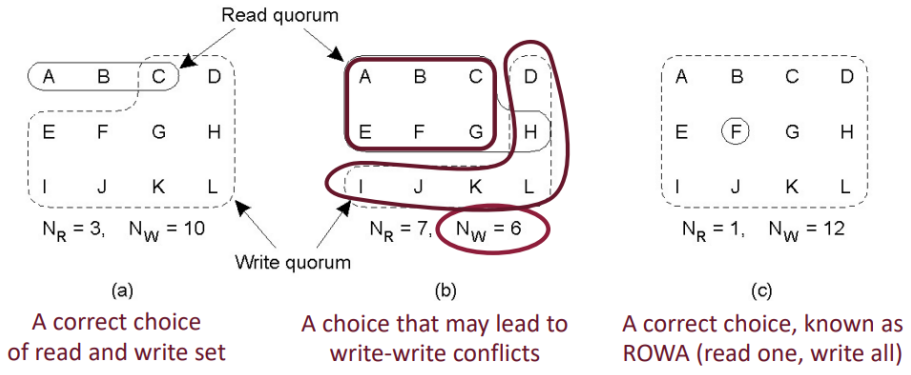

Figure 2: Sequential Consistency

2.2 Wait-Die Algorithm

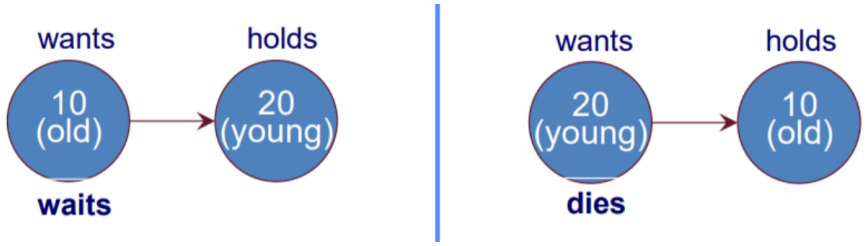

Figure 3: Wait-Die Algorithm, an example

In order to make deadlocks impossible by design we use global timestamps (in our case the Lamport Clock inserted into requests), and then implement the **Wait-Die Algorithm:**
When a process A is about to block for a resource that another process B is using, allow A to wait only if A has a lower timestamp (it is older) than B; otherwise kill the process A.
Following a chain of waiting processes, the timestamps will always increase (no cycles).

In our case, we have requests from clients (as processes) and nodes which handle the requests (as resources).
So the algorithm becomes:
When a request A arrives to a node, that is already handling another request B, the node allows A to wait only if A has a lower timestamp (it is older) than B; otherwise kills the request A.

2.3 Lamport Clock

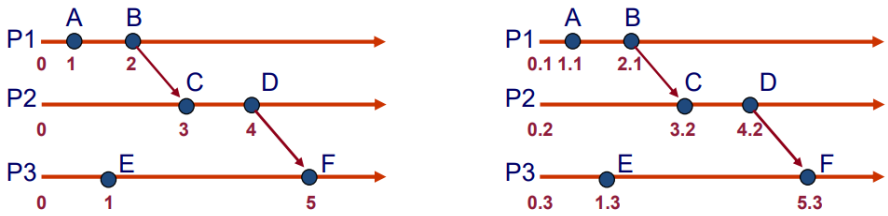

Figure 4: Partial Ordering & Total Ordering

We use Lamport clock in a little bit different way. Processes are clients, the Lamport Clock of the clients is made by:

ID-OF-THE-CLIENT.NUMBER-OF-THE-REQUEST, and it's incremented only when the client creates and sends a new request to the Coordinator.
Requests will contain their Lamport Clock as a field. If the request is a get request the lamport clock is used by the node to insert the request in order into the requestQueue.
In case of a put request, instead, the Lamport Clock is first used to decide if the request has to be killed or to be handled/inserted into the requestQueue, and then, if the put request has to be inserted into the requestQueue, the Lamport Clock is used to order it into the queue.
Nodes check if the number of the request (the decimal part of the Lamport Clock) that arrives is less or equal to the request that the node is handling, in that case, the request will be inserted into the requestQueue, otherwise it will be killed by sending a KILL message to the Coordinator.

Example:

The put message that arrives has a Lamport Clock field 3.3
The node is handling a message with Lamport Clock 5.7
decimal(3.3) $<$ decimal(5.7) $\Rightarrow$ the put message will be inserted in order into requestQueue.
decimal(3.3) $>$ decimal(6.2) $\Rightarrow$ the put message will be killed.

Also, if the decimal part of the put request is the same as that of the handled request, but the number of the client making the request is less than that of the one handled by the node, the request will be put in the queue, otherwise it will be killed.

Example:
The put message that arrives has a Lamport Clock field 3.3
The node is handling a message with Lamport Clock 4.3
decimal(3.3) = decimal(4.3) and
integer(3.3) $<$ integer(4.3) $\Rightarrow$ the put message will be inserted in order into requestQueue, otherwise it is killed.

3 Implementation

3.1 Topology

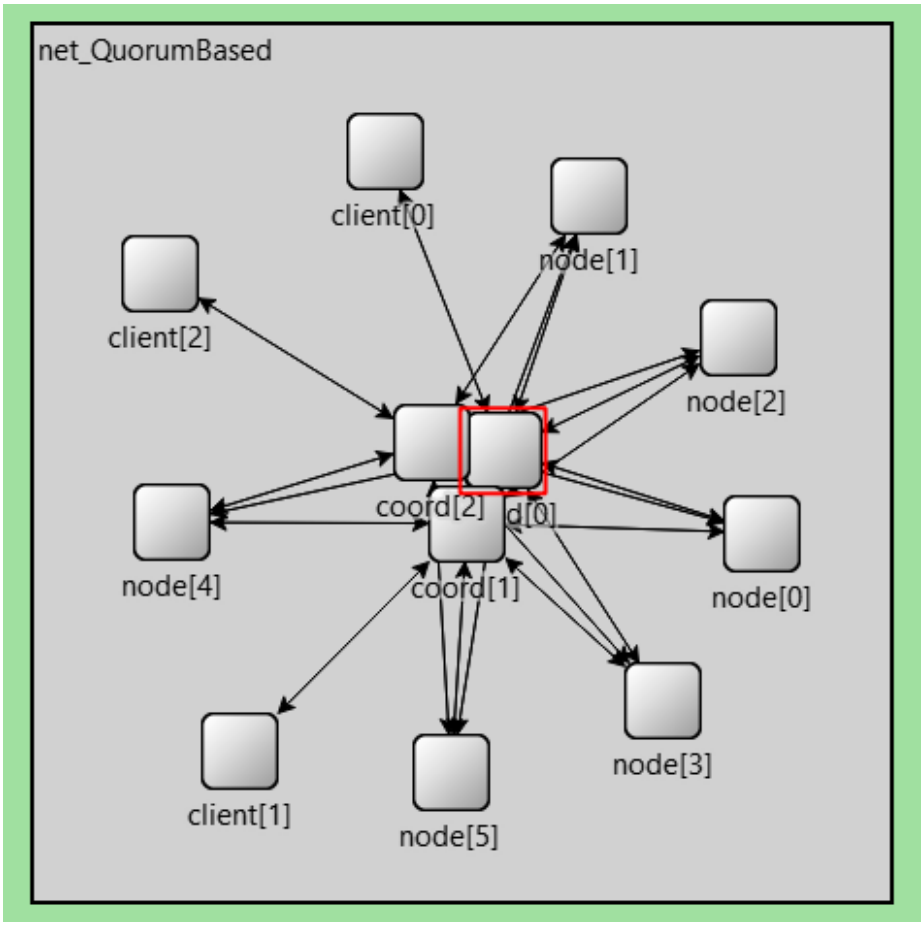

Figure 5: Star Topology

Each Client is connected with his own Coordinator, which is connected to all the Nodes in a Star Topology.

3.2 Client

All the requests start from the Client, which creates GET and PUT requests and sends them to his own Coordinator, that's in charge of communicate with the nodes for him.
The list of request that the node has to perform is read from the **input.txt** file, and it is put into a queue at Client side.

The Client then starts sending the requests to the Coordinator, one at a time, waiting for the answer from the Coordinator before sending the next request.

3.3 Coordinator

The Coordinator acts as a Client from the point of view of the Node.
It forwards the Client's requests to NR nodes if the request is a GET or to NW nodes if the request is a PUT, then waits for the answers by the Nodes, and elaborates the answer it has to send to the Client.
The Coordinator has a response queue, that it uses to store the responses from the nodes to be evaluated before sending the response to its Client.

If the Coordinator receives a GET response from a Node, it waits to have at least NR responses in the queue and, if the resource is found by the Nodes, the Coordinator sends back to the Client the latest version of the resource and its value.
If the Coordinator receives a PUT response, it waits to have at least NW responses in the queue, it finds the latest version and then sends a COMMIT Message to the Nodes and the response message to the Client with key and value committed.
If the Coordinator receives a KILL message from any Nodes, it reschedules its request after a while and, if it's the first kill message it receives, it forwards the kill message to the other nodes to which it had sent the request yet.

3.4 Node

Each Node has it's own resources queue in which all the resources the Node owns are stored with their own key, value and version.
The resource queue of a Node is initialized from the file inizializzazione.txt and is updated every time the Node receives from the Coordinator a COMMIT message for a PUT request.
A Node can receive the following types of message: GET, PUT, KILL, COMMIT.
It uses a request queue to store the requests (GET or PUT) to handle.
Basing on the type of message received it handles it:

- **GET or PUT**:
  If the requests queue is empty, and it's not already handling another request, it handles the GET/PUT request.

If it's handling a GET, it answers the coordinator by indicating the version and the value of the requested resource it owns.
If it is handling a PUT, instead, it waits for a COMMIT (or a KILL) message from the Coordinator before updating the resource in the resource queue and answering to the Coordinator with the putAck message (or cancelling the PUT request in the case of a KILL message).

If the request queue is not empty or it is already busy handling another request, it puts the GET/PUT request that just arrived into the request queue, ordering it by its Lamport's Clock; If the arriving request is a put request and it has a higher Lamport Clock with respect to the one of the request that the node is handling (it's more recent), than the Node will kill the arriving put request by sending a KILL message to the Coordinator and doesn't insert the request into its request queue.

- **KILL or COMMIT**:
  If it receives a COMMIT, either overwrites the value and the version into the resource queue, or creates a new resource, if it's not already existing.

  The Node will also insert the new value for the resource into the corresponding output vector of the simulation in order to let us visualize the evolution of the resources' values.

  In case of reception of a KILL, the Node pops the request from the queue and sends a KILL Message to the Coordinator, that will forward the KILL to all the nodes that were contacted asking for the PUT that's being killed.

After sending the answer to the coordinator, the node takes from the queue the next request to be handled and starts handling it as explained before.

4 Results

For testing and results we chosen:

- N = 6
- NW = 4
- NR = 3
- NumClient = 4

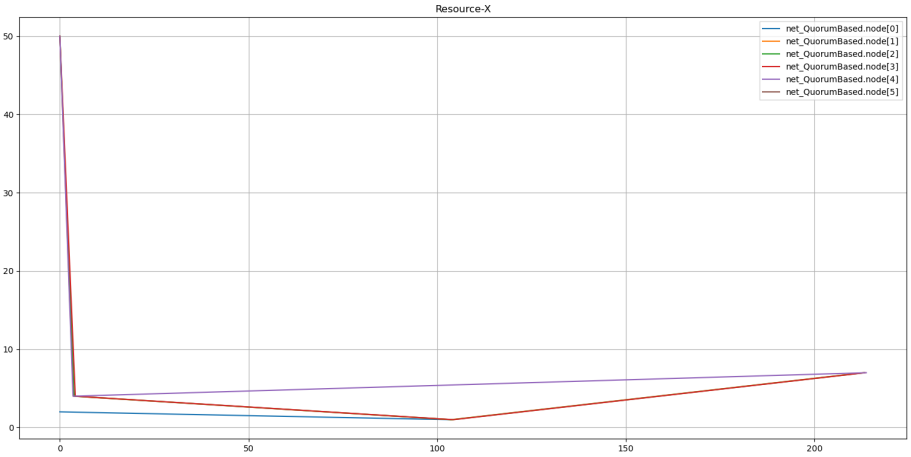

Figure 6: VectorX

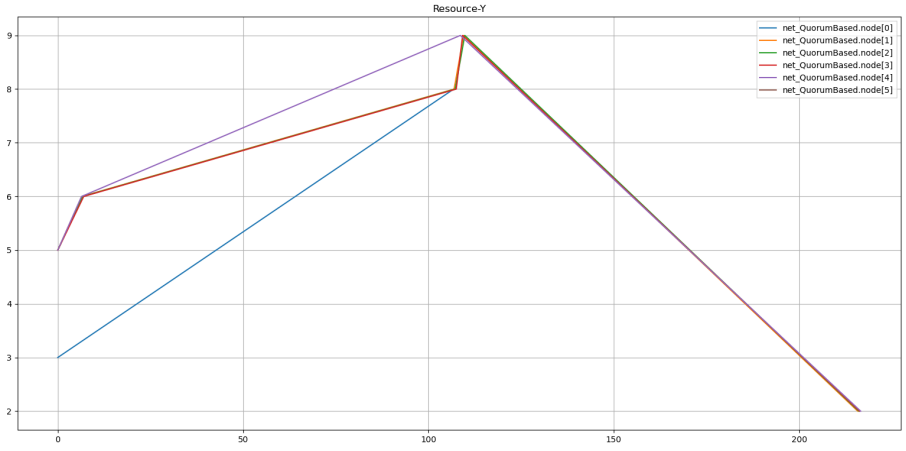

Figure 7: VectorY

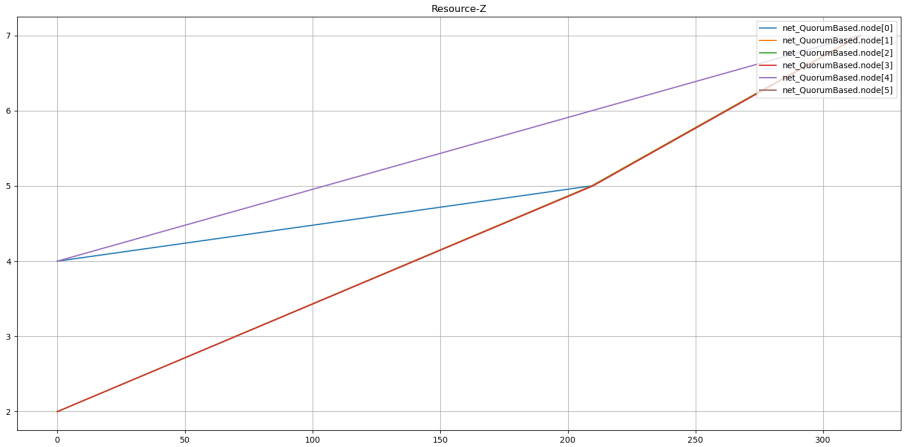

Figure 8: VectorZ

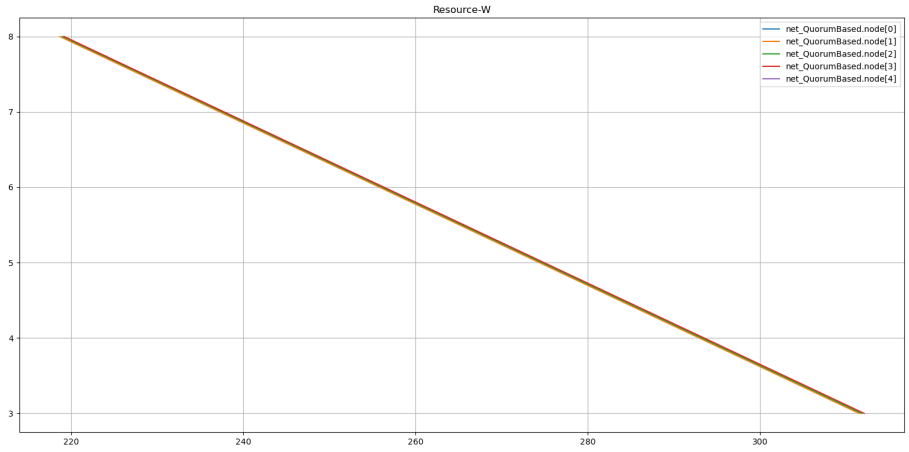

Figure 9: VectorW