Course Project:
Distributed Key-Value Store with Data Partitioning and Replication

Distributed Systems 1, 2022-2023

In this project, you will implement a DHT-based peer-to-peer key-value storage service inspired by Amazon Dynamo. The system consists of multiple storage nodes (just nodes hereafter) and provides a simple user interface to upload/request data and issue management commands.

The stored data is partitioned among the nodes to balance the load. The partitioning is based on the keys that are associated with both the stored items and the nodes. For simplicity, we consider only unsigned integers as keys, which form a circular space or "ring", i.e., the largest key value wraps around to the smallest key value like minutes on analog clocks. A data item with key K should be stored by the first N nodes in the clockwise direction from K on the ring, where N is a system parameter that defines the degree of replication. For example, if $N = 2$ and there are three nodes in the system with IDs 20, 30, and 40, a data item with the key 15 will be kept by nodes 20 and 30, while an item with the key 35 will go to nodes 40 and 20.

All the nodes are required to know all the other nodes in the system, allowing them to locally decide which of them are responsible for a data item. When nodes leave or join the network, the system should repartition the data items accordingly.

Your implementation should support management services and create clients supporting data services. The data service consists of two commands: `update(key, value)` and `get(key)->value`. Any storage node in the network must be able to fulfill both requests regardless of the key, forwarding data to/from appropriate nodes. The node contacted by a client for a given user request is called *coordinator*. The management service consists of a `join` operation that creates a new node in the storage network and a `leave` operation to remove a node, instead.

1 Replication

To implement replication, the system relies on quorums. Versions are associated internally with every data item to support consistent reads. The system-wide parameters W and R specify the write and read quorums, respectively. Note that $W \leq N$ and $R \leq N$. If an operation cannot be completed within a given timeout T, the coordinator reports an error to the requesting client.

Read. Upon receiving a `get` command from a client, the coordinator requests the item from the N nodes responsible for it. For better availability, the request coordinator sends the data item back to the requesting client as soon as R replies arrive.

By exploiting version numbers, your implementation must provide sequential consistency. Read and write operations appear to take place is some total order, consistent with the order of operations issued by each process (client).

Write. When a node receives an `update` request from the client, it first requests the currently stored version of the data item from the N nodes that have its replica. For better availability, the write coordinator reports success to the client as soon as it receives the first W replies required to support sequential consistency. It then sends the update to all N replicas, incrementing the version of the data item based on the W replies. Otherwise, on timeout, the coordinator informs the client of the failed attempt and stops processing the request. For simplicity, we assume that there are no failures within the request processing time.

Local storage. Every node should maintain a persistent storage containing the key, the version and the value for every data item the node is responsible for. You can simulate persistent storage in your implementation using a suitable data structure, whose data does not get deleted upon crash failures.

2 Item repartitioning

Repartitioning is necessary when a node joins or leaves the network. Note that both operations are performed only upon external request. A crashed node should not be considered as leaving but only temporarily unavailable; therefore, there is no need to implement a crash detection mechanism or repartition the data when a node cannot be accessed.

Joining. The key of the newly created node is specified in the `join` request together with the actor reference of one of the currently running nodes, which becomes its bootstrapping peer. First, the joining node contacts the bootstrapping peer to retrieve the current set of nodes constituting the network. Having received this information, the joining node should request data items it is responsible for from its clockwise neighbor (which holds all items it needs). It should then perform read operations to ensure that its items are up to date. After receiving the updated data items, the node can finally announce its presence to every node in the system and start serving requests coming from clients. Upon learning about a new node, the others should remove the data items assigned to it they are no longer responsible for.

Leaving. A node can be requested to leave the network. To do so, the leaving node announces its departure to the others and passes its data items to the nodes that become responsible for them after its departure.

Recovery. When a crashed node is started again, instead of performing the join operation it requests the current set of nodes from a node specified in the recovery request. It should discard those items that are no longer under its responsibility (due to other nodes joining while it was down) and obtain the items that are now under its responsibility (due to nodes leaving).

3 Other requirements and assumptions

- The project should be implemented in Akka, with nodes being Akka actors.
- The clients execute commands passed through messages, print the reply from the coordinator, and only then accept further commands. A client may perform multiple read and write operations.
- A node must be able to serve requests from different clients, and the network must support concurrent requests (possibly affecting the same item key).
- Nodes join and leave, crash and recover one at a time and only when there are no ongoing operations. Operations might resume while one or more nodes are still in crashed state.
- The network is assumed to be FIFO and reliable.
- Use (unsigned) integers for keys, and strings (without spaces) as data items. The keys are set by the user to simplify testing (though in real systems random-like hash values are used).
- The replication parameters N, R, W, and T should be configurable at compile time.

4 Project report

You are asked to provide a short document (typically 3-4 pages) in English explaining the main architectural choices. Your report should include a discussion on how your implementation satisfies consistency requirements (sequential consistency).

In the project presentation slides you can find a list of relevant questions your document should answer.

5 Grading criteria

You are responsible to show that your project works. The project will be evaluated for its technical content, i.e., algorithm correctness. A correct implementation of the whole requested functionality is worth 6 points. It is possible to submit a project with a reduced level of complexity, without the replication feature ($N = W = R = 1$), data versions and support for recovery. A correct implementation of this reduced function set is worth 3 points.

You are expected to implement this project with exactly one other student. However, the marks will be individual, based on your understanding of the program and the concepts used.

6 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (davide.vecchia@unitn.it), well in advance, i.e. a couple of weeks before the presentation.
- You can present the project at any time, also outside of exam sessions. In the latter case, the mark will be "frozen" until you can enroll in the next exam session.
- The code must be properly formatted otherwise it will not be accepted.
- Both the code and the report must be submitted in electronic format via email at least one day before the meeting. The report must be a single self-contained pdf/txt. It must be sent together with all code in a single tarball consisting of a single folder (named after your surnames). For instance, if your surnames are Rossi and Russo, put your source files and the report in a directory called `RossiRusso`, compress it with "`tar -czvf RossiRusso.tgz RossiRusso`" and submit the resulting `RossiRusso.tgz`.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. Do not hesitate to ask questions if you run into troubles with your implementation. We are here to help.