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

Distributed Systems 2016 – 2017

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'll consider only unsigned integers as the 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 the 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.

Every node provides the data and the management services to clients (user applications). The data service consists of two commands: `update(key, value)` and `get(key)->value`. Any node in the network must be able to fulfil both requests regardless of the key, forwarding data to/from appropriate nodes. The management service consists of a single `leave` command that requests the node to leave the network. We'll call the node a client connects to the *coordinator* for the given user request.

1 Replication

To implement replication, the system relies on quorums and versions that are associated internally with every data item. System-wide parameters W and R specify the write and read quorums respectively ($W + R > N$). For better availability, the request coordinator should reply to the client as soon as the quorum is reached, or report an error if the quorum was not reached within a timeout T.

Read. Upon receiving a `get` command from the client, the coordinator requests the item from the N responsible nodes. As soon as R replies arrive, it sends the data item with the highest received version back to the 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. As soon as $Q = max(R, W)$ replies have been received, the most recent stored version can be reliably determined because of the read quorum and the item can be reliably updated because of the write quorum (we assume that there will be no failures within the request processing time). In this case, the coordinator reports success to the client and then sends the update to all N replicas, incrementing the version of the data item. Otherwise, on timeout, the coordinator informs the client and stops processing the request. If the network contains fewer than N nodes, any write request should immediately return an error and stop.

Local storage. Every node should maintain a persistent storage (text file) containing the key, the version and the value for every data item the node is responsible for. It should be in the form of a space-separated table.

2 Repartitioning

Repartitioning is necessary when a node joins or leaves the network. Note that both operations are performed only upon a user request. The nodes are launched one at a time, joining the existing peer-to-peer network at start. A crashed node should not be considered left but only temporarily unavailable, therefore there is no need in implementing a crash detection mechanism or repartition the data when a node cannot be accessed.

Joining. The key of the new node is specified as a start parameter together with the address of any of the currently running nodes. First of all, the joining node contacts the specified node (bootstrapping peer) to request the current set of nodes constituting the network. Having received the list of nodes, the new node should request data it is responsible for from its clockwise neighbour, as it has all the data the new node needs. After receiving the data, the node can finally announce its presence to every node in the system. Having learned about a new node, the others should remove data items they are not responsible for any more. Having joined the network the new node should start serving requests coming from clients and other nodes.

Leaving. A node may be requested to leave the network, through the client interface. To do so, the node announces to the rest of the network that it is leaving and passes its data items to N clockwise peers that do not have them but become responsible for them due to the leave.

Recovery. When a crashed node is started again, instead of performing the join operation it requests the current set of participants from the specified network node and loads its local storage file. It should discard those items that became out of its responsibility due to other nodes joining while it was down.

3 Other requirements and assumptions

- The project should be implemented in Akka, with nodes being Akka actors.
- Every node should be started in a separate Java machine, possibly on a separate host.
- The client should be a dedicated Akka application agnostic of the internal structure of the DHT network. It should execute the command passed as the command line arguments and terminate after receiving and printing the reply from the node.
- It is not required for a node to serve multiple simultaneous client requests or for the network to handle simultaneous updates to the same key. Though concurrent requests served by different coordinators and affecting different keys should be supported.
- The storage file should be written to disk every time the local data is changed. For simplicity, use the blocking Java IO API for file operations.
- Nodes join, leave or crash one at a time when there are no ongoing requests.
- The network is assumed to be reliable.
- Use 16-bit unsigned integers for the keys and strings (without spaces) as the 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 Grading and the levels of complexity

You are responsible to show that your project works. The project will be evaluated for its technical content (algorithm correctness). *Do not* spend time implementing unrequested features — focus on doing the core functionality, and doing it well.

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 versioning 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.

5 Presenting the project

- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistant (timofei.istomin@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 enrol in the next exam session.
- The code must be properly formatted otherwise it will not be accepted (e.g. follow style guidelines).
- Provide a short document (1-3 pages) explaining the main architectural choices.
- Both the code and documentation must be submitted in electronic format via email at least one day before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surnames) containing all your source files. For instance, if your surnames are Rossi and Russo, put your source files and the documentation 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.