Course Project:
Distributed Key-value Store with Data Partitioning

Distributed Systems 2014 – 2015

In this project, you will implement a DHT-based peer-to-peer key-value storage 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 based on the value of keys identifying 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). We'll say that a key K_1 is less than K_2 if the shortest arc connecting them is in the clockwise direction from K_1. The nodes, too, have associated keys and therefore might be placed as points on the circle. A node is responsible for storing data items with keys that are less or equal than its own key but greater than the key of its counter-clockwise neighbour.

The interface for data operations 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 node(s) if required. We'll call such a node *coordinator* of the user request.

The user interface might be implemented as a separate client application or integrated into the node application. In case of a separate client, it may connect to any of the nodes and should be agnostic of the distributed nature and internal organization of the system.

1 Partitioning (2 points)

The program(s) should implement the key-value store with data partitioning described above. The nodes are launched one at a time, joining the existing peer-to-peer network at start.

Joining. The key of the new node is specified as a start parameter together with the address of any of the currently running nodes. The joining node contacts the specified peer to request the current set of nodes constituting the network and then announces its presence to every node in the system. Then it requests a transfer of the data that falls into its responsibility range from its clockwise neighbour. Having completed these steps the joined node should start serving requests coming from the user and the peer nodes.

Leaving. A node may be requested to leave the network, through the user interface. To do so, the node announces to the rest of the network that it is leaving and passes all its data items to its clockwise neighbour.

Local storage. The keys and the associated data items kept by a node should be stored in a text file. After joining the network, a newly started node should read the contents of the file considering the records as a sequence of `update` requests.

2 Replication (2 points)

In order to tolerate faults (crashes) the system relies on replication. In addition to storing the item at node A, it gets replicated to A's next $N-1$ clockwise neighbours (N nodes in total). When a node joins or leaves, the system should move data items accordingly. There are more two system-wide parameters, W and R, specifying the write and read quorums respectively ($W + R > N$); a version number is associated internally with every stored item.

Write. When a node receives an `update` request from the user, it first requests the currently stored version number of the object from at most N nodes that have its replica. If less than $Q = max(R, W)$ nodes responded to the request, the coordinator reports an error to the user and stops further processing of the user request.

If Q replies have been received, the most recent stored version number 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 node/network failures within the request processing time). In this case, the coordinator reports success to the user request and sends the update to all N replicas.

Read. Upon receiving a `get` command, the coordinator requests the item from at most N responsible nodes. As soon as R replies arrive, it retransmits the data item with the highest received version number to the user.

Recovery. To recover, a crashed node requests the current set of participants from any network node and loads its local storage file without performing the join procedure. The node should also pass and/or retrieve data items to or from other nodes if there were joined or left participants while the node was down.

3 Concurrent replication (2 points)

To make the system highly responsive, the read and write requests should be sent to the replicas in parallel (e.g. using separate threads) and the user should get the reply as soon as possible, i.e. immediately after the first Q (or R) nodes reply in case of `update` (or `get`) operation, or when a timeout T triggers.

4 Other requirements and assumptions

- It should be possible to run nodes on separate hosts.
- It is not required for a node to serve multiple simultaneous user 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 data the node maintains is changed, the file name should be configurable at node start (to simplify testing).
- Nodes join, leave or crash one at a time when there are no ongoing requests.
- Use 16-bit unsigned integers for the keys and fixed-length strings (e.g. 10) 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 maximum number of nodes supported by the system ($M \geq 10$) and the maximum number of locally stored items ($I \geq 100$) should be configurable at compile time, as well as the replication parameters N, R, W and T.
- It is preferred that the client implements a command-line interface, though GUIs are also acceptable.

5 Miscellaneous

Technology. You are free to implement this project using either plain sockets or RPC. The use of RPC will be considered more positively. You can in principle use any programming language but, unless you are doing it in Java or C, *ask the instructor first*, as your choice might not be suitable.

Grading. You are responsible to show that your project works; make sure you provision the code to optionally simulate delays at the key points of the protocol (to test multiple concurrent activities). You will be penalized for:

- improper use of synchronization mechanisms. You must not use busy waiting, or "sleep" statements to synchronize your code, and you must properly identify and protect the critical sections of your code;
- sockets that are kept open when there is no communication going on.
- presentations where you do not show that you understand the algorithms and basic aspects of the concepts you used in your implementation (sockets / RPC, threads, mutexes, etc.)

Provided code. Sample code that you may use in your program is provided. It contains basic marshalling routines (for socket-based implementations), functions to store/load data items in a file, and an example of how to run multiple copies of an RPC server on a single computer.

People. You are expected to implement this project with exactly one other student.

How/what to present

- 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. Follow style guidelines (e.g. this one).
- Provide a short document (2-3 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 in a directory called `RossiRusso`, compress it with "`tar -czvf RossiRusso.tgz RossiRusso`" and submit the resulting `RossiRusso.tgz`. Do not include binaries, nor the documentation in the tarball.
- 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.
- 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.