Fall 2015, Inf-3200 Mandatory Assignment 1: Distributed key-Value Store

In this assignment, you are to design and implement a distributed key-value storage system. You will run your system on the uvocks cluster (urocks.cs.uit.no).

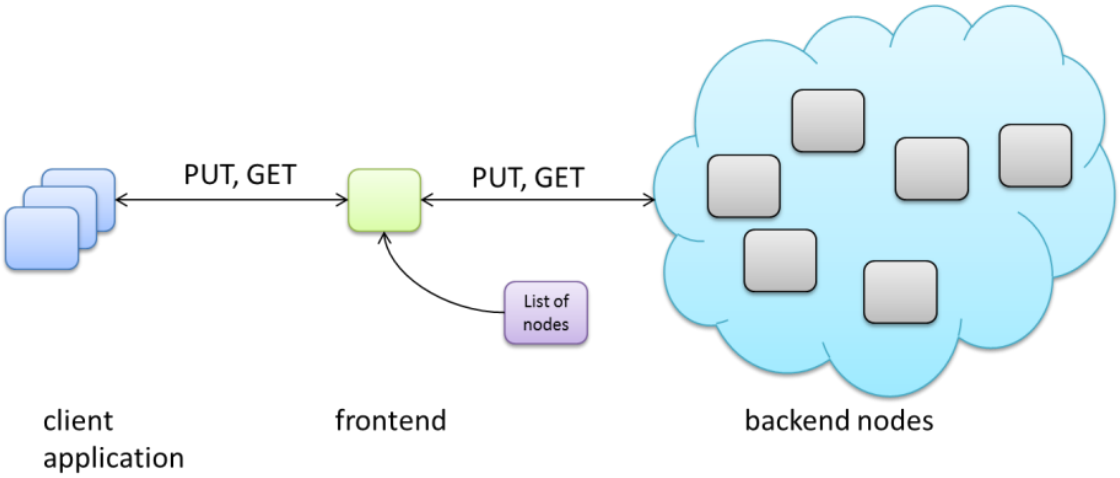

Figure 1:Key-value store architecture

Distributed data stores

A *distributed data store* is a network of computers providing a service for storing and retrieving data. Compared to a single machine system, distributing the data across multiple machines *may* improve performance, increase scalability and/or provide better fault-tolerance. These benefits come with the cost of higher complexity and a bigger developing effort. A *key-value store* is a type of distributed data store designed for storing and retrieving records identified by a unique key.

Processes across multiple machines provide a single service, passing messages between one another. The processes communicate according to the rules of the chosen architecture. Structured and unstructured *peer-to-peer* architectures are both suitable architectures for building a distributed data store. A *distributed hash table* is the most common way of organizing the processes in peer-to-peer systems.

The data is distributed between the nodes through a partitioning scheme. The partitioning scheme dictates which node is responsible for storing a given key *k*. A simple solution would be to simply assign *k* to a random server, with the cost of lookup being O(N) given N nodes. A more refined solution is to assign nodes with the responsibility of their respective partition of a key-space (e.g. assign *node1* with keys in the range *a-p node2* with q-z). Other solutions include consistent hashing (used in the Chord [1] protocol). For more information on these types of systems, you can look at Windows Azure Storage [2] and Dynamo [3].

Frontend

Figure 1 shows the high-level architecture of the system. A client application issues PUT and GET requests to the storage system. The frontend node is responsible for forwarding client requests to the backend nodes that holds the data. You are not required to implement a client application; however, you are required to implement the service that supports these types of requests.

Key Requirements

1. The only allowed state is a list of backend nodes. I.e. no data is stored on the frontend node.

2. Forward the request to a random backend node

You are provided with sample code to get you started. The Python script runs a web server to handle PUT and GET requests. It also simulates client behavior by inserting and retrieving data. This code is available on github: **https://github.com/uit-inf-3200.**

Backend Nodes

Upon receiving a request from the frontend node, the backend node is responsible for completing the requested operation. This likely involves contacting other nodes to handle the request.

Key Requirements:

1. Distribute data storage load between storage nodes according to a chosen architecture.

2. Support a minimum of three backend nodes.

3. Store the data in-memory.

4. Implement a monitoring tool that outputs/visualizes how much data each backend node stores.

API

The frontend allows client applications to issue PUT and GET requests to store and retrieve data using the HTTP protocol. You are free to translate this interface to something different when forwarding requests to the backend nodes.

- **PUT**: Store the message body at the specific URI (key). PUT requests issued with existing keys should overwrite the stored data.
- **GET**: Return the data stored under the URI.

Hand in

In your report, briefly present the details of your approach to the assignment. Then discuss the details that are interesting to you. We want to see that you can assess the strengths and weaknesses of your implementation. Think outside of the context of the rocks cluster. How would it behave under different workloads such as more users or a higher number of requests per second?

The delivery should include:

1. Source code (programming language of your choice) with instructions on how to run.
2. Report

Other Practical Details

- You are free to use any language supported by the cluster.
- Groups of two is preferred, but not required.
- There will be a demo presentation in connection with the hand in. This will occur during the colloquium following the deadline. You are expected to briefly present your work and demonstrate it.
- The hand in will be on github. You will receive further information on how to do so.
- Deadline is **Monday, October 5th.**

References

[1] Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.* 31, 4 (August 2001), 149-160. DOI=10.1145/964723.383071 http://doi.acm.org/10.1145/964723.383071

[2] Windows Azure Storage: a highly available cloud storage service with strong consistency. In*Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (SOSP '11). ACM, New York, NY, USA, 143-157. DOI=10.1145/2043556.2043571 http://doi.acm.org/10.1145/2043556.2043571

[3] Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (SOSP '07). ACM, New York, NY, USA, 205-220. DOI=10.1145/1294261.1294281 http://doi.acm.org/10.1145/1294261.1294281