INTRODUCTION

The aim of this project is gain practical insight into building a distributed key-value store.

The key concepts this project touches upon are the following:

- Basic client-server communication through sockets
- Concept of cluster swith one master and multiple slaves
- Implementing Zookeeper-cluster coordination
- Implementing replication of data across servers (with one server acting as the primary data store and another acting as the secondary data store)
- Demonstrating high availability, or in other words, reliability

A key-value store is capable of storing data indexed by a key. Keys are strings of characters, whereas values can be any JSON object.

RELATED WORK

In addition to the document on the class project shared on Google Drive, there were a couple of useful sources of information that went a long way towards understanding how to implement the key-value store.

One main source is Hadoop's DFS and it's working. Understanding this is a step in the right direction. Our implementation of the key-value store is a barebones version of actual distributed data stores in practice.

SYSTEM DESIGN

The basic flow of the system is as follows:

1. Cluster is initialized as follows:
   - N servers processes are created and register with Zookeeper
   - 1 server becomes the master and updates the Zookeeper directory
   - [N-1] servers acknowledge the presence of the master and register as slaves under the master
2. Master updates the cluster status to READY and sends the primary and secondary key-ranges to slaves
3. All nodes update their mappings in Zookeeper
4. The client gets the address of the master from the Zookeeper directory listing, and then requests the master for the particular server address for some data
5. The master responds with the requested mapping, and the client then sends a PUT/GET request to the server

Program Structure

There are 3 main components to the distributed key-value simulator:

1. *cluster.py*

   This file contains the **Cluster** class, which initializes a process pool of workers. Each worker in the pool is, in fact, a server in the cluster. This class also initializes the Zookeeper handler with some initial state information.

2. *server.py*

   This file contains the **Server** class as well as a number of user defined **Exception** classes. All the core functionality for servers, be it for the master or slaves, is all written here.

3. *client.py*

   This file contains the **Client** class, whose objects are instantiated and then used to communicate with the cluster to carry out GET/PUT operations.

Client-Server Communication

This is very simply done through the use of TCP sockets for reliable data transfer. All nodes in the cluster initialize a listener socket and perpetually listen to requests on that socket until the node is shut down.
All requests and responses are sent in the form of JSON objects.

A request message has the common structure:

```
request = { 'op'      : <OP code>,
            'data'    : <JSON or String>
          }
```

- **op** refers to the type of operation, and indicates to the receiver, the action that is to be performed.
- **data** refers to any supplementary data that is sent along with the operation code.

A response message has the following structure:

```
response = {'status'   : <status>,
            'message'  : <optional message parameter>
            }
```

- **status** indicates whether the request was successful or caused an error, similar to HTTP status codes. List of all implemented status codes are:
  - **200 OK**
  - **400 Bad Request**
  - **404 Not Found**
  - **500 Internal Server Error**
- **message** is an optional parameter sent to indicate what went wrong in the event of an error

A client can make only two types of requests to nodes in the cluster:

1. **GET**
2. **PUT**

Servers have an additional number of operation codes that are required during cluster initialization, such as **__reg__** (to register with the master) and **__config__** (to exchange information on key-value mappings).

All the servers run a listening socket on a separate thread of control where they listen to requests. If a server wants to send a request, it will create a new socket and then send it through that.
Servers also run yet another thread of control in order to carry out periodic backups of data.

Working with Zookeeper

Zookeeper's primary purpose is to provide protected access to shared information.
Zookeeper's main functions are:

1. Maintain a list of servers in the cluster
2. Maintain a list of key-server mappings for the cluster
3. Keep track of the cluster master and status

There are quite a few Python-Zookeeper APIs, but one of the most common and easy-to-use ones is called **nd_service_registry**.
This API allows users to easily create ephmeral (temporary) nodes or data (permanent) nodes, and control access to these nodes using locks.

Implementing high availability

All servers run a separate thread of control, which periodically sends primary key data to the secondary server responsible for that key.
In order to demonstrate high availability, simply kill the process that is running a particular server using kill –KILL <pid> and then attempt a GET operation for data for which the killed process was responsible for.

EXPERIMENTAL RESULTS

The program will simulate the working of a cluster for as many servers as required. However, we see that the timeout parameter (the period after which the master stops accepting registrations) needs to be increased proportional to the size of the cluster.

FUTURE ENHANCEMENTS

A major limitation with the current implementation is that there is no failover mechanism in the event of the master crashing. As a future enhancement, this feature would receive the highest priority.

Implementing a heartbeat mechanism, so that the master can quickly get to know when a slave crashes and take pre-emptive measures to ensure a smooth, seamless experience for the client.

As of now, high availability is guaranteed when only 1 server goes down. In order to make the system more robust, we can assign 3 (or more) different servers to handle the same set of keys.

Another possible enhancement is to allow new servers to join the cluster. As of now, the cluster size is fixed upon cluster initialization, and is not expected to change. Some of the challenges involved with implementing this would be:

- Reshuffling key responsibilities
- Redistributing data over the network (a significant overhead)
- Registration with Zookeeeper, and intimating the master of the arrival of a new slave

REFERENCES

Python socket documentation
https://docs.python.org/2/library/socket.html

How a server cluster works
https://technet.microsoft.com/en-in/library/cc738051(v=ws.10).aspx

Installing and running Zookeeper
https://zookeeper.apache.org/doc/r3.1.2/zookeeperStarted.html

Python-Zookeeper API documentation
https://github.com/Nextdoor/ndserviceregistry

Implementing Hadoop's hash function
http://stackoverflow.com/questions/18470833/partitioning-how-does-hadoop-make-it-use-a-hash-function-what-is-the-default