MPCS 52040 Distributed Systems

Project: Distributed key-value store

Due: 6/3 @ 11:59pm

The aim of this project is to create a distributed key-value data store (KV store). As the name suggests, KV stores implement a simple data storage model in which a collection of keys are mapped to associated values (e.g., name='bob', age='22'). Unlike a database that stores tables with typed columns (using a schema to define that table), a record in a KV store may be comprised of a collection of (typically untyped) key-value pairs.

KV stores have become increasingly common, especially in cloud computing, as they can be elastically scaled. Common examples include Dynamo, Memcached, and Redis. KV stores are desirable for many reasons including the simplicity of the interface and model, the flexibility of the storage system for arbitrary data, and potential for scalability. These same advantages separate KV stores from SQL databases.

The basic interface to a key-value store is

> **PUT** (key, value) -> success/failure: add a key-value pair to the data store. Overwrite previous value if one exists. Return true if the key-value pair was added successfully, or false if it could not be added.
>
> **GET** (key) -> value/error: retrieve the value corresponding to the given key from the KV store. Return an error if the key could not be found or the value could not be returned.

KV stores exemplify many of the topics covered in this course such as distribution, consistency, replication, fault tolerance, and consensus. As we will discuss in lectures, one of the challenges when developing a distributed data store is the need to be available (that the service will stay online in the presence of failures) and consistent (that the service provides the most recent state when read). More information is available here on the challenges involved with maintaining consistency and availability: https://en.wikipedia.org/wiki/CAP_theorem. In order to achieve availability, we generally replicate data, however this of course makes it challenging to provide consistency as different replicas must now agree on their state.

In this project you will implement a distributed KV store using whatever underlying communication approach you like (e.g., ZeroMQ, REST, sockets). As part of this project you will need to ensure that the KV store is distributed across several nodes and is fault tolerant. That is, it should be resilient to node failure (fail-stop). It is up to you how you choose to implement this KV store, for example you may use Raft or Paxos, or another approach (please talk to us first if you decide to take a different approach). Below we provide a rough guideline to completing the project using Raft.

Requirements

The project should be done in Python.

The project may be conducted by teams of up to two students. You are welcome to do the project alone if you like.

You cannot use existing consensus libraries e.g., Zookeeper or an implementation of Raft or Paxos. You also should not build upon an existing distributed KV store (e.g., Redis). If you are in doubt about using a particular library, please ask on Piazza.

Your KV store should be fault tolerant. That is, it should be resistant to failure of one or more nodes (up to N/2 – 1). You can assume that all failures are fail-stop failures. That is, nodes may fail but they will not rejoin the system. You also do not need to address byzantine failures or network partitions.

Irrespective of what consensus model you choose to implement you should provide a JSON-based interface to your KV store. Each instance of your KV store should be able to accept and reply to these messages (in latter stages of the project you might need to provide a new redirect message to tell a client to contact another server). The basic PUT/GET interface should be implemented as follows.

PUT:

Send:

```
{'type': 'put', 'payload': {'key': 'k', 'value': 'v'}}
```

Return

```
{'code': 'success'/'fail'}
```

GET:

Send:

```
{'type': 'get', 'payload': {'key': 'k'}}
```

Returns:

```
{code: 'success'/'fail', `payload': {
      'message': 'optional', 'key': 'k', 'value': 'v'}
}
```

Note: you can expand/extend this interface if required, but please document your interface.

You will also need to add internal messages between services to implement the consensus algorithm.

Report

You should submit a report (5 pages or less) that outlines:

- The design and implementation of your distributed KV store including description of your interface and summary of how PUT and GET requests are handled.
- A summary of how your implementation is fault tolerant, including the fault-tolerant algorithm you have used.
- A description of how you have tested your implementation to ensure that it meets all requirements.
- Please remember to reference all sources used in developing your implementation.

Grading

The rubric for the project is as follows.

- 20 points: Basic KV store implementation. Your implementation should support GET and PUT in a single server without replication and fault tolerance.
- 20 points: Basic safety. Your implementation should provide a way to speak to a leader (or similar) such that we can ensure that we cannot have conflicting operations (although the state of the system need not be replicated).
- 20 points: Basic replication: Your implementation should provide a way to replicate the contents of the KV store across several nodes (although it need not be fault-tolerant).
- 20 points: Fault tolerance: Given a call to GET or PUT, the call must eventually succeed or return an error message. Either call can be delayed in the presence of failures but, in the absence of failures, all calls must eventually succeed. These conditions should be supported for fail-stop failures (i.e., nodes do not recover from failure).
- 20 points: Report outlining your implementation and testing process.

Please include a README file in the root of your repository that includes:

- Names and e-mail addresses of all the students in the group
- A description of what each student worked on
- Concise instructions on how we should run your scripts

Example approach using Raft

Raft is one of the most well-known consensus algorithms designed to provide fault-tolerant agreement amongst a set of distributed servers. Raft is particularly popular as it has been designed to be easy to understand. The basic algorithm is decomposed into several independent tasks (e.g., leader election, log synchronization, state machine updates) and the required state and message format are well documented online.

We suggest you review the Raft website (https://raft.github.io/) and the original Raft paper (https://raft.github.io/raft.pdf). The paper is written with the goal of being understandable and therefore provides the most clear and accessible documentation of the algorithm.

Below we outline one approach to this project. You do not need to follow these steps and they are designed to serve as a fairly high level outline.

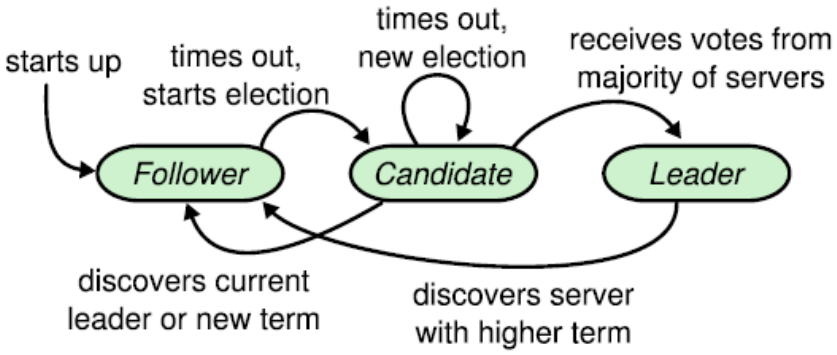
Step 1: Create a centralized KV store.

First you should create a single server (e.g., using ZeroMQ) that implements the PUT and GET operations described above. You will need to define a data structure for storing the KV store (e.g., a Python dictionary) and the functions for PUT/GET of keys in that store. You will need to implement functionality to parse the JSON-based requests and return responses in JSON.

You should then create a client that can test your server by sending workloads to your server. You should make this client submit a number of different requests (e.g., updating keys, putting new keys, getting keys that don't exist, etc.) so that in subsequent steps of the project you can identify consistency issues.

Step 2: Implement the leader election algorithm.

In the second step you should augment your server so that it can be distributed and that your client can always coordinate with a single "leader" of your system. To do so you will need to implement the Raft leader election algorithm. The best source of information for this step is in the Raft paper in Figure 2. It is important to note that this is a fairly precise specification of the algorithm and thus requires that you follow it fairly accurately. The basic lifecycle of the algorithm is depicted in the picture above. You will need to implement these transitions between the various states (follower, candidate, leader).



You will need to add new message types (beyond the aforementioned 'PUT' and 'GET' messages) to your server such that it can exchange information with other deployed servers. You may assume that you can hard code the list of other servers in the system (and client) upon startup.

Note: in the Raft paper it refers to RPC methods. You do not need to implement these methods using RPC, but rather you can map the format to your own messaging format.

Briefly, you will need to do the following things:

1. Implement the Raft election protocol. You will need to implement much of the state described in Figure 2 of the Raft paper, store the state of the server (candidate, follower, leader), and implement the transitions shown in the figure above.
2. Extend your KV store interface such that your server can reply with a redirect message. That is, when the client sends a PUT/GET request to a server that is not the leader you can reply with a redirect message to the appropriate leader.

3. Add the timeout capability to your server such that it can detect leader failure. That is, if you have not heard from the leader in some period of time, you will start a new election process.
4. Add a heartbeat message (Using the AppendEntries message) such that a leader can periodically notify all other servers that it is still alive.
5. Test your server by removing leaders and validating that a new leader takes over.
6. Extend your client so that if given a redirect message it will change the server that it is talking to and repeat the request.

You should now have a safe KV store in which you cannot have conflicting PUT/GET messages. However, the state of your store is not yet replicated between the various services in your system.

Step 3: Implement the distributed replication model

Now you have a leader that is capable of servicing all incoming requests you will need to implement the distributed replication model to enable the leader to distribute updates to other nodes in the network. The basic model, as outlined in the paper, relies on the leader recording each command to a local log (i.e., a list) and then sending messages to other nodes in the network to distribute that log. Note, when distributing the log you will need to share it to a quorum of other servers so as to ensure that there is no potential for inconsistency.

You will need to do the following steps:

1. Implement the state machine for each node (you should already have this as the dictionary containing key-value pairs from step 1).
2. Implement the transaction log and make sure that the leader correctly stores operations in this log.
3. Enhance the AppendEntries message so that it is able to distribute log data to other servers.
4. Implement functionality to make sure that a quorum is in agreement before committing the log changes to the local state machine.

Note

Raft is a relatively complicated algorithm and consensus in general is a fairly complicated area. It will take a lot of testing and debugging to understand how the algorithm is working and to convince yourself (and the graders) that you have the algorithm implemented correctly.

Rough Timeline of Milestones

While none of the assignment is due until the final due date, we provide the following suggested timeline for finishing various components of this project so that your team may remain ‘on track’:

- Milestone 1 (05/16): centralized key-value store – PUT and GET requests functional.
- Milestone 2 (05/23): Distributed key-value store with leader election.
- Milestone 3 (05/30): Distributed replication of log/state.
- Milestone 4 (06/03): Report. Project due.