HW4 spec is up: Scaleable, Fault-Tolerant Key Value Store

Due : Wednesday, December 6th, 11:59pm

The goal of the final homework is to develop a distributed key-value store that can store lots of data: At least in principle, more data than can fit into a single machine. In this homework, you will be building a key-value store that is scalable as well as fault tolerant. That is to say, your KVS can accommodate increasing data sizes by adding more nodes.

The key to scalability is dividing the work of storing keys across many machines -- that is, partitioning the keyspace. How can we partition a key-value store across several machines? There are many strategies available for assigning keys to nodes. For example, one can use hashing, random placement, consistent hashing, a directory-based approach or round robin partitioning strategies. Every strategy has its own advantages and disadvantages, as we discussed in class. You can create you own partition strategy or you can implement an existing one. A partition strategy should satisfy the following 2 properties:

1. Every key belongs to a one partition, consisting of some number of replicas. (explained in the next section).
2. Keys are distributed (approximately) uniformly across the partitions.

In this assignment, you need to develop a key-value store with a partitioning strategy that satisfies the above 2 conditions. Moreover, the key-value store should be resizable: we can add and remove nodes from the cluster while the key-value store is running. Therefore the partition strategy should be dynamic, that is, after the number of nodes have changed, the key-value store should rebalance keys between partitions.

Starting the Key Value Store

To start a key value store we use the following environmental variables.

- "K" is the number of replicas per partition. Each partition owns a subset of keys.
- "VIEW" is the list of ip:ports pairs of nodes.
- "IPPORT" is the ip address and port of the nodes

An example of starting a key-value store with 4 nodes and partition size 2:

```
docker run -p 8081:8080 --ip=10.0.0.21 --net=mynet -e K=2 -e
VIEW="10.0.0.21:8080,10.0.0.22:8080,10.0.0.23:8080,10.0.0.24:8080" -e
IPPORT="10.0.0.21:8080" mycontainer
```

```
docker run -p 8082:8080 --ip=10.0.0.22 --net=mynet -e K=2 -e
VIEW="10.0.0.21:8080,10.0.0.22:8080,10.0.0.23:8080,10.0.0.24:8080" -e
IPPORT="10.0.0.22:8080" mycontainer
docker run -p 8083:8080 --ip=10.0.0.23 --net=mynet -e K=2 -e
VIEW="10.0.0.21:8080,10.0.0.22:8080,10.0.0.23:8080,10.0.0.24:8080" -e
IPPORT="10.0.0.23:8080" mycontainer
docker run -p 8084:8080 --ip=10.0.0.24 --net=mynet -e K=2 -e
VIEW="10.0.0.21:8080,10.0.0.22:8080,10.0.0.23:80,10.0.0.24:8080" -e
IPPORT="10.0.0.23:8080" mycontainer
```

As mentioned above, the environmental variable "K" is the number of replicas per partition. What this means is that if the number of nodes in your initial view is 4 and K=2, you have two partitions of two nodes each. Each of these partitions should be responsible for approximately half the keys and the keys must be replicated within the partition.

As an example, you may decide that all the keys starting with the letters "a" through "m" should go to partition 1 and those starting with the letters "n" through "z" should go to partition 2. The keys which are stored in partition 1 should not be stored in partition 2 and vice versa. All keys stored on a single node in a partition should be replicated on all other nodes in the partition.

Let us consider a few more scenarios:

1. N=7, K=2: Three partition sets of two nodes each, with each partition set responsible for approximately a third of the keys. One node behaves as proxy.
2. N=6, K=3: Two partition sets of 3 nodes each. Each set is responsible for approximately half the keys.

While the number of nodes in the system can increase or decrease, the value of K remains constant. However, a change in the number of nodes impacts the number of partitions.

Key-value Operations:

Adding and Deleting Nodes:

Your key-value store needs to be resizable. We can use environmental variables only when we start the nodes. Once all nodes are running, we need another mechanism to notify existing nodes about cluster changes. Therefore your key-value store needs to support an API which notifies

current nodes about a "view change" that could involve the addition of a new member or loss of an old one.

Use the following API:

When a node receives an update_view request, it is responsible for notifying all of the other nodes of the view change and for moving its own keys to where they belong in the new view. Once the view change is successful on all nodes, the node who received the original update_view request then sends the client the "success" response. We guarantee that we will wait until view changes are complete before sending any more traffic to a given node.

An update_view request always describes the addition or removal of a single node. Note that such an operation might change the number of partitions. For example, say we started a key value-store with 6 nodes and partition size K=3. It follows that the key-value store should have 2 partitions with 3 replicas each, because we should aim to have the largest possible number of partitions while maintaining the value of K in each. Each partition should be responsible for approximately half the keys. Now suppose a node was added into the system. We do not want to have partitions which have fewer than “K” replicas in them. Therefore, we designate the new node as proxy rather than as a replica. If two more nodes are added, we now have enough nodes to create a third partition while maintaining K. A new partition should be created and each should now be responsible for roughly a third of the keys.

Now, consider another system. If N=6, K=2, we should have three partitions. If one node is removed, that partition can no longer operate, since the required replication factor (K) has not been maintained. Therefore, we must now drop to 2 partitions with 1 node behaving as proxy.

Below are some examples of update_view requests:

- A PUT request on "/kv-store/update_view?type=add" with the data payload "ip_port=<ip_port>" adds the node to the key-value store. It increments the number of partitions, if needed. A successful response returns the partition id of the new node, and the total number of partitions. It should look like:

```
{
    "result":"success",
    "partition_id": 2,
    "number_of_partitions":3,
}
```

- A PUT request on "/kv-store/update_view?type=remove" with the payload "ip_port=<ip_port>" removes the node. It decrements the number of partitions, if needed. A successful response object contains the total number of partitions after the deletion, for example:

```
{
     "result":"success",
     "number_of_partitions": 2
 }
```

Obtaining Partition Information:

The following methods allow a client to obtain information about partitions.

- A GET request on "/kv-store/get_partition_id" returns the partition id where the node belongs to. For example, the following curl request curl -X GET http://localhost:8083/kv-store/get_partition_id returns the id of the node that we can access via localhost:8083. A successful response looks like:

```
{
    "result":"success",
    "partition_id": 3,
}
```

- A GET request on "/kv-store/get_all_partition_ids" returns a list with ids of all partitions.

```
{
     "result":"success",
     "partition_id_list": [0,1,2,3]
 }
```

- A GET request on "/kv-store/get_partition_members" with data payload "partition_id=<partition_id>" returns a list of nodes in the partition. For example the following curl request curl -X GET http://localhost:8083/kv-store/get_partition_members -d 'partition_id=1' will return a list of nodes in the partition with id 1.

```
{
     "result":"success",
     "partition_members": ["10.0.0.21:8080", "10.0.0.22:8080",
"10.0.0.23:8080"]
 }
```

GET and PUT requests remain the same as the previous assignment. The only difference is that you will be returning a “partition_id” instead of “node_id” in the response JSON.

- A GET request to "/kv-store/<key>" with the data field "causal_payload=<causal_payload>" retrieves the value that corresponds to the key. The "causal_payload" data field is the causal payload observed by the client's most recent read or write operation. A response object has the following fields: "msg", "value", "partition_id", "causal_payload", "timestamp". A response to a successful request looks like:

```
{
     "result":"success",
     "value":1,
     "partition_id": 3,
     "causal_payload": "1.0.0.4",
     "timestamp": "1256953732"
 }
```

- A PUT request to "/kv-store/<key>" with the data fields "val=<value>" and "causal_payload=<causal_payload>" creates a record in the key value store. The "causal_payload" data field is the causal payload observed by the client's most recent read or write operation (why we need it? See the example above). If the client did not do any reads, then the causal payload is an empty string. The response object has the following fields: "msg", "partition_id", "causal_payload", "timestamp". An example of a successful response looks like:

```
{
     "result":"success",
     "partition_id": 3,
     "causal_payload": "1.0.0.4",
     "timestamp": "1256953732"
 }
```

- DELETE. You do not need to implement deletion of keys in this assignment.

Error Responses:

All error responses contain 2 fields "result" and "error". The error field contains the details about the error, for example:

```
{
    "result":"error",
    "error":"key value store is not available"
}
```

Functional Guarantees:

We will evaluate your key-value store using the following criteria:

1. Operations "put" and "get" should behave as expected on a single site key-value store .
2. Every key should be present in one partition only.
3. Keys should be distributed (approximately) evenly across the partitions.. You can assume that keys are sampled uniformly at random. In other words, the probability of observing any key is the same. (We will test this by doing a large number of PUTs, and then observing their distribution via their returned partition_id on GETs)
4. When a new node is added or removed, the keys should be re-distributed across the nodes so the above properties hold.
5. After a view change, the nodes should be able to accept put and get requests as before.
6. This assignment builds upon assignment 3. In addition to the requirements of this assignment, your KVS must uphold the fault tolerance and consistency guarantees of assignment 3 . That is to say, fault events including network partitions and crashes should not cause data loss. Additionally, the causal ordering and bounded staleness constraints required by assignment 3 should be upheld.

When you complete this assignment, you will have implemented a scalable, fault-tolerant key value store (!), which you have certainly used and may even build as you go forward in your careers! It’s something to be proud of, and no easy task. :)