CMPS128: Distributed Systems

Assignment #4

Spring 2019

Instructions

General

In the previous assignment, you implemented a fault-tolerant key-value store using replication. To put it another way, you added more nodes to make your key-value store more resilient. However, resilience is not the only reason to take on the complexity of distributed systems. By adding more nodes to your key-value store and distributing key-value pairs across nodes in an intelligent way, you can increase the capacity and throughput of the key-value store. We call such a key-value store a **sharded key-value store**. In this assignment, you will extend the implementation of your fault-tolerant key-value store and add sharding to it. Your implementation will be a **sharded, fault-tolerant key-value store**, with better fault tolerance, capacity, and throughput than a single-site key-value store can offer.

- You will use **Docker** to create an image implementing the distributed key-value store described in the next section.
- You must do your work as part of a team.
- You need to implement your own key-value store, and not use an existing key-value store such as Redis, CouchDB, MongoDB, etc.
- We will only grade the most recently submitted commit ID for each repository, so it is OK to submit more than once.
- The assignment is due **06/07/2019 (Friday) 11:59 PM**. Late submissions are accepted, with a 10% penalty per day of lateness. Submitting during the first 24 hours after the deadline counts as one day late; 24-48 hours after the deadline counts as two days late; and so on.
- You may consider the *order* of name/value pairs in JSON objects to be irrelevant. For example, `{"foo":"bar","baz":"quux"}` is equivalent to `{"baz":"quux","foo":"bar"}`.

Testing

- We have provided a short test script `test_assignment4.py` that you can use to test your work. It is critical that you run the test script before submitting your assignment. The tests provided are the same ones we will run on our side during grading. We may also run additional tests consistent with the assignment specification.

Submission workflow

- A private repository named `CMPS128_Assignment4` should be created by one of the members of a team.

- The GitHub accounts of the other members of the team as well as `ucsc-cmps128-staff` should be added as collaborators to the repository.
- The repository should contain:
  - the project file(s) implementing the key-value store
  - the `Dockerfile` instructing how to create your Docker image
  - a file `members.txt` listing the members of the team
  - a file `contribution-notes.txt` describing the contributions of each member of the team
  - a file `mechanism-description.txt` including the description of the mechanisms implemented for sharding, causal dependency tracking, and detecting that a replica is down
- Your team name, repository URL, and commit id (you would like to be used for grading) should be submitted through the following Google form: https://forms.gle/k3qH1CjBkD2v6cJJ7
- Only one of the team members needs to submit the form.

Sharded Fault-Tolerant Key-Value Store with Causal Consistency

Your sharded fault-tolerant key-value store supports three kinds of operations: **view operations**, **shard operations**, and **key-value operations**. The main endpoints for these operations are `/key-value-store-view`, `/key-value-store-shard`, and `/key-value-store`, respectively.

The term **view** refers to the current set of nodes that are up and running. To do view operations, a **node** sends GET (for retrieving the view), PUT (for adding a new node to the view), and DELETE (for deleting a node from the view) requests to another node.

A **shard** is a group of nodes that store a **particular subset** of key-value pairs. We also refer to a shard as a **replica group**. Each shard has a unique ID, `shard-id`, and a list of nodes belonging to the shard, `shard-members` . To do shard operations, **a client or node** sends GET (for retrieving information about shards) and PUT (for changing number of shards or changing the membership of nodes in the shards) requests to a node in the store.

To do key-value operations on key `<key>`, a **client** sends GET (for retrieving the value of key `<key>`), PUT (for adding the new key `<key>` or updating the value of the existing key `<key>`), and DELETE (for deleting key `<key>`) requests to the `/key-value-store/<key>` endpoint at a replica. The store returns a response in JSON format as well as the appropriate HTTP status code, as described in the spec of the assignment.

Assume a scenario in which we have six nodes named `node1`, `node2`, `node3`, `node4`, `node5`, and `node6`, each running inside a Docker container (instance) connected to a subnet named `mynet` with IP address range `10.10.0.0/16`. The IP addresses of the replicas are `10.10.0.2`, `10.10.0.3`, `10.10.0.4`, `10.10.0.5`, `10.10.0.6`, and `10.10.0.7`, respectively. All instances are exposed at port number 8080. We might have three shards, in which each shard contains two nodes; or two shards, each containing three nodes; or two shards in which the first one contains four nodes and the second one has two nodes. In this assignment, we assume that each node belongs exactly to one shard for the sake of simplicity.

Each node knows its own socket address (IP address and port number), the view of the store (socket addresses of all nodes), and the number of shards to begin with. You should give this external information to the node when you start the corresponding container. The following shows how you can start your key-value store implemented using Docker. We will assume that the key-value store has two shards.

Create subnet

To create the subnet `mynet` with IP range `10.10.0.0/16`, execute

```
$ docker network create --subnet=10.10.0.0/16 mynet
```

Build Docker image

Execute the following command to build your Docker image:

```
$ docker build -t assignment4-image .
```

Run Docker containers

To run the replicas, execute

```
$ docker run -p 8082:8080 --net=mynet --ip=10.10.0.2 --name="node1" -e SOCKET_ADDRESS="10.10.0.2:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image

$ docker run -p 8083:8080 --net=mynet --ip=10.10.0.3 --name="node2" -e SOCKET_ADDRESS="10.10.0.3:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image

$ docker run -p 8084:8080 --net=mynet --ip=10.10.0.4 --name="node3" -e SOCKET_ADDRESS="10.10.0.4:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image

$ docker run -p 8085:8080 --net=mynet --ip=10.10.0.5 --name="node4" -e SOCKET_ADDRESS="10.10.0.5:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image

$ docker run -p 8086:8080 --net=mynet --ip=10.10.0.6 --name="node5" -e SOCKET_ADDRESS="10.10.0.6:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image

$ docker run -p 8087:8080 --net=mynet --ip=10.10.0.7 --name="node6" -e SOCKET_ADDRESS="10.10.0.7:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080"
  -e SHARD_COUNT="2" assignment4-image
```

Shard Operations

Nodes are organized into shards based on the initial number of shards given by the environment variable `SHARD_COUNT`. You are free to choose any mechanism to create the shards. **The only requirement is that each shard must contain at least two nodes to provide fault tolerance.** As an example, for a key-value store with 6 nodes and shard count of 2, your implemented mechanism might put `node1`, `node2`, and `node3` into one shard and `node4`, `node5`, and `node6` into the other shard. Or `node1`, `node2`, `node3`, and `node4` can be the members of the first shard and `node5`, `node6` can be in the second shard. You need to provide the description of the mechanism you implemented in the `mechanism-description.txt` file.

Here are the shard operations your key-value store should support:

Get the shard IDs of the store

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://<node-socket-address>/key-value-store-shard/shard-ids

  {"message":"Shard IDs retrieved successfully","shard-ids":<shard-ids>}
```

```
200
```

Example: For the scenario with 6 nodes and 2 shards, we might have

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/shard-ids

  {"message":"Shard IDs retrieved successfully","shard-ids":"1,2"}
  200
```

We used integer shard IDs in the example, but you don't have to use integers for shard IDs. They can be any unique values.

Get the shard ID of a node

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://<node-socket-address>/key-value-store-shard/node-shard-id

  {"message":"Shard ID of the node retrieved successfully","shard-id":<shard-id>}
  200
```

Example: For the scenario with 6 nodes and 2 shards, we might have

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/node-shard-id

  {"message":"Shard ID of the node retrieved successfully","shard-id": 1}
  200
```

or

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/node-shard-id

  {"message":"Shard ID of the node retrieved successfully","shard-id": 2}
  200
```

depending on which shard `node1` belongs to.

Get the members of a shard ID

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://<node-socket-address>/key-value-store-shard/shard-id-members/<shard-id>

  {"message":"Members of shard ID retrieved successfully","shard-id-members":<shard-id-members>}
  200
```

Example: For the scenario with 6 nodes and 2 shards, in which `node1`, `node2`, `node3`, and `node4` are the members of the shard with ID `1` and `node5`, `node6` belong to shard ID `2`, we have

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/shard-id-members/1
```

```
  {"message":"Members of shard ID retrieved successfully",
   "shard-id-members":"10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080"}
  200

$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/shard-id-members/2

  {"message":"Members of shard ID retrieved successfully",
   "shard-id-members":"10.10.0.6:8080,10.10.0.7:8080"}
  200
```

Get the number of keys stored in a shard

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://<node-socket-address>/key-value-store-shard/shard-id-key-count/<shard-id>

  {"message":"Key count of shard ID retrieved successfully","shard-id-key-count":<shard-id-key-count>}
  200
```

Example: For the scenario with 6 nodes and 2 shards, in which shard `1` contains 4000 keys and shard2 contains 3500 keys, we have

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://<node-socket-address>/key-value-store-shard/shard-id-key-count/1

  {"message":"Key count of shard ID retrieved successfully","shard-id-key-count":4000}
  200

$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"

  http://10.10.0.2:8082/key-value-store-shard/shard-id-key-count/2

  {"message":"Key count of shard ID retrieved successfully","shard-id-key-count":3500}
  200
```

Add a node to a shard

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"

  --data '{"socket-address": <new-node-socket-address>}'

  http://<node-socket-address>/key-value-store-shard/add-member/<shard-id>
```

For instance, to add the new node `node7` with socket address `10.10.0.8:8082` to shard `2`, we have

```
$ docker run -p 8088:8080 --net=mynet --ip=10.10.0.8 --name="node7" -e SOCKET_ADDRESS="10.10.0.8:8080"
  -e VIEW="10.10.0.2:8080,10.10.0.3:8080,10.10.0.4:8080,10.10.0.5:8080,10.10.0.6:8080,10.10.0.7:8080,
  10.10.0.8:8080" assignment4-image

$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"
```

```
  --data '{"socket-address": "10.10.0.8:8082"}'

  http://10.10.0.2:8082/key-value-store-shard/add-member/2
```

Note: To add a new node to the store, we start the corresponding Docker container **without** the `SHARD_COUNT` environment variable. Afterwards, we send a PUT request to explicitly add the new node to a particular shard.

Resharding the key-value store

We might find out that a shard contains only one node due to the failure of the other nodes in the shard, and as a result, it is not fault-tolerant anymore. Or we might add new nodes to the store in order to increase its capacity or throughput. In these cases, we need to reshard the key-value store.

To reshard, a PUT request is sent to endpoint `/key-value-store-shard/reshard` at a node in the store.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"

  --data '{"shard-count": <shard-count> }' http://<node-socket-address>/key-value-store-shard/reshard
```

The node receiving the request should initiate resharding of the entire key-value store, and keys should be redistributed across new shards. You can choose any resharding mechanism as long as it meets the requirement of having at least two nodes in each shard. You must give a description of your resharding mechanism in `mechanism-description.txt` file.

Consider a scenario with 6 nodes and 2 shards, in which `node1`, `node2`, `node3`, and `node4` are the members of shard `1` and `node5` and `node6` belong to shard `2`. `node5` fails and the administrator learns of the failure and sends a reshard request to `node1` with shard count of 2.

If resharding is successful, we have

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"

  --data '{"shard-count": 2 }' http://10.10.0.2:8082/key-value-store-shard/reshard

  {"message":"Resharding done successfully"}
  200
```

If the adminstrator sends a resharding request with shard count of 3, the node should respond with an error message and status code 400 because, since only 5 nodes are up, it is not possible to have 3 shards with at least 2 nodes in each shard.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"

  --data '{"shard-count": 3 }' http://10.10.0.2:8082/key-value-store-shard/reshard

  {"message":"Not enough nodes to provide fault-tolerance with the given shard count!"}
  400
```

Resharding may also change the total number of shards, as long as the property is maintained that there are at least 2 nodes in each shard. **When the total number of shards changes, the resharding process must ensure that keys are more or less evenly distributed across the new set of shards. This is known as *rebalancing*.**

Note: Resharding is initiated whenever the administrator (from a client) sends a resharding request to a node. That is, nodes do **not** automatically initiate resharding on their own when the number of nodes in a shard decreases to 1 or new nodes are added to the store.

Note: Your key-value store does **NOT** need to provide causal consistency for shard operations.

View Operations

Your key-value store must support the view operations specified in Assignment 3. There is no change to the view operations for this assignment.

Key-Value Operations

A client sends GET, PUT, and DELETE operations on key `<key>` to a node in the sharded fault-tolerant key-value store. The store needs a strategy to assign `<key>` to a shard in the store. There are many key-to-shard mapping (key partitioning) strategies available. For example, in class we discussed **partitioning by hash of key** and **consistent hashing**. You are free to create your own strategy or implement an existing one provided that it satisfies the following properties:

- Each key belongs to exactly one shard.
- Keys are (more or less) evenly distributed across the shards.
- Any node should be able to determine what shard a key belongs to (without having to query every shard for it).

Partitioning by hash of key and **consistent hashing** both ensure all these properties. As discussed in class, consistent hashing has the additional advantage that when a shard is added or removed, the number of keys that must be moved around is minimal.

You need to provide a description of the key-to-shard mapping strategy you implement in the `mechanism-description.txt` file.

As in Assignment 3, your key-value store must support **causal consistency** for key-value operations, which include PUT/DELETE and GET operations.

PUT/DELETE

Whenever a client sends a PUT/DELETE request for key `<key>` to a node in the store, the node first determines the shard ID `<shard-id>` of `<key>` using the key-to-shard mapping strategy it employs. If `<shard-id>` is the shard ID of the node itself, it first applies the operation, then replies back to the cleint, and finally replicates the operation among the other members of `<shard-id>` in a causally consistent manner.

If the node does not belong to `<shard-id>`, it forwards the PUT/DELETE request to one of the members of `<shard-id>`. The member receiving the forwarded request recomputes the shard ID of `<key>` to make sure that the request has been correctly forwarded to it. If so, it applies the PUT/DELETE operation and responds to the forwarding node, which in turn replies back to the client. Afterwards, it replicates the operation among the other members of `<shard-id>` so that they can apply the operation in a way that respects causal consistency.

The response message contains a message and status code indicating the success of the operation, the version generated for the operation, the causal metadata to be used in the next request from the client, and the shard ID corresponding to the key.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"
  --data '{"value": "<value>", "causal-metadata": <this-operation-causal-metadata>}'
  http://<node-socket-address>/key-value-store/<key>

  {"message":"Added successfully", "version": "<version>",
   "causal-metadata": "<next-operation-causal-metadata>", "shard-id": <shard-id>}
  201

$ curl --request PUT --header "Content-Type: application/json" --write-out "%{http_code}\n"
  --data '{"value": "<value>", "causal-metadata": <this-operation-causal-metadata>}'
```

```
  http://<node-socket-address>/key-value-store/<key>

  {"message":"Updated successfully", "version": "<version>",
   "causal-metadata": "<next-operation-causal-metadata>", "shard-id": <shard-id>}
  200

$ curl --request DELETE --header "Content-Type: application/json" --write-out "%{http_code}\n"
  --data '{"causal-metadata": <this-operation-causal-metadata>}'
  http://<node-socket-address>/key-value-store/<key>

  {"message":"Deleted successfully", "version": "<version>",
   "causal-metadata": "<next-operation-causal-metadata>", "shard-id": <shard-id>}
  200
```

GET

If a node receives a GET request for key `<key>`, it must first find out the shard ID `<shard-id>` corresponding to `<key>`. If it is the same as its shard ID, it responds to the client. Otherwise, it forwards the GET request to one of the members of `<shard-id>`. The node receiving the forwarded GET request recalculates the shard ID of `<key>` to ensure that it has received the correct request. Afterwards, it responds to the forwarding node, which in turn replies back to the client.

```
$ curl --request GET --header "Content-Type: application/json" --write-out "%{http_code}\n"
  http://<node-socket-address>/key-value-store/<key>

  {"message":"Retrieved successfully", "version": "<version>",
  "causal-metadata": "<next-request-causal-metadata>", "value": "<value>"}
  200
```