CSE 138: Distributed Systems

Fall 2019 - Assignment #3

Assigned: Wednesday, 10/30/19
Due: Friday, 11/08/19

Instructions

Overview

In this assignment, your team will develop a distributed key-value store that stores data (key-value pairs) on more than one machine. In the previous assignment, you were able to query the data from multiple machines, but the data were only stored on the main instance. Sometimes, we come up to use cases that motivate us to split our data, our keys, across multiple machines. In this assignment, we will call the partitions that we separate keys into as **shards**, where the keys in a particular shard are stored on a particular key-value store instance, or **node**.

```
+----------+              +----------+
|  shard1  |              |  shard2  |
|  (node1) |              |  (node2) |
|  ------  |              |  ------  |
|  - key1  |              |  - key2  |
|  - key3  |              |  - key4  |
|  - key5  |              |          |
+----------+              +----------+
```

A key-value store that distributes keys across two nodes, as in the ascii diagram above, must partition the keys into two shards, where each shard is then stored on a single node. To tie in some more descriptive terms, this scenario has two **storage nodes**, whereas the previous assignment had only one storage node–the main instance.

There are many strategies available for partitioning keys into shards, but we expect your key-value store to partition a roughly equal amount of key-value pairs to each shard. For a two node key-value store, this means that each node should store roughly half of the keys, because each shard is stored by a single node. You are free to choose any mechanism to partition keys across nodes.

If a node is queried (receives a request) for a key, the node must first find the node that stores the requested key–a variety of these mechanisms are discussed in lecture. If the queried node is the node that stores the key, then it is able to directly handle the request and respond to the client (service the request). If the queried node does not store the key, then it may service the request by forwarding the request to the correct storage node and forwarding the response back to the client, as in the previous assignment. This is demonstrated in the ascii diagram below, which should look quite similar to the ascii diagram of the previous assignment illustrating the overall behavior.

```
+--------+                             +-------+
| client |                             | Node1 |
+--------+                             +-------+
    |                                     |
    |                                     | ---
    | ---------------request----------->  |  ^
    |                                     |  | node1
    |                                     |  | has
    |                                     |  | key
    | <--------------response----------   |  v
    |                                     | ---
    |             +---------+             |
    |             |  Node2  |             |
    |             +---------+             |
    |                  |                  | ---
    | --- request -->  |                  |  ^
    |                  | --  request -->  |  | node1
    |                  |                  |  | has
    |                  | <-- response --  |  | key
    | <-- response --  |                  |  v
    |                  |                  | ---
    v                  v                  v
```

The process of adding a node to the key-value store dynamically (while it is running) is called a **view change** and allows us to increase capacity or throughput of the key-value store. To execute a view change, the key-value store must re-partition keys into new shards (one additional shard is used for the additional node), and then re-distribute each key-value pair to the correct storage node. Re-partitioning keys is also known as a **reshard**. When a view change request is received, the receiving node should notify the other nodes of the new view and initiate the reshard of the entire key-value store.

Requirements

In this assignment, you will extend your multi-site key-value store by distributing the storage of key-value pairs across many (more than one) instances (nodes).

- You must do your work as part of a team. Your team may not use an existing key-value store (e.g. Redis, MongoDB, etc.)
- You will use **Docker** to create an image that runs a key-value store node and listens to (receives requests from) port 13800.
- Your team's **distributed key-value store** must:
  - Run as a collection of communicating instances, or nodes, where each node is capable of handling a request and responding to the client.
  - Partition key-value pairs into disjoint subsets, or shards.
    - Each key belongs to exactly one shard
    - Each shard is stored by exactly one node
    - You are free to choose any mechanism to partition key-value pairs across nodes
  - The size of each shard (the count of key-value pairs in the shard) should remain relatively balanced (equal in size, within reason) throughout the life of the key-value store.
  - Be able to add new nodes to the key-value store and, accordingly, re-partition keys into shards, or **reshard**, when requested. The key-value store **does not** reshard automatically.

- Each node of your team's distributed key-value store:
  - Must be able to insert, update, get, and delete key-value pairs
  - Does not need to persist data; storing keys only in memory is okay.
  - Must be able to get the number of key-value pairs that is stored
  - Will be provided with:
    - Its own IP address and port number
    - The addresses of all other nodes of the key-value store

Building and testing your container

- We will provide test script(s) that you **should** use to test your work.
- The provided tests are similar to the tests we will use to evaluate your submitted assignment.
- For all JSON responses, whitespace does not have to match expected outputs exactly as we will make comparisons based on structure. However, be sure that string contents do match exactly.

Submission workflow

- Create a new private repository, or reuse the previous private repository, for the team.
- Add the appropriate Github accounts (team members and `ucsc-cse138-staff`) to your private repository, if necessary.
- The repository should contain:
  - The `Dockerfile` defining how to build your Docker image.
  - The project file(s) implementing the key-value store.
  - A file, `member-contributions.tsv`, describing contributions of each team member.
  - A file, `mechanism-description.txt`, describing the mechanisms implemented for partitioning keys across nodes.
- Submit your team name, repository URL, and the commit ID (aka commit hash) to be evaluated here: https://forms.gle/zQJBoHQkaFztqjcd7
  - The commit timestamp **must be no later than 11/08/2019 11:59 PM PDT**
  - The google form must be submitted within a reasonable time of the due date (preferably 10 minutes).
  - Late submissions are accepted, with a 10% penalty for every 24 hours after the above deadline.

Command-line Examples

Previous specs have included basic docker and bash examples, but to keep this specification concise we are moving such examples to separate files to be released with this specification. Please see those files for basic, illustrative examples of what is described here.

API

Endpoints

All endpoints will accept JSON content type ("Content-Type: application/json"), and will respond in JSON format and with the appropriate HTTP status code.

Endpoint URI	accepted request types
`/kv-store/keys/<key>`	GET, PUT, DELETE
`/kv-store/key-count`	GET
`/kv-store/view-change`	PUT

Administrative Operations

GET key count for a shard

- To get the number of keys stored by a node, send a GET request to the endpoint, `/kv-store/key-count` at any node.
  - On success, the response should have status code 200 and JSON: `{"message":"Key count retrieved successfully","key-count":<key-count>}`.

PUT request for view change

- To change the view, or add a new node to the key-value store, send a PUT request to the endpoint, `/kv-store/view-change`, with a JSON payload containing the list of addresses in the new view. For example, the JSON payload to add `node3`, with IP address `10.10.0.4:13800`, to a view containing `node1` and `node2` would be: `{"view": "10.10.0.2:13800,10.10.0.3:13800,10.10.0.4:13800"}`. A view change requires two operations:
  - Propagating the view update to every node
  - Reshard of the keys (a re-partition of keys across nodes)
  - On success, the response should have status code 200 and JSON:

```
{
    "message": "View change successful",
    "shards" : [
        { "address": "10.10.0.2:13800", "key-count": 5 },
        { "address": "10.10.0.3:13800", "key-count": 3 },
        { "address": "10.10.0.4:13800", "key-count": 6 }
    ]
}
```

    where each element in the “shards” list is a dictionary with two key-value pairs: the “address” key maps to the IP address of a node storing a shard, and the “key-count” key maps to the number of keys that are assigned to that node. For the above example, the node at address “10.10.0.4:13800” has 6 key-value pairs, meaning that after the view-change, 6 of the 14 keys in the key-value store were re-partitioned into the shard stored on `node3`.

Key-Value Operations

For all key-value operations:

- If a node receiving a request acts as a proxy, its response **should include** the address of the correct storage node.
- If a node receiving a request does not act as a proxy (it is the correct storage node for the requested key), its response should **not include** the address of the storage node (should not include its own address).

For the below operation descriptions, it is assumed that `node1` (10.10.0.2:13800) does not store the key, `sampleKey`, and that `node2` (10.10.0.3:13800) does store the key.

Insert new key

- To insert a key named `sampleKey`, send a PUT request to `/kv-store/keys/sampleKey`.
  - If no value is provided for the new key, the key-value store should respond with status code 400 and JSON: `{"error":"Value is missing","message":"Error in PUT"}`.
  - If the value provided for the new key has length greater than 50, the key-value store should respond with status code 400 and JSON: `{"error":"Key is too long","message":"Error in PUT"}`.
  - On success, the key-value store should respond with status code 201 and JSON: `{"message":"Added successfully","replaced":false,"address":"10.10.0.3:13800"}`. This example assumes the receiving node does not have address "10.10.0.3:13800" and it acted as a proxy to the node with that address.

Update existing key

- To update an existing key named `sampleKey`, send a PUT request to `/kv-store/keys/sampleKey`.
  - If no updated value is provided for the key, the key-value store should respond with status code 400 and JSON: `{"error":"Value is missing","message":"Error in PUT"}`
  - The key-value store should respond with status code 200 and JSON: `{"message":"Updated successfully","replaced":true}`. This example assumes the receiving node stores the key, `sampleKey`.

Read an existing key

- To get an existing key named `sampleKey`, send a GET request to `/kv-store/keys/sampleKey`.
  - If the key, `sampleKey`, does not exist, the key-value store should respond with status code 404 and the JSON: `{"doesExist":false,"error":"Key does not exist","message":"Error in GET"}`
  - On success, assuming the current value of `sampleKey` is `sampleValue`, the key-value store should respond with status code 200 and JSON: `{"doesExist":true,"message":"Retrieved successfully","value":"sampleValue"}`

Remove an existing key

- To delete an existing key named `sampleKey`, send a DELETE request to `/kv-store/keys/sampleKey`.
  - If the key, `sampleKey`, does not exist, the key-value store should respond with status code 404 and JSON: `{"doesExist":false,"error":"Key does not exist","message":"Error in DELETE"}`
  - On success, the key-value store should respond with status code 200 and JSON: `{"doesExist":true,"message":"De successfully","address":"10.10.0.3:13800"}`. This example assumes the receiving node does not have address "10.10.0.3:13800" and it acted as a proxy to the node with that address.

Container Options and Environment

- VIEW – a required environment variable that provides the address of each node in the store
- ADDRESS – a required environment variable that provides the address of the node being started
- ip – a required container property that assigns the given IP address to the container
- port – a required container property that binds the given host port to the given container port
- net – a required container property that connects the container to the given network

- name – a convenience property so that we can distinguish between containers using a human readable name