CSE138 (Distributed Systems) Assignment 4

CSE138 Course Staff (Shun Kashiwa, Yan Tong, Lindsey Kuper)

Winter 2024

Sharded Replicated Key-Value Store

- Create a *sharded*, *replicated*, *fault-tolerant* and *causally consistent* **key-value store**.
  - In the previous assignment, you made the key-value store fault-tolerant 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 such that not all nodes have all the keys, 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 key-value store and add sharding to it. Your implementation will be have better fault tolerance, capacity, and throughput than a single-site key-value store can offer.

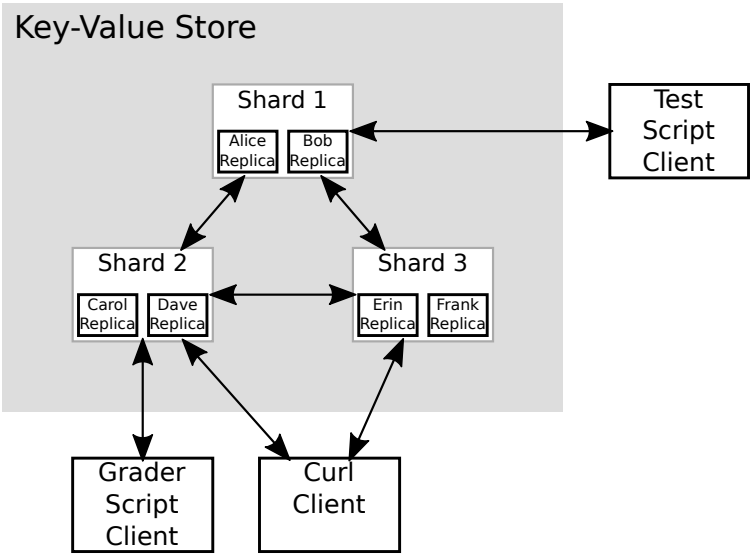

Figure 1: Box-diagram of Sharded Replicated Key-Value Store with clients.

General instructions

- You must do your work as part of a team.
- Due **03/15/2024 (Friday) by 11:59:59 PM**. Late submissions are accepted, with a 10% penalty for each 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.

Submission workflow

1. One of the members of your team should create a **private** GitHub repository named `CSE138_Assignment4`. Add all the members of the team as collaborators to the repository.
2. At the top level of your repository, create a `Dockerfile` to describe how to create your container image.
3. Include a `README.md` file in the top level directory with sections:
   - **Acknowledgements**, **Citations**, and **Team Contributions**: Please refer to the course overview website to learn what needs to go in these sections: https://decomposition.al/CSE138-2024-01/course-overview.html#academic-integrity-on-assignments. *This is a team assignment, and contributions from all the team members are required.*
   - **Mechanism Description**:
     - Describe how your system tracks causal dependencies (can be same as for assignment 3).
     - Describe how your system detects when a node goes down (can be same as for assignment 3).
     - Decide how to shard keys across nodes (new for assignment 4).
4. The team member who created the repository should submit the assignment on Gradescope: https://www.gradescope.com/courses/703589. They should connect their GitHub account to Gradescope, select the repository they created for this assignment, and add the other group members to the submission (see Gradescope's documentation if you need help doing this).

Building and 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. We will run these tests, as well as some additional ones during grading. The tests are *not exhaustive*, and you should do your own testing.

Sharded Replicated Key-Value Store APIs

Your store will support three kinds of operations:

- **View operations** at `/view` manage which nodes (instances or replicas) are part of the key-value store. The view is the list of all the nodes which are up and running.
- **Shard operations** at `/shard` manage which nodes in the view are part of each shard, and how many shards there are. A **shard** is a group of nodes that store a *particular subset* of key-value pairs. Each shard has a unique ID, `shard-id`, and a list of nodes belonging to the shard, `shard-members`.
- **Key-value operations** at `/kvs` perform the useful work of a key-value store.

View operations

Your key-value store must support the same view operations specified in the previous assignment. There is no change to the view operations for this assignment. Recall that there are two required automatic actions:

- If a node finds out another node is down, it deletes the downed node from its view and from the views of the other nodes which are still up.
- When a new node is brought up, it must add itself to the view of each of the existing nodes and retrieve state from somewhere.

As with the previous assignment, nodes will be provided with `VIEW` and `SOCKET_ADDRESS` environment variables on startup.

Shard operations

The store is made up of shards, each of which contain a subset of key-value pairs. A shard is composed of nodes (instances or replicas), each of which contains the same subset of key-value pairs as the other nodes in the shard.

- On initial startup, nodes will be given the environment variable `SHARD_COUNT` and must decide how to organize the view into shards. Your nodes should arrive at the same sharding, whether independently or through communication.
- When a new node is brought up, it is not given `SHARD_COUNT`. Instead, it should join the view as usual but not participate in any shard until the store receives an "add-member" request (described in the next section).
- A reshard request is used to change the shard count, or reassign nodes to shards. Resharding only ever happens in response to a reshard request, never automatically.

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.** For example, in a key-value store with 6 nodes and shard count of 2, your implemented mechanism might choose one of these shardings, which both have at least two nodes per shard.

- 3:3 – Alice, Bob, and Carol in one shard. Dave, Erin, and Frank in another.
- 4:2 – Alice, Bob, Carol, and Dave in one shard. Erin and Frank in another.

There are many other possibilities. Describe the approach you take to divide nodes into shards in the **Mechanism Description** section of your `README.md`.

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

`GET request at /shard/ids`

- Retrieve the list of all shard identifiers unconditionally.
  - Response code is 200 (Ok).

  - Response body is JSON `{"shard-ids": [<ID>, <ID>, ...]}`.
  - Shard identifiers can be strings or numbers and their order is not important. For example, if you have shards `"s1"` and `"s2"` the response body could be `{"shard-ids": ["s2", "s1"]}`.

GET request at /shard/node-shard-id

- Retrieve the shard identifier of the responding node, unconditionally.
  - Response code is 200 (Ok).
  - Response body is JSON `{"node-shard-id": <ID>}`.
  - Here is an example, if the node at `<IP:PORT>` belonged to the shard `"apples"`:

```
$ curl --request GET http://<IP:PORT>/shard/node-shard-id
{"node-shard-id": "apples"}
```

GET request at /shard/members/<ID>

Look up the members of the indicated shard.

- If the shard `<ID>` exists, then return the list of nodes from the view who are in the shard.
  - Response code is 200 (Ok).
  - Response body is JSON `{"shard-members": ["<IP:PORT>", "<IP:PORT>", ...]}`.
- If the shard `<ID>` does not exist, then respond with a 404 error.

GET request at /shard/key-count/<ID>

Look up the number of key-value pairs stored by the indicated shard. To implement this endpoint, it may be necessary to delegate the request to a node in the indicated shard.

- If the shard `<ID>` exists, then return the number of key-value pairs are stored in the shard.
  - Response code is 200 (Ok).
  - Response body is JSON `{"shard-key-count": <INTEGER>}`.
- If the shard `<ID>` does not exist, then respond with a 404 error.

PUT request at /shard/add-member/<ID> with JSON body {"socket-address": <IP:PORT>}

Assign the node `<IP:PORT>` to the shard `<ID>`. The responding node may be different from the node indicated by `<IP:PORT>`.

- If the shard `<ID>` exists and the node `<IP:PORT>` is in the view, assign that node to be a member of that shard.
  - Response code is 200 (Ok).
  - Response body is JSON `{"result": "node added to shard"}`
  - Here is an example, to add a node with socket address `10.10.0.5:8085` to shard `"s4"`:

```
curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n" --data
    '{"socket-address": "10.10.0.5:8085"}' http://localhost:8082/shard/add-member/s4
{"result": "node added to shard"}
200
```

- If either the shard `<ID>` or the node `<IP:PORT>` doesn't exist, respond with a 404 error.
- For other error conditions, respond with a 400 error. This isn't tested.

PUT request at /shard/reshard with JSON body {"shard-count": <INTEGER>}

Trigger a reshard into `<INTEGER>` shards, maintaining fault-tolerance invariant that each shard contains at least two nodes. There's more information in the section Resharding the key-value store, below.

- If the fault-tolerance invariant would be violated by resharding to the new shard count, then return an error.

  - Response code is 400 (Bad Request).
  - Response body is JSON `{"error": "Not enough nodes to provide fault tolerance with requested shard count"}`
- If the fault-tolerance invariant would not be violated, then reshard the store.
  - Response code is 200 (Ok).
  - Response body is JSON `{"result": "resharded"}`

Resharding the key-value store

There are several cases where we need to **reshard**, by reassigning nodes to shards.

- We might find out that a shard contains only one node due to the failure of the other nodes in the shard. In this situation, the shard is not fault-tolerant anymore.
- We might add new nodes to the store in order to increase its capacity or throughput.

To trigger a reshard make a request like this:

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"shard-count":<INTEGER>}' http://<IP:PORT>/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 fault-tolerance invariant. Describe the approach you take to divide nodes into shards in the **Mechanism Description** section of your `README.md`.

Consider a scenario with 6 nodes in a 4:2 sharding: Alice, Bob, Carol, and Dave are in `"shard1"`. Erin and Frank are in `"shard2"`. Erin fails and the administrator learns of the failure and sends a reshard request to Alice with shard count of 2. If resharding is successful, we have

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"shard-count":2}' http://<ALICE>/shard/reshard
{"result": "resharded"}
200
```

If the adminstrator had instead sent a reshard request with shard count of 3, Alice should respond with an error message and status code 400. 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 "\n%{http_code}\n"
    --data '{"shard-count":3}' http://<ALICE>/shard/reshard
{"error": "Not enough nodes to provide fault tolerance with requested shard count"}
400
```

Resharding may also change the total number of shards, as long as the fault-tolerance invariant is maintained. **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*.** See the Key-Value Operations section below for more on the topic of how to assign keys to shards.

Key-Value operations

The supported **key-value operations** are similar to the previous assignment, and must still enforce causal consistency. Differences from the previous assignment are discussed here.

Before handling a key-value operation, a node which receives a request must decide whether the key belongs to that node's shard. 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. With `K` keys spread across `N` shards each shard is responsible for approximately `K/N` keys ±25%. For example, With 2345 keys and 4 shards each shard is responsible for between 440 and 732 keys.
- 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. Describe the approach you take to map keys-to-shards in the **Mechanism Description** section of your `README.md`.

Forwarding requests to shards – Example

Recall the scenario with 6 nodes in a 4:2 sharding: Alice, Bob, Carol, and Dave are in shard `"apples"`. Erin and Frank are in shard `"oranges"`. Alice receives a `GET`, `PUT`, or `DELETE` request to `/kvs/matcha`. First Alice computes the shard ID of the key using her key-to-shard mapping strategy.

Local Key

Alice finds that the key `"matcha"` maps to Alice's shard `"apples"` and so checks the causal metadata of the request to decide whether to reject it (as in the previous assignment). Alice decides to process the request. For write operaitons (PUT or DELETE) Alice broadcasts the operation among the members of shard `"apples"` so that Bob, Carol, and Dave can replicate it in a causally consistent manner. Finally, Alice responds to the request with causal metadata reflecting the operation and also the shard ID to which the key belongs.

If it were a PUT request with value `"ice-cream"` it would look like:

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value":"ice-cream","causal-metadata":<V1>}' http://<ALICE>/kvs/matcha
{"result": "created", "causal-metadata": <V2>, "shard-id": "apples"}
201
```

Remote Key

If instead Alice finds that the key `"matcha"` mapped to a different shard, `"oranges"`, Alice would forward the request to a member of that shard, either Erin or Frank. Let's say Erin receives Alice's forwarded request. First, Erin computes the shard ID of `"matcha"` to confirm that it belongs to Erin's shard `"oranges"`. *Erin now behaves as in the Local Key section, broadcasting any writes to her own local shard. Afterward, Erin responds to Alice with causal metadata reflecting the operation and the shard ID to which the key belongs.* Alice is responsible for forwarding Erin's response, with its causal metadata and shard ID, to the original requester.

If it were again a PUT request with value `"ice-cream"` it would look like:

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value":"ice-cream","causal-metadata":<V1>}' http://<ALICE>/kvs/matcha
{"result": "created", "causal-metadata": <V2>, "shard-id": "oranges"}
201
```

The request and response are the same as the previous example, but the `"shard-id"` is now `"oranges"` to indicate where the key was stored.

Your own distributed system

To try out the key-value store comprised of shards of nodes, you will need to construct a cluster of your own. In this section we'll construct the following scenario:

- 3:3 – Alice, Bob, and Carol in one shard. Dave, Erin, and Frank in another.

Nodes will have subnet addresses from the range `10.10.0.2` to `10.10.0.7`, but all will listen on container port `8090`. Nodes will have their container port `8090` published on host ports from the range `8082` to `8087`. On startup each node will be provided with the environment variables:

- `SOCKET_ADDRESS` is a string in the format `"IP:PORT"` describing the current node.
- `VIEW` is a comma-delimited string containing the socket addresses of all the running nodes.
- `SHARD_COUNT` is the number of shards to divide nodes (and keys) into.

Initial setup

- Build your container image and tag it `asg4img`:

  ```
  $ docker build -t asg4img .
  ```

- Create a subnet called `asg4net` with IP range `10.10.0.0/16`:

  ```
  $ docker network create --subnet=10.10.0.0/16 asg4net
  ```

Run instances in the network

Run each of the nodes in the subnet:

```
$ docker run --rm -p 8082:8090 --net=asg4net --ip=10.10.0.2 --name=alice -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.2:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
$ docker run --rm -p 8083:8090 --net=asg4net --ip=10.10.0.3 --name=bob   -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.3:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
$ docker run --rm -p 8084:8090 --net=asg4net --ip=10.10.0.4 --name=carol -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.4:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
$ docker run --rm -p 8085:8090 --net=asg4net --ip=10.10.0.5 --name=dave  -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.5:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
$ docker run --rm -p 8086:8090 --net=asg4net --ip=10.10.0.6 --name=erin  -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.6:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
$ docker run --rm -p 8087:8090 --net=asg4net --ip=10.10.0.7 --name=frank -e=SHARD_COUNT=2
    -e=SOCKET_ADDRESS=10.10.0.7:8090
    -e=VIEW=10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090,10.10.0.5:8090,10.10.0.6:8090,10.10.0.7:8090
    asg4img
```

Refer to directions on previous assignments about how to deconstruct the cluster, keeping in mind that the names of the instances and subnet are different on this assignment.

Acknowledgement

This assignment was originally based on Peter Alvaro's course design. We gratefully acknowledge the contributions of past CSE138 course staffers, including Reza NasiriGerdeh and Patrick Redmond.

Copyright