CSE138 (Distributed Systems) Assignment 2

Fall 2023

Key-value Store

- **Part 1: Single-site Key-value Store**
  - Create a Key-value Store that will support adding a new key-value pair to the store, retrieving and updating the value of an existing key, and deleting an existing key from the store.
  - The store does not need to persist the key-value data, i.e., it can store data in memory only. That is, if the store goes down and then gets started again, it does not need to contain the data it had before the crash.
- **Part 2: Key-value Store with proxies**
  - Your key-value store will run as a collection of communicating instances, in which exactly one instance has a **main** role and one or more instances have a **forwarding** role.
  - The **main** instance responds to requests directly as in Part 1.
  - A **forwarding** instance forwards each request to the main instance, gets a response from it, and returns the response to the client.

Additional notes:

- You need to implement your own key-value store, and not use an existing key-value store such as Redis, CouchDB, MongoDB, etc.
- For this assignment, you will implement HTTP endpoint handlers and also make HTTP requests. You may use an HTTP *server* library to define your endpoint handlers, and you may also use an HTTP *client* library to make requests.
- Package your web service in a container image, the same as on previous assignments.

General instructions

- You must do your work as part of a team. Teams must have **a minimum of two and a maximum of three students**; we recommend having three.
- You should talk to your teammates about the **programming language for this assignment**.
- Due **Wednesday, October 24th, 2023 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_Assignment2`. Add all the members of the team as collaborators to the repository.
2. Invite the `ucsc-cse138-fall2023-staff` GitHub account as a collaborator to the repository.

3. At the top level of your repository, create a Docker `Dockerfile` or Podman `Containerfile` to describe how to create your container image.
4. Include a `README.md` file in the top level directory with sections:
   - **Acknowledgements**, **Citations**, and **Team Contributions**: Please refer to the below Academic integrity on assignments section.
5. Choose one team member to submit the assignment. Submit your team name, the CruzIDs of all teammates, your repository URL, and the commit ID that you would like to be used for grading to the following Google form: https://forms.gle/AkbgZpK9FP7BNTB39

Academic integrity on assignments

You're expected and encouraged to discuss your work on assignments with others. That said, **all the work you turn in for this course must be your own, independent work (for assignment 1) or the independent work of your team (for subsequent assignments).** Students who do otherwise risk failing the course.

You can ask the TAs, the tutors, and classmates for advice, but you cannot copy from anyone else: once you understand the concepts, you must write your own code. While you work on your own homework solution, you can:

- Discuss with others the general techniques involved, *without sharing your code with them.*
- Use publicly available resources such as online documentation.

In the `README.md` file you include with each assignment, you are **required** to include the following sections:

- *Team Contributions* lists each member of the team and what they contributed to the assignment. (There's no need to include this for assignment 1, since assignment 1 is done independently.)
- *Acknowledgments* lists people you discussed the assignment with and got help from. List each person you talked to and the concept that they helped with. If you didn't get help from anyone, you should explicitly say so by writing "N/A" or "We didn't consult anyone."
- *Citations* is for citing sources you used. For anything you needed to look up, document where you looked it up.

Thorough citation is the way to avoid running afoul of accusations of misconduct.

Building and testing

- To evaluate the assignment, the course staff will create a container image using the Dockerfile in your project directory by running something like:

  ```
  docker build -t your-project .
  docker run --rm -p 8090:8090 your-project
  ```

- We will test Part 1 of your project by sending requests to the key-value store running inside your container. We will be checking that the correct responses and status codes are sent back from your key-value store.
  - We have provided a test script `test_assignment2_part1.py`. The test script expects you to run your web service such that it's available at `localhost:8090`, the same as on the previous assignment. This test script runs quickly.
- For Part 2 of your project we will launch multiple containers with **main** and **forwarding** roles and create a subnet for them to communicate. With that setup we will test by sending requests to particular containers and checking the resultant responses and status codes.
  - The test script `test_assignment2_part2.py` will run several containers in a small virtual network and test communication among the instances. Since the containers have your web service running inside, you don't need to run it manually (and indeed, shouldn't run it manually because it could cause a port collision). This suite takes a couple of minutes to run.

Part 1: Single-site Key-value Store

In this part of the assignment, you will implement a key-value store API over HTTP, backed by an in-memory key-value store (such as a dictionary or map data structure).

`PUT` request at `/kvs/<key>` with JSON body `{"value": <value>}`

This endpoint is used to create or update key-value mappings in the store. It is analogous to hash-map or dictionary operations which add a new key.

- If the key `<key>` does not exist in the store, add a new mapping from `<key>` to `<value>` into the store.
  - Response code is 201 (Created).
  - Response body is JSON `{"result": "created"}`.
- Otherwise, if the `<key>` already exists in the store, then update the mapping to point to the new `<value>`.
  - Response code is 200 (Ok).
  - Response body is JSON `{"result": "replaced"}`.
- If the request body is not a JSON object with key `"value"`, then return an error.
  - Response code is 400 (Bad Request).
  - Response body is JSON `{"error": "PUT request does not specify a value"}`
  - Here is an example for this case:

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"message":"oops"}' http://127.0.0.1:8090/kvs/CSE138
{"error": "PUT request does not specify a value"}
400
```

- If the length of the key `<key>` is more than 50 characters, then return an error.
  - Response code is 400 (Bad Request).
  - Response body is JSON `{"error": "Key is too long"}`.
- `<value>` can be in any value type defined in the JSON specification.

`GET` request at `/kvs/<key>`

This endpoint is used to read values from existing key-value mappings in the store. It is analogous to hash-map or dictionary operations which look up keys and return values.

- If the key `<key>` exists in the store, then return the mapped value in the response.
  - Response code is 200 (Ok).
  - Response body is JSON `{"result": "found", "value": "<value>"}`.
  - Here is an example for this case:

```
$ curl --write-out "\n%{http_code}\n" http://127.0.0.1:8090/kvs/CSE138
{"result": "found", "value": "Distributed Systems"}
200
```

- If the key does not exist in the store, then return an error.
  - Response code is 404 (Not Found).
  - Response body is JSON `{"error": "Key does not exist"}`.

`DELETE` request at `/kvs/<key>`

This endpoint is used to remove key-value mappings from the store. It is analogous to hash-map or dictionary operations which delete keys.

- If the key `<key>` exists in the store, then remove it.
  - Response code is 200 (Ok).
  - Response body is JSON `{"result": "deleted"}`.
- If the key `<key>` does not exist in the store, then return an error.
  - Response code is 404 (Not Found).

- Response body is JSON `{"error": "Key does not exist"}`.

Additional considerations

- If your web service receives a request to the path `/kvs/` or `/kvs` with no key, you could interpret this either as a request for a key with the empty string, or you could interpret this as an error. Either behavior is acceptable, as long as your endpoints are consistent about their treatment of this case.
- Not all requests include a body, but requests with a body will include a `Content-Type` header to describe how the body is encoded. The example `PUT` request above gives the `--data` argument to `curl`, which causes the request to have a body, whereas the example `GET` request above does not. `DELETE` requests also do not have a body.

Part 2: Key-value Store with proxies

In this part of the assignment, you will implement the same key-value store API over HTTP by forwarding requests to a different server and returning its responses.

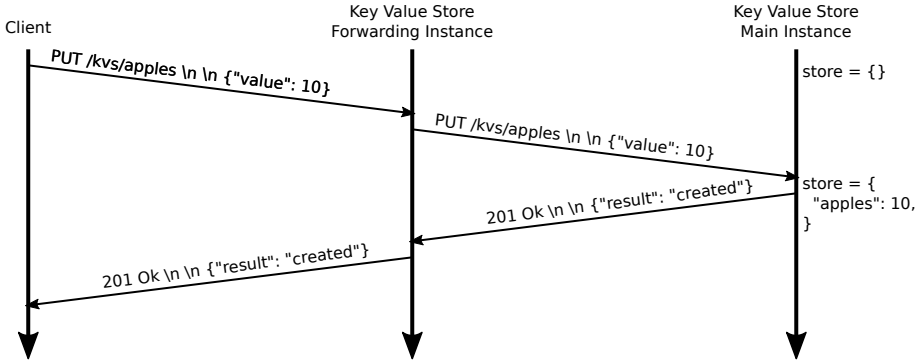

Figure 1: A **forwarding** instance delegates requests to the instance specified in `FORWARDING_ADDRESS`.

Main and Forwarding roles

- When your server starts up, it will be provided with an *environment variable* which describes whether it should be a **main** instance or a **forwarding** instance.
  - The environment variable is named `FORWARDING_ADDRESS`.
  - If `FORWARDING_ADDRESS` is empty (contains the empty string) or is unset, then your server should behave as a **main** instance.
  - If `FORWARDING_ADDRESS` is set and non-empty, then your server should behave as a **forwarding** instance, and forward all requests to the given forwarding address.
- A **main** instance behaves just the same as your key-value store in Part 1.
- A **forwarding** instance exposes the same API as the key-value store in Part 1, however its behavior is not the same.
  - A **forwarding** instance will accept requests, and then forward them along to another server (specified by the forwarding address) to be handled.
  - If the handling server isn't available, then the **forwarding** instance should report an error.

When the Main instance is down

As mentioned above, when a **forwarding** instance receives an API request, such as a `PUT`, `GET`, or `DELETE`, it attempts to forward that request to another server to be handled. However, if the **main** instance is down, then the forwarded request will fail, and this failure must be passed back to the client.

- If a **forwarding** instance is unable to forward a request and obtain a response, then return an error.
  - Response code is 503 (Service Unavailable).
  - Response body is JSON `{"error": "Cannot forward request"}`.

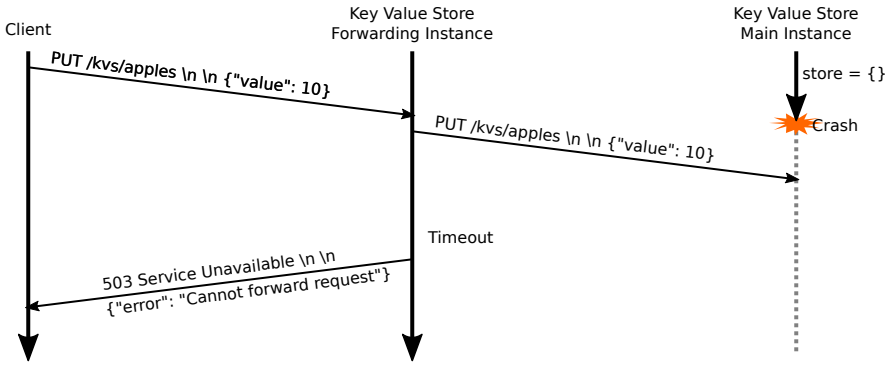

Figure 2: A **forwarding** instance returns an error when it is unable to delegate a request.

Your own distributed system

To test the functionality of **forwarding** and **main** instances working together, here are instructions for creating a small distributed system with multiple docker containers running alongside each other on your computer. These instructions should work similarly for Docker or Podman.

Scenario

We will have a **main** instance and a **forwarding** instance, each running inside Docker containers connected to the same subnet with IP address range `10.10.0.0/16`. The IP address of the **main** instance will be `10.10.0.2` and the IP address of the **forwarding** instance will be `10.10.0.3`. If a client sends a GET, PUT, or DELETE request to the **main** instance, the **main** instance will respond directly as in Part 1 of the assignment. If a client sends a GET, PUT, or DELETE request to the **forwarding** instance, the **forwarding** will make a corresponding request to the **main** instance, wait for a response, and then make a corresponding response to the client. From the client's perspective, their behavior is approximately the same.

Initial setup

- Build your container image and tag it `asg2img`:

  ```
  $ docker build -t asg2img .
  ```

- Create a container subnet called `asg2net` with IP range `10.10.0.0/16`:

  ```
  $ docker network create --subnet=10.10.0.0/16 asg2net
  ```

Run instances in the network

1. Run the **main** instance at IP `10.10.0.2` and map the container's port `8090` to your host machine's port `8082`:

   ```
   $ docker run --rm -p 8082:8090 --net=asg2net --ip=10.10.0.2 --name main-instance asg2img
   ```

2. Run the **forwarding** instance at IP `10.10.0.3` and map the container's port `8090` to your host machine's port `8083`:

```
$ docker run --rm -p 8083:8090 --net=asg2net --ip=10.10.0.3 -e FORWARDING_ADDRESS=10.10.0.2:8090
    --name forwarding-instance1 asg2img
```

Containers will connect to each other directly, over the subnet. Notice that we provide a `FORWARDING_ADDRESS` value to the **forwarding** instance which indicates that the **main** instance can be found using its `asg2net` subnet IP address `10.10.0.2` and *container port* `8090`.

Whereas, if you want to make connections to the docker containers from the host machine, you'll need to use `localhost` or the IP address `127.0.0.1` and the container's *host port*. Use `8082` for the **main** instance or `8083` for the **forwarding** instance.

At this point, you should test all of the above functionality:

1. `PUT`, `GET`, and `DELETE` requests to the **main** instance.
2. `PUT`, `GET`, and `DELETE` requests to the **forwarding** instance, which delegates to the **main** instance.
3. `PUT`, `GET`, and `DELETE` requests to the **forwarding** instance, *while the **main** instance is down*. The next section describes how to take down the instances.

Stop and remove instances

When you're done with the container instances, you must shut them down. Leaving servers running which you aren't using could be a security vulnerability.

1. Check which instances are running. Note the names of the containers in the final column.

```
$ docker ps
CONTAINER ID   IMAGE     COM...   PORTS                                          NAMES
07xxd70a2x89   asg2img   "py...   0.0.0.0:8083->8090/tcp, :::8083->8090/tcp      forwarding-instance1
14xx37419x55   asg2img   "py...   0.0.0.0:8082->8090/tcp, :::8082->8090/tcp      main-instance
```

2. Stop and remove a container. You can remove the containers in either order.

```
$ docker stop main-instance
$ docker rm main-instance
```

   If you stop here, how does the **forwarding** instance behave while the **main** instance is down?

```
$ docker stop forwarding-instance1
$ docker rm forwarding-instance1
```

   The `docker rm` commands might be unnecessary, depending on how you called `docker run` above.

Acknowledgement

This assignment was written by the CSE138 Fall 2021 course staff, based on Peter Alvaro's course design and with input from the staff from past instances of the course, and was later modified by CSE138 Fall 2023 course staff.

Copyright