CSE138 (Distributed Systems) Assignment 3

Spring 2021

Instructions

General

- In this assignment, you will implement a **replicated fault-tolerant key-value store** that provides **causal consistency**. In your key-value store, every key-value pair is replicated to all nodes (replicas). If a node goes down, the key-value store is still available and can respond to clients' requests because the other nodes have a copy of all key-value pairs. The causal consistency model captures the causal relationship between operations and ensures that all clients see the operations in causal order regardless of which replica they contact to do the operations. That is, if an event A happens before event B (B is potentially caused by A), then all clients must first see A and then B.
- You must do your work as part of a team.
- You will use Docker to create an image that exposes a key-value store implementing the HTTP interface described in the next section.
- Your key-value store does not need to persist the data, i.e., it can store data in memory only.
- 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 **05/21/2021 (Friday) 11:59:59 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 test script `test_assignment3.py` that you can use to test your work. It is critical that you run the test script before submitting your assignment. (Note that the test script will take several minutes to run.) The tests provided are the same ones we will run on our side during grading, and we will also run additional tests consistent with the assignment specification. The provided tests are not even close to being exhaustive, and you should certainly do more testing on your own, but they should be enough to get you started.

Submission workflow

- One of the members of your team should create a **private** GitHub repository named `CSE138_Assignment3`. Add all the members of the team as collaborators to the repository.
- Invite the `ucsc-cse138-staff` GitHub account as a collaborator to the repository.

- In addition to the project file(s) implementing the key-value store, the repository should contain **at the top level**:
  - the `Dockerfile` instructing how to create your Docker image.
  - a `README.md` file. The readme should have sections for **Acknowledgements**, **Citations**, and **Team Contributions**. Please refer to the course overview website to learn what needs to go in these sections: http://composition.al/CSE138-2021-03/course-overview.html#academic-integrity-on-assignments
  - a file `mechanism-description.md` including the description of the mechanisms implemented for causal dependency tracking and detecting that a replica is down (see below for more information about this).
- **Commit early and often** as you work on the assignment! We want to see your progress.
- Submit your team name, your repository URL, and the commit ID that you would like to be used for grading to the following Google form: https://forms.gle/apE7S1xhkqP9A2WP7
- Only one of the team members needs to submit the form.

Fault-Tolerant Key-Value Store with Causal Consistency

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

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

To do key-value operations on key `<key>`, a **client** sends GET requests (for retrieving the value of key `<key>`), PUT requests (for adding the new key `<key>` or updating the value of the existing key `<key>`), and DELETE requests (for deleting key `<key>`) 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.

Assume a scenario in which we have three replicas named `replica1`, `replica2`, and `replica3`, each running inside Docker containers 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`, and `10.10.0.4`, respectively. All instances are exposed at port number 8085. Each replica knows its own socket address (IP address and port number) and the view of the store (socket addresses of all replicas). You should give this external information to the replica when you start the corresponding container.

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 assignment3-img .
```

Run Docker containers

To run the replicas, execute

```
$ docker run -p 8082:8085 --net=mynet --ip=10.10.0.2 --name="replica1"
  -e SOCKET_ADDRESS="10.10.0.2:8085" -e VIEW="10.10.0.2:8085,10.10.0.3:8085,10.10.0.4:8085"
  assignment3-img

$ docker run -p 8083:8085 --net=mynet --ip=10.10.0.3 --name="replica2"
  -e SOCKET_ADDRESS="10.10.0.3:8085" -e VIEW="10.10.0.2:8085,10.10.0.3:8085,10.10.0.4:8085"
  assignment3-img

$ docker run -p 8084:8085 --net=mynet --ip=10.10.0.4 --name="replica3"
  -e SOCKET_ADDRESS="10.10.0.4:8085" -e VIEW="10.10.0.2:8085,10.10.0.3:8085,10.10.0.4:8085"
  assignment3-img
```

View Operations

In practice, one or more replicas can go down. We need view operations to read or update the view of the replicas in that case. **To do view operations, the requests need to be sent from a replica (not a client) to the `/key-value-store-view` endpoint at another replica.**

Here are the view operations that your fault-tolerant key-value store should support:

Read a replica's view of the store

```
$ curl --request GET  --write-out "\n%{http_code}\n"
    http://<replica-socket-address>/key-value-store-view

  {"message":"View retrieved successfully","view":<view>}
  200
```

Delete a replica from another replica's view of the store

```
$ curl --request DELETE --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"socket-address": <socket-address-replica-to-be-deleted> }'
    http://<replica-socket-address>/key-value-store-view

  {"message":"Replica deleted successfully from the view"}
  200

$ curl --request DELETE --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"socket-address": <socket-address-replica-to-be-deleted> }'
    http://<replica-socket-address>/key-value-store-view

  {"error":"Socket address does not exist in the view","message":"Error in DELETE"}
  404
```

Add a replica to another replica's view of the store

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"socket-address": <socket-address-replica-to-be-added> }'
    http://<replica-socket-address>/key-value-store-view

  {"message":"Replica added successfully to the view"}
  201

$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"socket-address": <socket-address-replica-to-be-added> }'
    http://<replica-socket-address>/key-value-store-view

  {"error":"Socket address already exists in the view","message":"Error in PUT"}
```

If a replica finds out another replica is down, it deletes that replica from its view and broadcasts a DELETE view request so that the other replicas can do the same. If a new replica is added to the system, it first broadcasts a PUT view request to enable the other replicas to add the new replica to their view. Afterwards, it retrieves all the key-value pairs from one of the replicas and put them into its own local store.

Notice that your key-value store does **not** require causal consistency for view operations. Moreover, it is up to you to implement the mechanism to detect a replica that is down. You need to provide the description of the chosen mechanism in your `mechanism-description.md` file.

Key-value operations under the causal consistency model

You probably already have an intuition for what causal consistency is based on our discussions of happens-before, causal broadcast, and consistent global snapshots. This is the first time, however, that we are applying the idea to a storage system supporting operations such as reads and writes.

In class, we defined causal consistency as follows: **Writes that are potentially causally related must be seen by all processes in the same (causal) order.** (Writes that are *not* potentially causally related, that is, writes that are concurrent or independent, may be seen in different orders on different processes.)

What does it mean for writes to be "potentially causally related"? Consider the happens-before relation that we discussed in class:

The happens-before relation $\rightarrow$ is the smallest binary relation such that:

1. If A and B are events on the same process and A comes before B, then $A \rightarrow B$.
2. If A is a send and B is a corresponding receive, then $A \rightarrow B$.
3. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

With some tweaks to wording, we can adapt the happens-before relation to our key-value store setting:

1. If A and B are operations issued by the same client and A happens first, then $A \rightarrow B$.
2. If A is a **write** operation (i.e., PUT or DELETE) and B is a **read** operation (i.e., GET) that **reads the value written by** A, then $A \rightarrow B$.
3. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

For the purposes of this assignment, you can assume that it is *not* the case that multiple write operations will write the same value to the same location. In other words, if B is a read operation that reads the value x from a given location, then assume that there is a unique write operation A that wrote x to that location.

Consider the following example execution:

Client1: PUT(x,1) → PUT(y,2) → PUT(x,3)
Client2: GET(y)=2 → PUT(x,4)
Client3: GET(x)=4 → PUT(z,5)

In this example, the following operations are related by the → relation:

PUT(x,1) → PUT(y,2) (Case 1)
PUT(y,2) → PUT(x,3) (Case 1)
GET(y)=2 → PUT(x,4) (Case 1)
GET(x)=4 → PUT(z,5) (Case 1)

PUT(y,2) → GET(y)=2 (Case 2)
PUT(x,4) → GET(x)=4 (Case 2)

PUT(x,1) → PUT(x,3) (Case 3)
PUT(x,1) → GET(y)=2 (Case 3)
PUT(y,2) → PUT(x,4) (Case 3)
PUT(x,1) → PUT(x,4) (Case 3)

PUT(x,4) → PUT(z,5) (Case 3)
and more, according to transitivity (Case 3)

Notice that PUT(x,3) and PUT(x,4) are concurrent, i.e., they are *not* ordered by the → relation.

We can now define **causal consistency** for the purposes of this assignment. A key-value store is *causally consistent* if both of the following conditions are satisfied:

1. The effect of a write operation by a client on key `<Key>` will always be visible to a successive read operation on `<Key>` by the same client. In other words, if a client issues a write and then later issues a read, it must either see what it wrote, or it must see the effect of another write that happened causally later.
2. Let `<Key1>` and `<Key2>` be any two keys in the store, and let versions `<Version1>` and `<Version2>` be versions of `<Key1>` and `<Key2>`, respectively, such that `<Version2>` causally depends on `<Version1>`. If a client reads `<Version2>` (i.e., GET(`<Key2>`) returns version `<Version2>`), then `<Version1>` of `<Key1>` is visible to the client too.

Our definition of causal consistency implies that if, for example, PUT(x, 1) → PUT(y, 2), all replicas will first do PUT(x, 1) and then PUT(y, 2), because otherwise, the value of 2 for y might be visible to the client while the value of 1 for x is not visible, which violates condition 2 above. To put it simply, we can say a key-value store is causally consistent if all write (PUT or DELETE) operations on all replicas take effect in the order that respects their causality relationship. However, it would be overkill to try to ensure that all operations take place on all replicas in the same order.

Implementation

You will enforce causal consistency in your key-value store by tracking causal dependencies using **causal metadata** in request and response messages. The approach you take to do this is entirely up to you, as long as you enforce causal consistency. You need to provide a description of your chosen mechanism for tracking causal dependences in your `mechanism-description.md` file. Causal metadata can take various forms; for example, **vector clocks** are one form of causal metadata (and we recommend that you use vector clocks, but we don't require it, and there are other approaches that would work).

To implement causal consistency, the client needs to participate in the propagation of causal metadata. In particular, when a client issues a write, it needs to indicate *which of its prior reads* may have caused the write. **Therefore, your key-value store should include a `causal-metadata` field in responses to client requests.** The representation of causal metadata doesn't matter to the client, but clients will be sure to include it in all requests to your key-value store after receiving it in the first response.

Let's walk through the first few steps of the example execution from above. Suppose that `<V1>`, `<V2>`, `<V3>`, `<V4>`, and `<V5>` are versions of causal metadata generated by PUT(x,1), PUT(y,2), PUT(x,3), PUT(x,4), and PUT(z, 5), respectively.

First, Client1 sends PUT(x,1) to a replica. The causal metadata for this request is empty because it does not depend on any other PUT operation.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
  --data '{"value": 1, "causal-metadata": ""}' http://<replica-socket-address>/key-value-store/x

  {"message":"Added successfully", "causal-metadata": "<V1>"}
  201
```

The replica receiving the request first checks the causal metadata. Because the causal metadata is empty, the replica knows that the request is not causally dependent on any other PUT operation. Therefore, it generates causal metadata for the PUT operation, stores the key, the value, and the causal metadata in its local store, and responds to the client with a message containing the causal metadata that the client needs to use in its next operation. It also broadcasts the write to the other replicas.

Note: a replica that gets a client request does **not** necessarily have to wait to hear from the other replicas before acknowledging the write to the client. In other words, this is **not** like primary-backup replication or chain replication, both of which provide strong consistency. The only time a replica has to wait for other replicas is if it is waiting to get a write that it needs to ensure causal consistency.

Next, Client1 sends PUT(y,2) to a replica (it doesn't have to be the same replica it sent the PUT(x,1) request to). The client sets the causal metadata of the request to `"<V1>"`, which is what was returned from the last operation (PUT(x,1)) .

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
  --data '{"value": 2, "causal-metadata": "<V1>"}'
  http://`<replica-socket-address>`/key-value-store/y

  {"message":"Added successfully", "causal-metadata": "<V2>"}
  201
```

When the replica gets PUT(y,2) from Client1, it checks the causal metadata in the request and finds that it causally depends on PUT(x,1). The replica checks whether it has already done PUT(x,1). If so, it generates the causal metadata `<V2>` for the operation and stores the key, value, and corresponding causal metadata in its local store. It also responds to the client with a message containing the causal metadata that the client will use in the next request and broadcasts the request to the other replicas. Otherwise, it waits until it receives the PUT(x,1) operation from another replica. After receiving that request, it first applies PUT(x,1) and then PUT(y,2) to respect the causal dependency between the PUT operations.

The other replicas that receive PUT(y, 2) must also make sure that they have already done the PUT(x,1) operation. If not, they must wait to receive it. You are responsible for implementing this behavior, based on the causal metadata.

Next, Client2 sends GET(y) to a replica. The GET request does not contain any value for the `causal-metadata` field.

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

  {"message":"Retrieved successfully", "causal-metadata": "<V2>", "value": 2}
  200
```

The replica responds with the latest value of the key `y` (in this case 2), and the causal metadata that the client will use in its next PUT request. There is no need to forward the GET request to the other replicas. Please notice that in our example, the GET(y) request reads the version written by PUT(y,2) because the replica receives GET(y) after it has done PUT(y,2). However, it is possible that the replica receives the GET(y) request before doing PUT(y,2), in which case the key-value store should return an error message indicating that the key does not exist, as in the Assignment 2 specification.

Moreover, if another client sent a PUT request on key `y` before the GET(y) request from Client2, the value and causal metadata returned to the client might be different. For GET requests on a particular key `<key>`, the replica responds either with the client's latest write to the key `<key>`, or a value written by a causally later write, along with the corresponding causal-metadata.

Client2 sends PUT(x,4) to a replica. The causal metadata of the request is `<V2>`, which is the causal metadata returned by the last GET request from Client2.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value": 4, "causal-metadata": "<V2>"}'
    http://<replica-socket-address>/key-value-store/x

  {"message":"Updated successfully", "causal-metadata": "<V4>"}
  200
```

The replica receiving the request first makes sure that it has already done any write operations that causally precede this PUT, according to the causal metadata. If not, it waits for those operations to arrive. Finally, it broadcasts the request so that the other replicas can do the same.

DELETE operations

A DELETE operation can be thought of as a PUT operation that sets the value of a key to NULL. For example, suppose that Client2 sends a DELETE request to a replica to remove key `y` after sending the request PUT(x,4). The causal metadata for the DELETE operation is `"<V4>"`, which was returned from the latest operation of PUT(x,4).

```
$ curl --request DELETE --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"causal-metadata": "<V4>"}' http://`<replica-socket-address>`/key-value-store/y

  {"message":"Deleted successfully", "causal-metadata": "<V6>"}
  200
```

The replica that receives the DELETE request checks the causal metadata and makes sure that it has completed all causally preceding operations before applying the DELETE. To do the DELETE operation, the replica generates causal metadata `<V6>` and adds the NULL value and the causal metadata to the store. It also broadcasts the DELETE request to the other replicas.

If the key does not exist, the key-value store should return the appropriate error message, as in the Assignment 2 specification.

Additional notes

- If your service receives a request to `/key-value-store/` you can choose to interpret it as an error, or as a request where the `key` is the empty-string `""`. Either behavior is acceptable.
- Requests which include a body will have a `Content-Type` header, but requests which don't have a body won't have that header. That's why the `curl` commands in this document don't always send that header. In this assignment all requests, except the `GET` requests to the endpoint `/key-value-store-view/`, include a body.

A note about conflict resolution

The causal consistency model ensures that PUT operations take effect in causal order. However, it doesn't say anything about PUT operations that are concurrent. For instance, in the example above, PUT(x,3) and PUT(x,4) are concurrent, and replicas can do them in any order without violating causal consistency.

Conflicts can occur if PUT operations on the same key are concurrent. Consider the following scenario:

- Client1 sends PUT(x,3) to Replica1 and Client2 sends PUT(x,4) to Replica2.
- Replica1 checks the causal metadata of PUT(x,3) and ensures that all PUT operations that PUT(x,3) causally depends on have already been done. Replica2 does the same for PUT(x,4).
- Replica1 applies PUT(x,3) and forwards the PUT(x,3) request to Replica2. At the same time, Replica2 applies PUT(x,4) and forwards the PUT(x,4) request to Replica1.
- Replica1 applies PUT(x,4) and Replica2 applies PUT(x,3). In this situation, the value of the latest key in Replica1 is 4 but it is 3 in Replica2. This is an example of a conflict that occurred because of concurrent writes on the same key.

There are various mechanisms to detect and resolve conflicts in a fault-tolerant causally consistent key-value store. For example, replicas can employ **gossiping** to find out the conflicts and use last-write-wins (e.g., the last write can be determined by a timestamp value added by the client to the PUT operation) as the conflict resolution mechanism.

Conflicts might also be identified during GET operations. That is, if a replica receives a GET operation for a particular key, it can forward the GET request to the other replicas and compare the values from all replicas, including itself, to see if there is all conflict. It might then select one of the values as the final value and ask the other replicas to change their value to that value.

For this assignment, you may assume that there are no concurrent writes to the same key, and therefore, no conflict resolution should be necessary. Therefore, you do not need to implement a conflict resolution mechanism. However, a realistic key-value store would not be able to make this assumption, and we invite you to implement a conflict resolution mechanism as an additional challenge, if you want to.

Acknowledgment

This assignment was written by Reza NasiriGerdeh and Lindsey Kuper, based on Peter Alvaro's course design. The current version includes updates suggested by the CSE138 Spring 2021 course staff (http://composition.al/CSE138-2021-03/course-overview.html#course-staff).

Copyright