CSE138 (Distributed Systems) Assignment 3

Fall 2023

Replicated Key-Value Store

- Create a *replicated*, *fault-tolerant*, and *causally consistent* **key-value store**.
  - Your key-value store will run as a collection of communicating instances. Key-value pairs are replicated to all instances (replicas). Replicas communicate state changes (the additions and removals of keys) among each other. Replicas respect the causal order of events when updating their copy of the store and expose a causally consistent view of the store to clients.
  - If one replica goes down (or crashes) the store is still available because other replicas are still up. Replicas do not persist the key-value data. A replica's local copy is lost when the replica stops, but due to replication almost no data is lost.
  - Replicas provide causal metadata to clients in responses. Clients send causal metadata to replicas in their requests to ensure they see a causally consistent view of the store.
  - Clients can make requests to any replica in the cluster.

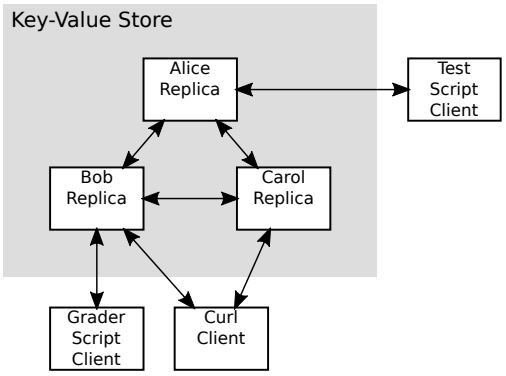

Figure 1: Box-diagram of Replicated Key-Value Store with clients.

Additional notes:

- You need to implement your own key-value store, not use an existing one such as Redis, CouchDB, etc, the same as on previous assignments.
- 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. Feel free to form a new team.
- You should talk to your teammates about the **programming language for this assignment**.

- Due **Saturday, November 18th, 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_Assignment3`. 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. *This is a team assignment, and contributions from all the team members are required.*
   - **Mechanism Description**: Describe how your system tracks causal dependencies, and how your system detects when a replica goes down.
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/dFe2FCPKsvk4vGXu6

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 from your source code using the Docker `Dockerfile` or Podman `Containerfile` you provide. We will instantiate several copies of the container in a cluster. We will test your implementation to check that the API is correct, and that it ensures causal consistency.
- We have provided a test script `test_assignment3.py` that you can use to check your work. 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.

Replicated Key-Value Store APIs

Your store will support two kinds of operations: View operations are handled at the `/view` endpoint, and Key-Value operations are handled at the `/kvs` endpoint. **View operations** manage which replicas are part of the cluster, and **key-value operations** perform the useful work of a key-value store.

View operations

The "view" refers to the current set of replicas among which the store is replicated. A replica supports **view operations** to describe the current view, add a new replica to the view, or delete an existing replica from the view. Your key-value store does *not* maintain causal consistency for view operations.

In normal operation of a distributed system one or more replicas may be added or may go down (or crash). The view must change in response to the participating replicas.

- If a replica finds out another replica is down, it deletes the downed replica from its view and broadcasts a `DELETE`-view request so that the other replicas can do the same. Describe how your system detects when a replica goes down in the Mechanism Description section of your README.
- When a new replica is added to the system, it broadcasts a `PUT`-view request so the existing replicas add the new replica to their view. Afterwards the new replica retrieves all the key-value pairs from an existing replica and adds them to its own store.

`PUT` request at `/view` with JSON body `{"socket-address":"<IP:PORT>"}`

This endpoint adds a new replica to the view.

- If the `<IP:PORT>` is not already part of the view, add it to the view.
  - Response code is 201 (Created).
  - Response body is JSON `{"result": "added"}`.
  - Here is an example for this case:

```
$ curl --request PUT --header "Content-Type: application/json" --data
    '{"socket-address":<NEW-REPLICA>}' http://<EXISTING-REPLICA>/view
{"result": "added"}
```

- If the `<IP:PORT>` is already part of the view, then be idempotent and do nothing.
  - Response is 200 (Ok).
  - Response body is JSON `{"result": "already present"}`.

`GET` request at `/view`

This endpoint retrieves the view from a replica.

- Retrieve the current view unconditionally.
  - Response code is 200 (Ok).
  - Response body is JSON `{"view": ["<IP:PORT>", "<IP:PORT>", ...]}`.
    - The response body is a JSON object with a key `"view"` to a list of strings identifying the replicas in the view by IP address and port number. For example, if the view contains the three replicas `10.10.0.2:8090`, `10.10.0.3:8090`, and `10.10.0.4:8090` then the response body is:

      `{"view": ["10.10.0.2:8090", "10.10.0.3:8090", "10.10.0.4:8090"]}`

`DELETE` request at `/view` with JSON body `{"socket-address":"<IP:PORT>"}`

This endpoint removes an existing replica from the view.

- If the `<IP:PORT>` is already part of the view, then remove it.
  - Response is 200 (Ok).

  - Response body is JSON `{"result": "deleted"}`.
- If the `<IP:PORT>` is not already part of the view, then return an error.
  - Response code is 404 (Not Found).
  - Response body is JSON `{"error": "View has no such replica"}`.

Key-Value operations

The supported **key-value operations** are similar to the previous assignment, with the additional responsibility of enforcing causal consistency. A replica supports adding or updating a key-value pair, retrieving the value associated with a key, and deleting an existing key-value pair. Additionally, operations track causal dependencies by propagating **causal metadata**. A discussion of the details of how these operations should function is available in the Example walkthrough, but it's better to continue reading with the section on Safety and Liveness. Here is a brief overview of the endpoint signatures.

`PUT` request at `/kvs/<key>` with JSON body

- Request body is JSON `{"value": <value>, "causal-metadata": <V>}`.
  - `<V>` is `null` when the PUT does not depend on prior writes.
- Response is one of:
  - 201 (Created) `{"result": "created", "causal-metadata": <V'>}`
  - 200 (Ok) `{"result": "replaced", "causal-metadata": <V'>}`
    - The `<V'>` here and in the 201 response indicates a causal dependency on `<V>` and this PUT.
  - 400 (Bad Request) `{"error": "PUT request does not specify a value"}`
  - 400 (Bad Request) `{"error": "Key is too long"}`
  - 503 (Service Unavailable) `{"error": "Causal dependencies not satisfied; try again later"}`

`GET` request at `/kvs/<key>` with JSON body

- Request body is JSON `{"causal-metadata": <V>}`.
  - The `<V>` is `null` when the client does not know of prior writes.
- Response is one of:
  - 200 (Ok) `{"result": "found", "value": "<value>", "causal-metadata": <V'>}`
    - The `<V'>` indicates a causal dependency on the PUT of `<key>`,`<value>`.
  - 404 (Not Found) `{"error": "Key does not exist"}`
  - 503 (Service Unavailable) `{"error": "Causal dependencies not satisfied; try again later"}`

`DELETE` request at `/kvs/<key>` with JSON body

- Request body is JSON `{"causal-metadata": <V>}`.
  - The `<V>` is `null` when the DELETE does not depend on prior writes. **Note:** *This should never happen. Think about why.*
- Response is one of:
  - 200 (Ok) `{"result": "deleted", "causal-metadata": <V'>}`
    - The `<V'>` indicates a causal dependency on `<V>` and this DELETE.
  - 404 (Not Found) `{"error": "Key does not exist"}`
  - 503 (Service Unavailable) `{"error": "Causal dependencies not satisfied; try again later"}`

Safety and Liveness

Your key-value store must ensure the safety property *causal consistency* and the liveness property *eventual consistency*.

- The safety property **causal consistency** requires that participants in a system agree on the relative ordering of writes that are potentially causally related.
- The liveness property **eventual consistency** requires that if writes stop arriving, replicas will eventually agree on the data.

To ensure these properties, your key-value store replicas must share updates among themselves. Eventual consistency only requires that replicas eventually agree, but causal consistency needs more discussion. The rest of this section is focused on causal consistency.

Potentially causally related

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., a PUT or DELETE at `/kvs`) and B is a **read** operation (i.e., GET at `/kvs`) that **reads the value written by** A, then $A \rightarrow B$.
3. If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Example execution

Consider the following example execution in which time flows to the right:

- Client1: PUT(x,1) $\rightarrow$ PUT(y,2) $\rightarrow$ PUT(x,3)
- Client2: GET(y)=2 $\rightarrow$ PUT(x,4)
- Client3: GET(x)=4 $\rightarrow$ PUT(z,5)

In this execution, the following operations are related by the happens-before "$\rightarrow$" relation:

- Case 1
  - PUT(x,1) $\rightarrow$ PUT(y,2)
  - PUT(y,2) $\rightarrow$ PUT(x,3)
  - GET(y)=2 $\rightarrow$ PUT(x,4)
  - GET(x)=4 $\rightarrow$ PUT(z,5)
- Case 2
  - PUT(y,2) $\rightarrow$ GET(y)=2
  - PUT(x,4) $\rightarrow$ GET(x)=4
- Case 3
  - PUT(x,1) $\rightarrow$ PUT(x,3)
  - PUT(x,1) $\rightarrow$ GET(y)=2
  - PUT(y,2) $\rightarrow$ PUT(x,4)
  - PUT(x,1) $\rightarrow$ PUT(x,4)
  - PUT(x,4) $\rightarrow$ PUT(z,5)

. . . and more, according to transitivity (Case 3).

Notice that PUT(x,3) and PUT(x,4) are concurrent. They are *not* ordered by the $\rightarrow$ relation.

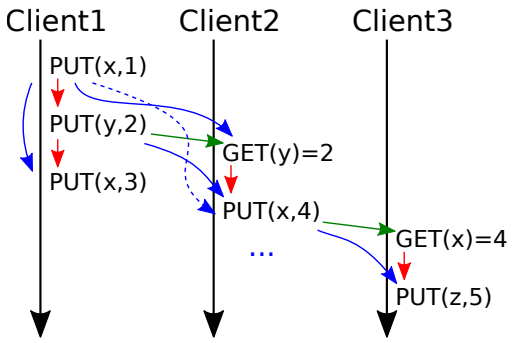

Figure 2: Example execution showing which events are related by which case of the happens-before "$\rightarrow$" relation. Case 1 is red, case 2 is green, and case 3 is blue. The key-value store is not shown.

Definition of causal consistency

We can now define **causal consistency** for the purposes of this assignment. Since both clients and replicas (processes) are participating in the system, both will play a role in enforcing causal consistency. Our definition is in terms of what clients can observe about the data in the key-value store. A key-value store is *causally consistent* if both of the following conditions are satisfied:

1. Suppose two writes A and B occur, such that B causally depends on A (i.e., $A \rightarrow B$). They may be any combination of DELETE or PUT writes. If the effect of write B is visible to some client via a GET, then the effect of write A is also visible to that client.
2. The effect of a write operation by a client must be visible to a subsequent read operation by the same client. (i.e., If a client issues a write to a key K and then later issues a read to K, the client must either see what it wrote or see the effect of some causally later write. This is called *read your writes* consistency.)

For example, our definition of causal consistency implies that if PUT(x,1) $\rightarrow$ PUT(y,2), then all replicas will first do PUT(x,1) and then PUT(y,2). Otherwise, the value of 2 for y might be visible to a client while the value of 1 for x is not visible, which would be a violation of the first condition above.

Tracking causal metadata

To enforce causal consistency your key-value store must track causal dependencies by passing **causal metadata** in requests and responses. The choice of what representation to use for causal metadata and how to enforce causal consistency is your decision. There are various approaches that will work, and one such approach is to use **vector clocks**, but you may choose another. Describe how your system tracks causal dependencies in the Mechanism Description section of your README.

To implement causal consistency both replicas and clients must 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, requests made by clients and responses returned by your key-value store to clients must both include a `causal-metadata` field. The client will not interpret the causal metadata value, but clients will always include the latest causal metadata value they received in their subsequent requests. When a client first starts interacting with the key-value store, its initial request will have `null` for the `causal-metadata` field.

```
{ ... "causal-metadata": <causal-metadata>, ... }
```

Example walkthrough

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

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

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value":1,"causal-metadata":null}' http://<REPLICA>/kvs/x
{"result": "created", "causal-metadata": <V1>}
201
```

- The replica receiving PUT(x,1) first checks the causal metadata and then processes the request.
  - Since the causal metadata is empty, the replica knows that the request is not causally dependent on any other PUT operation.
  - The replica generates causal metadata `<V1>` for the PUT operation, stores the key-value pair, updates local causal metadata, and responds to the client with a JSON object including the causal metadata that the client needs to use in its next operation. Finally, the replica 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.*

- Next, Client1 sends PUT(y,2) to a replica, which could be different from the replica which processed the prior request. The client sets the causal metadata of the request to `<V1>` because that was returned from the prior operation.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value":2,"causal-metadata":<V1>}' http://<REPLICA>/kvs/y
{"result": "created", "causal-metadata": <V2>}
201
```

- When the replica receives PUT(y,2) from Client1, it first checks the causal metadata.
  - The causal metadata indicates that PUT(y,2) causally depends on PUT(x,1).
  - The replica checks whether it has already applied PUT(x,1).
  - If PUT(x,1) was already applied, the replica generates the causal metadata `<V2>` for the PUT(y,2) operation, stores the key-value pair, updates local causal metadata, and responds to the client with a JSON object including the causal metadata that the client will use in the next request. The replica then broadcasts the write to the other replicas.
  - Otherwise, the replica should return an error.
    - Response code is 503 (Service Unavailable).
    - Response body is JSON `{"error": "Causal dependencies not satisfied; try again later"}`.

The other replicas that receive PUT(y,2) must also make sure that they have already done the PUT(x,1) operation, otherwise they must return the 503 error described above.

- Client2 sends GET(y) to a replica. Since this is the first request made by Client2, it has `null` for the `causal-metadata` field.

```
$ curl --request GET --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"causal-metadata":null}'  http://<REPLICA>/kvs/y
{"result": "found", "value":2, "causal-metadata": <V2>}
200
```

- The replica first checks the causal metadata.
  - If the causal metadata is `null` or the writes indicated by it have all been applied at the current replica, then the replica responds with the latest value of the key `y` which is `2`, and the causal metadata which the client will use in its next request. There is no need to broadcast a GET request to the other replicas.
  - Otherwise, the replica should return an error, as specified above.

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 previous assignment specification.

- Client2 sends PUT(x,4) to a replica. The causal metadata of the request is `<V2>` because that is the causal metadata Client2 received in its prior request.

```
$ curl --request PUT --header "Content-Type: application/json" --write-out "\n%{http_code}\n"
    --data '{"value":4,"causal-metadata":<V2>}' http://<REPLICA>/kvs/x
{"result": "replaced", "causal-metadata": <V4>}
200
```

- The replica receiving the request first checks the request causal metadata.
  - If the replica has already performed the write operations which precede this PUT according to the causal metadata, it performs the usual steps as described above including broadcasting this write to the other replicas.
  - Otherwise, the replica should return an error as specified above.

Note: *A DELETE operation can be thought of in the same way as a PUT operation because both are writes to the key-value store.*

A note about broadcasts

In the course of broadcasting a write to all the replicas, you'll get error **503 Service unavailable** `{"error": "Causal dependencies not satisfied; try again later"}`. This error means your write was rejected by one replica, but you are still responsible for making sure that the write reaches that replica eventually. You will need to implement some mechanism to complete the broadcast.

- One method would be to sleep 1 second and retry the write at that replica until it is successful. This is *buffering messages at the sender*.
- Another method would be to have an alternate API for receiving writes which always accepts the write, but keeps it for later if it cannot be applied immediately. This is *buffering messages at the receiver*.
- Yet another method would be to have an alternate API for receiving writes which doesn't respond until the causal metadata necessary to apply the write is available. This is *HTTP long-polling*.

Each of these approaches comes with tradeoffs. In previous offerings of this class we required students implement HTTP long-polling, but this quarter we're leaving it up to you.

A note about conflict resolution

The causal consistency model ensures that write operations take effect in causal order, but the model doesn't say anything about write operations that are concurrent. In the Example execution, the writes PUT(x,3) and PUT(x,4) are concurrent, meaning that replicas can apply them in any order without violating causal consistency.

Conflicts can occur when PUT operations on the same key are concurrent. If one replica applies PUT(x,3) before PUT(x,4), then that replica will have 4 for x. Meanwhile, if another replica applies them in the opposite order then that replica will have 3 for x. The two replicas aren't in agreement about the value of x!

There are various mechanisms to detect and resolve conflicts in a fault-tolerant causally consistent key-value store. To fully satisfy the *eventual consistency* liveness property of this assignment, you would need to find a way to make your replicas come to agreement about conflicts. **However, for this assignment you may assume that there are no concurrent writes to the same key, and so you do not need to implement a conflict resolution mechanism.** 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 extra challenge if you want to.

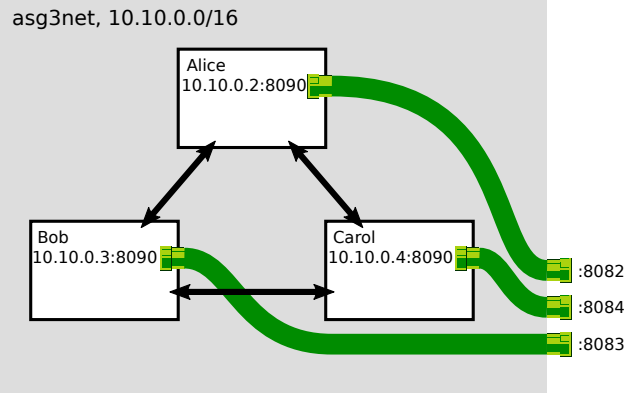

Figure 3: The key-value store running in a container subnet has two socket addresses for each replica.

Your own distributed system

To try out the key-value store comprised of communicating replicas, you will need to construct a cluster of your own.

We will have three replicas named `alice`, `bob`, and `carol` running in containers connected to the same subnet, with IP address range `10.10.0.0/16`. The subnet IP addresses of the replicas will be `10.10.0.2`, `10.10.0.3`, and `10.10.0.4`, respectively. All replicas must listen on container port 8090. On startup, each replica will be provided with environment variables `SOCKET_ADDRESS` and `VIEW`.

- `SOCKET_ADDRESS` is a string in the format `"IP:PORT"` which corresponds to the current replica. Here is an example, for `alice`:

  ```
  SOCKET_ADDRESS="10.10.0.2:8090"
  ```

- `VIEW` is a comma-delimited string containing the socket addresses of the replicas participating the key-value store.

  ```
  VIEW="10.10.0.2:8090,10.10.0.3:8090,10.10.0.4:8090"
  ```

Initial setup

- Build your container image and tag it `asg3img`:

  ```
  $ docker build -t asg3img .
  ```

- Create a subnet called `asg3net` with IP range `10.10.0.0/16`:

  ```
  $ docker network create --subnet=10.10.0.0/16 asg3net
  ```

Run instances in the network

Run each of `alice`, `bob`, and `carol` in the subnet:

```
$ docker run --rm -p 8082:8090 --net=asg3net --ip=10.10.0.2 --name=alice
    -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 asg3img
$ docker run --rm -p 8083:8090 --net=asg3net --ip=10.10.0.3 --name=bob
    -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 asg3img
$ docker run --rm -p 8084:8090 --net=asg3net --ip=10.10.0.4 --name=carol
    -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 asg3img
```

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 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