CSE 138: Distributed Systems

Winter 2023 - Assignment #3

Assigned: Monday, 02/13/2023 — Due: Monday, 02/27/2023

Contents

General Instructions

This specification document is long and full of details; make sure you read it thoroughly.

- You must be able to handle GET and DELETE requests with JSON bodies.
- You must do your work as part of a team.
- You may not use an existing key-value store (e.g. Redis, MongoDB, etc.).
- You will use **Docker** to create a container that runs a RESTful distributed key-value store.
- In this assignment, you will implement a **fault-tolerant key-value store** using a **replication strategy** that provides **causal consistency**. 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, and all clients must **first see A and then B**. Causal consistency will be explained later in the document.
  - The key-value store node must be able to:
    1. Do all data operations similar to Assignment #2.

2. Always return a causally consistent response to data operations or, if it cannot, respond with an error (never inconsistent).
3. Create a new cluster, join a cluster, or idly wait until added to a cluster by another node.

– A key-value store cluster (of nodes) must:

1. Replicate every key to all nodes (replicas).
2. Remain avaliable if a node goes down, and handle client requests. That is, the nodes that are still up (even if only 1 is still up) should remain available (and responsive), subject to the constraint 2 above.

Building and testing your container

- We provide a test script, `test_assignment2.py` that you **should** use to test your work.
- The tests provided are the same ones we will run on our side during grading. We may also run additional tests consistent with the assignment specification.

Submission workflow

- A private repository should be created by one of the members of a team.
- The GitHub accounts of the other members of the team as well as `akarbas` should be added as collaborators to the repository.
- The repository should contain:
  - the project file(s) implementing the key-value store
  - the `Dockerfile` instructing how to create your Docker image. **The `Dockerfile` should be at the root of the repository.**
  - a file `mechanism-description.txt` including the description of the mechanisms implemented for causal dependency tracking and detecting that a replica is down.
  - a file `contributions.yml` describing the contributions of each member of the team. The file should be formatted as follows: each new entry begins with a dash and a space (`-`), then comes the CruzID of the member, a colon and a space, and then the contributions of the member in quotes. E.g.:

```
- <cruzid1>: '<Contributions of member 1>'
- <cruzid2>: '<Contributions of member 2>'
```

- Your team name (CruzID of one of the members — should be unique and consistent across submissions), CruzIDs of all members, repository URL, and commit id (the commit to be graded) should be submitted through the following Google form: https://forms.gle/3mYCxgCf2Mwvo9NYA
- Only one of the team members should submit the form.
- We will only grade the last commit ID of a team submitted to the Google Form below, so it is OK to submit more than once.
- The assignment is due **Saturday 02/25/2023 11:59 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.

Key-value operations under the causal consistency model

In class, we define causal consistency as follows: **Writes that are potentially causally related must be seen by all processes in the same (causal) order.** What does it mean for writes to be "potentially causally related"? Consider the happens-before relation:

The happens-before relation `-->` 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 --> B`.

2. If `A` is the sending of a message and `B` is the corresponding receive, then `A --> B`.
3. Transitivity: If `A --> B` and `B --> C`, then `A --> C`.

Note that this allows pairs of events to be "concurrent" — not be ordered by the happens-before relation. While causal consistency does not require such operations to have the same order in different processes, we require you ensure that such operations are ordered similarly in all processes as well (see the discussion about tie-breaking).

In our setting, any of the following is an event (the list is not exhaustive and more cases may exist):

- A client PUT request to write a new key.
- A client PUT request to update the value of a key.
- A client DELETE request to remove a key.
- A client GET request to get the value of a key. (depending on the mechanism; e.g., in the possible solution outlined below, GET requests do not count.)

Example 1. Consider the case that there are two replicas and two clients, and the replicas cannot connect to each other (e.g., because of a network problem).

- `A1`: client1 sends `PUT(y, 10)` to replica2; `A2`: replica2 receives it.
- `B1`: client1 sends `PUT(y, 20)` to replica1; `B2`: replica1 receives it.
- `C1`: client1 sends `GET(y)` to replica1; `C2` replica1 receives it.
- `C3`: replica1 sends `GET(y)=20` to client1; `C4` client1 receives it.
- `D1`: client1 sends `PUT(x, 5)` to replica2; `D2`: replica2 receives it.
- `E1`: client2 sends `GET(x)` to replica2; `E2`: replica2 receives it.
- `E3`: replica2 sends `GET(x)=5` to client2; `E4`: client2 receives it.

1. Same process:

```
client1: A1 --> B1 --> C1 --> C4 --> D1
client2: E1 --> E4
replica1: B2 --> C2 --> C3
replica2: A2 --> D2 --> E2 --> E3
```

2. Messages:

```
A1 --> A2
B1 --> B2
C1 --> C2
C3 --> C4
...
```

3. Transitivity:

```
A1 --> A2 --> D2 --> E2 --> E3
A1 --> B1 --> B2 --> C1 --> C2 --> C3 --> C4 --> D1 --> D2 --> E2 --> E3 --> E4
...
```

Notes:

- Events like A2 and B2 are concurrent and are are *not* ordered by the happens-before relation.
- You can assume that multiple writes of the same value to the same replica will not happen. In other words, if `B` is a read operation that reads the value of `x` from a location, then there is a unique write operation `A` that wrote `x` to that location.

See the discussion about tie-breaking to see how to solve the first two.

We now define two more important concepts, **causal dependency** and **visibility**. For demonstration purposes, we use *versions* to identify write operations. A version is generated by any write (PUT or DELETE) operation and it uniquely identifies the corresponding operation.

Causal Dependency

Let `V1` and `V2` be the versions of keys `Key1` and `Key2` generated by `PUT(Key1, Value1)` and `PUT(Key2, Value2)`, respectively. We say that version `V2` *causally depends* on version `V1` if `PUT(Key1, Value1) --> PUT(Key2, Value2)`.

Potential Visibility

We say that a version `Vx` (of any key) is *potentially visible* to the client if some GET(`Key`) sent from the client to any replica returns version `Vx` or version `Vy` such that `Vy` causally depends on `Vx`. In other words, if GET returns version `Vy`, then version `Vy` and all versions that `Vy` causally depends on are potentially visible to the client. By returning version `Vy`, we mean returning the value written by the PUT/DELETE operation that generated version `Vy`.

Notes:

- Writes to different keys, and hence different keys' versions, can be causally dependent on each other, and potential visibility is defined across all keys.
- Not all versions that are potentially visible to a client can be returned to the client. For example, if `Vx` and `Vy` (with `Vy >= Vx`) affect the same key and are both potentially visible to a client, returning the result of `Vx` to the client breaks causal consistency. In other words, if multiple versions of the same key are potentially visible to a client, from those versions, only the version(s) that is not depended on can be returned to that client.

In Example 1, if `GET(x)` returns version `V3` generated by `PUT(x, 5)` to the client, then both versions `V1`, `V2` generated by `PUT(y, 10)` and `PUT(y, 20)` are potentially visible to the client because `V3` causally depends on `V1` and `V2`.

We can now define **causal consistency** for the purposes of this assignment:

Causal Consistency

A key-value store is *causally consistent* if the following conditions are satisfied:

1. A client's operations causally depend on that client's past writes.
    - The effect of a write operation by a client on key `Key` will always be potentially visible to a successive read operation on `Key` by the same client. In other words, if a client does a write and then later does a read, it will either see what it wrote, or if it sees a different value, it was not written causally earlier.
    - A client's write operations causally depend on all of the client's causally previous write operations. That is, if a client writes `Vx` and then writes `Vy` (possibly to a different replica), `Vy` causally depends on `Vx`.
2. Let `Key1` and `Key2` be any two keys in the store and let versions `V1` and `V2` be versions of `Key1` and `Key2`, respectively, such that `V2` causally depends on `V1`. If a client reads `V2`, i.e., `GET(Key2)` returns version `V2`, then `V1` is also potentially visible to the client.
3. If multiple versions of a key are potentially visible to a client, only a version that causally depends on all others should be returned in response to a GET operation.

In a simplified manner, we can say a key-value store is causally consistent if all PUT operations on all replicas are done in the order that respects their causality relationship. In this simplified way, if `PUT(Key1, Value1) --> PUT(Key2, Value2)`, all replicas will first do `PUT(Key1, Value1)` and then `PUT(Key2, Value2)` because, otherwise, `V2` might be visible to the client while `V1` is not visible, which violates condition 2 above. However, as shown in Examples 1 and 2, the operations `PUT(x, 2)` (i.e., `V3`) and `GET(x)` *can* be performed by replica2 even before it performs the writes that constitute `V1` and `V2`.

Example 2. This is the same as Example 1, with the addition of events `F1` and `F2`

- `A1`: client1 sends `PUT(y, 10)` to replica2; `A2`: replica2 receives it.
- `B1`: client1 sends `PUT(y, 20)` to replica1; `B2`: replica1 receives it.
- `C1`: client1 sends `GET(y)` to replica1; `C2` replica1 receives it.

- `C3`: replica1 sends `GET(y)=20` to client1; `C4` client1 receives it.
- `D1`: client1 sends `PUT(x, 5)` to replica2; `D2`: replica2 receives it.
- `E1`: client2 sends `GET(x)` to replica2; `E2`: replica2 receives it.
- `E3`: replica2 sends `GET(x)=5` to client2; `E4`: client2 receives it.
- `F1`: client2 sends `GET(y)` to replica2; `F2` replica2 receives it.

Again, `V1` corresponds to `PUT(y, 10)`; `V2` corresponds to `PUT(y, 20)`; and `V3` corresponds to `PUT(x, 5)`. As a result, we have:

1. Since client1 does all of the writing, `V3` causally depends on `V2`, which in turn, causally depends on `V1`.
2. Both `V1` and `V2` are visible to client2 after `E4` (i.e., when client2 receives the value of `V3`).

So, both previous versions of key `y` are visible to client2 at the end, and replica2 cannot return `y=1` in response to request `F`, as that would break causal consistency. In both examples, replica2 can respond to `GET(x)` (`V2`) even though it had not applied `V1`, only because they affected different keys.

In such a scenario, you are expected to stall for up to 20 seconds, and return an error if not successful after that time.

Causal Metadata

To enforce causal consistency, you will track causal dependencies using **causal metadata** in request and response messages. Causal metadata can take various forms. For example, vector clocks are one form of causal metadata that can be attached to messages. The details of the approach you take for tracking causal dependencies are up to you. You need to provide the description of your chosen mechanism in the `mechanism-description.txt` file.

Conflict Resolution

The causal consistency model ensures that PUT/DELETE operations take effect in causal order. However, it doesn't say anything about operations that are concurrent. If any two requests are concurrent, the 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(z, 1)` to replica1 and client2 sends `PUT(z, 2)` to replica2.
- Both requests have empty dependency lists, and can be done directly.
- replica1 applies `PUT(z, 1)`, generates version `Vx`, and broadcasts the request and the version. Concurrently, replica2 applies `PUT(z, 2)`, generates version `Vy`, and broadcasts the request and the version.
- replica1 receives the broadcast of `Vy` and applies it, and replica2 receives `Vx` and applies it. Both are allowed to do so as both PUT requests had empty dependency lists.

Now, replica1 has `z = 2`, and replica2 has `z = 1`. This is called a conflict, occurring because of concurrent writes on the same key.

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

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

You must ensure that after all updates are propagated, i.e., after a reasonable amount of time since the last update, all replicas contain consistent results. This is called **eventual consistency**, and we expect your cluster to converge after 10 seconds of no updates.

Implementation

Environment Variables

When starting up, the key-value store process should expect an environment variable `ADDRESS` of the form `host:port`, which specifies the address that the node will be reachable on. The node should exit with return value 1 if the variable does not exist or is malformed. The node will use this address to find itself in view membership lists (described below).

You should always either bind to the host specified in the `ADDRESS` environment variable or `0.0.0.0`; and the port specified in the `ADDRESS` environment variable

Node States

The key-value store nodes should allow the following states: uninitialized and initialized.

1. Uninitialized: when a node is started (or reset).
2. Initialized: when a node is part of a view/cluster.

The transition between the two states should only happen through view changes, described below.

When a node is in the uninitialized state, it should respond to all requests, other than PUT and GET requests to `/kvs/admin/view` (defined below), with status code 418 and body `{"error": "uninitialized"}`.

Replication

You should design a mechanism to replicate the state of the key-value store across the nodes. You are required to ensure the following:

1. Causal consistency.
2. Eventual consistency after 10 seconds, assuming there are no more PUT/DELETE operations and the network is healthy (i.e., messages will reach their destinations). Note that, if the network is healthy and a set of keys do not get any updates for 10 seconds, they are expected to be consistent across the nodes, even if other keys do receive updates.
3. Consistent tie-breaking between concurrent events in all nodes.

For example, you can use any of gossip, chain replication, quorum-based replication, or your protocol made from scratch. Also, it is up to you to implement the mechanism to detect that a replica is down, if you need one. You should provide the description of your chosen mechanism(s) in the `mechanism-description.txt` file.

Tie-Breaking

You can use any method to order concurrent events, or writes of the same value to the same replica. Possible approaches:

- A specific node that performs tie-breaking. This requires a means for choosing that node, and makes that node a single-point-of-failure — if that node goes down, the system breaks (at least for a while).
- Each write (PUT/DELETE) can be accompanied by the client's ID (assuming clients have some sort of ID) and the client's local time, and the key-value store can use this information to order the operations.
- You can order by the key, then by the value, then by the ID of the replica that received the request (e.g., its address), and finally by the receiving server's local time when the request was received.

API

Your fault-tolerant key-value store should support two kinds of operations: **data operations** and **view operations**. The term "view" refers to the current set of replicas that are up and running (the current cluster).

- Data operations allow the client to add a value for a key or update it (PUT), get the value of a key (GET), and delete a key (DELETE). You must ensure causal consistency between requests, as described above.

- View operations allow an administrator to change the set of nodes that are in the view (the cluster). You should not trigger view operations yourself, e.g., when a node is not reachable.

The APIs that should be supported are the following:

Endpoint URI	Request Types
/kvs/admin/view	GET, PUT, DELETE
/kvs/data/<Key>	GET, PUT, DELETE
/kvs/data	GET

Note that the `curl` requests below are examples; e.g., the number of entries in the arrays can differ from the examples.

Malformed Requests. Any request that does not contain the required JSON keys or contains "invalid contents" is considered malformed. Invalid can be general, like a string that does not parse as JSON, or context-dependent, like a number in the place of a string. You should respond to a malformed request with a status code of `400` and a JSON body of `{"error": "bad request"}`. Note that the order of the keys in a JSON object, the whitespaces in a JSON object outside the keys and the values, and extra keys (and values) in a JSON object are not considered malformed. E.g., `{"foo":"bar","baz":"quux"}` is equivalent to `{ "baz" : "quux","foo":"bar"  }`.

View Operations

PUT /kvs/admin/view. Update the view with the new view, supplied in the request body.

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"view": ["address1:port1", "address2:port2", "address3:port3"]}' \
  http://<node-address:port>/kvs/admin/view

200
```

The node that receives this request should:

1. Compare the node list in the new view with the old view, and determine the set of nodes that are removed.
2. Inform the nodes that have been removed from the view to **reset**. The nodes that have been removed from the previous view should transition to the uninitialized state. If a node is already down and unresponsive, there is no need to retry.
3. Inform the nodes that are part of the new view. The nodes that have been added should receive a copy of the key-value store state from the nodes that have remained in the view.

Note: You can assume the following:

- the node that receives the request is either starting a new cluster with no prior data, or if it already was part of a view, it is also in the new view.
- view change requests are only sent after at least 10 seconds from the last update on the data. As stated above, you should already have a consistent state across all nodes before this call. Therefore, your key-value store does **not** require causal consistency for view operations.

GET /kvs/admin/view. Return the current view of the cluster.

```
$ curl \
  --request GET \
  --write-out "%{http_code}\n" \
  http://<node-address:port>/kvs/admin/view

{"view": ["address1:port1", "address2:port2", "address3:port3"]}
200
```

DELETE /kvs/admin/view. Reset the node and transition to the uninitialized state. Do not send anything to other nodes. This endpoint is used **by other nodes** to remove a replica that is not part of the new view.

```
$ curl \
  --request DELETE \
  --write-out "%{http_code}\n" \
  http://<node-address:port>/kvs/admin/view

200
```

Data Operations

The data operations will receive and send causal metadata objects without changing them. You can design and use whatever mechanism that achieves causal consistency. To highlight this, the examples below will show an invalid value of `{OPAQUE}` as the causal metadata.

- **All data requests and responses should include a JSON object (even if empty) as the causal metadata.**
- The clients can share whatever causal metadata they have with each-other (e.g., by gossiping).
- For example, the first operation of a client will send an empty JSON object as the value of `causal-metadata`; the client will store the causal metadata received, and send it with its next request.
- Versions are not required at all — the "version" key in the possible solution (Appendix 1) are only for demonstration purposes. (You are welcome to incorporate them into your causal metadata, however.)

All data operations are expected to respect causal dependencies. When a replica is unable to answer a request while maintaining causal consistency, it should stall the request for 20 seconds, or until the inconsistency is resolved (i.e., the awaited writes have been received). If the inconsistency is not resolved within 20 seconds, the replica should respond with status code 500 and JSON body: `{"error": "timed out while waiting for depended updates"}`

PUT /kvs/data/<Key>. Add a value for a key or update it's value if it already exists. The causal dependencies of the operation must be respected. Contrary to the previous assignment, you don't need to return the previous value of a key if it already exists. The only difference is the status code: 201 for new values, 200 for updates.

- URLs have to be less than 2048 characters, which limits the keys; no further limit is needed on key lengths. If a value is larger than 8MB, the key-value store should reject the write and respond with status code 400 and JSON: `{"error": "val too large"}`.

1. If a previous value (not deleted) is not present in the causal history of the operation:

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"val": "sampleVal", "causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey

{"causal-metadata": {OPAQUE}}
201
```

2. If a previous value (not deleted) is present in the causal history of the operation:

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"val": "sampleVal", "causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey
```

```
{"causal-metadata": {OPAQUE}}
200
```

3. If the request is concurrent with other operations on the same key, either response is acceptable.

GET /kvs/data/<Key>. Return the value of the key or inform that it does not exist. The causal dependencies of the operation must be respected.

1. If a previous value is not present in the causal history of the operation:

```
$ curl \
  --request GET \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey

{"causal-metadata": {OPAQUE}}
404
```

2. If a previous value is present in the causal history of the operation:

```
$ curl \
  --request GET \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey

{"val": "sampleVal", "causal-metadata": {OPAQUE}}
200
```

3. If the request is concurrent with other operations on the same key, either response is acceptable (must be consistent across all replicas, however).

DELETE /kvs/data/<Key>. Delete the value of the key or inform that it does not exist. The causal dependencies of the operation must be respected.

1. If a previous value is not present in the causal history of the operation:

```
$ curl \
  --request DELETE \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey

{"causal-metadata": {OPAQUE}}
404
```

2. If a previous value is present in the causal history of the operation:

```
$ curl \
  --request DELETE \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data/sampleKey

{"causal-metadata": {OPAQUE}}
200
```

3. If the request is concurrent with other operations on the same key, either response is acceptable.

GET /kvs/data. Return the keys in the key-value store. The causal dependencies of the operation should be respected. For example, if it depends on all previous PUTs and DELETEs, the result should be accurate.

```
$ curl \
  --request GET \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {OPAQUE}}' \
  http://<node-address:port>/kvs/data

{"count": <number of keys>, "keys": ["key1", "key2", "key5"],
 "causal-metadata": {OPAQUE}}
404
```

Docker Operations

Here, we will create three replicas named `replica1`, `replica2`, `replica3`. Each instance will run inside a Docker container, all in a Docker network named `kv_subnet` with IP address range `10.10.0.0/16`. The IP addresses of the replicas will be `10.10.0.2`, `10.10.0.3`, and `10.10.0.4`, respectively. All instances will listen on port number 8080.

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 kv_subnet
```

Build Docker image. Execute the following command to build your Docker image:

```
$ docker build -t kvs:2.0 .
```

Run Docker containers. To run the replicas, execute

```
$ docker run \
  --net kv_subnet \
  --ip 10.10.0.2 \
  --name "kvs-replica1" \
  --publish 8080:8080 \
  --env ADDRESS="10.10.0.2:8080" \
  kvs:2.0

$ docker run \
  --net kv_subnet \
  --ip 10.10.0.3 \
  --name "kvs-replica2" \
  --publish 8081:8080 \
  --env ADDRESS="10.10.0.3:8080" \
  kvs:2.0

$ docker run \
  --net kv_subnet \
  --ip 10.10.0.4 \
  --name "kvs-replica3" \
  --publish 8082:8080 \
  --env ADDRESS="10.10.0.4:8080" \
  kvs:2.0
```

Appendix 1: Possible Solution

In this section, we describe an alternative technique that uses **explicit causal dependency lists**. You are welcome to use the described approach or find another approach, as long as it correctly tracks causal dependencies. An explicit causal dependency list contains the versions (PUT/DELETE operations) that a particular operation depends on. Note that versions are unique and each version corresponds to exactly one PUT/DELETE operation. Therefore, given a particular version, the corresponding PUT/DELETE operation can be determined.

Example 3.

1. client1 sends `PUT(x, 10)` to a replica. The causal dependency list is empty because it does not depend on any other PUT operation.

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"val": "10", "causal-metadata": {"dependency-list": []}}' \
  http://<replica-address:port>/kvs/data/x

{"replaced": false, "version": "V1",
 "causal-metadata": {"dependency-list": ["V1"]}}
201
```

The replica receiving the request first checks the causal metadata (i.e., the causal dependency list). Because it is empty, the replica knows that the request is not causally dependent on any other PUT operation, and it can be applied directly. The replica generates the unique version `V1` for the PUT operation, stores the key and the value, the version, and the corresponding causal metadata (empty in this case) in its local store, and responds to the client with a message containing the version assigned to the operation (`V1`) and causal metadata that the client needs to use in the next operation (PUT, GET, DELETE) (`["V1"]`). It also broadcasts the request and the version so that the other replicas can do the same.

Note that the client needs to use the causal metadata it receives in its next request in order to ensure causal consistency.

Other replicas that receive the request (from the replica, not the client) check the causal metadata and the version. Because the causal metadata is empty, they can apply it directly. They add the key, value, version (`V1`), and corresponding causal metadata (empty) to their local store. Notice that the version is generated by the replica that receives the request from the *client*, and the other replicas use the same version and do not generate new ones. They also respond to the replica forwarding the request with status code 200.

Note: In this example, 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.

2. client1 sends `PUT(y, 20)` to a replica (it does not have to be the same replica it sent the first request to). The client sets the causal metadata of the request to `"V1"`, which was returned from the last operation.

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"val": "20", "causal-metadata": {"dependency-list": ["V1"]}}' \
  http://<replica-address:port>/kvs/data/y

{"replaced": false, "version": "V2",
 "causal-metadata": {"dependency-list": ["V1", "V2"]}}
201
```

The replica checks the causal metadata in the request and finds out it causally depends on `V1`. The replica checks the

history of the operations it has done to find out whether it has done the PUT operation corresponding to `V1` (`PUT(x, 10)`). If so, it generates the version `V2` for the operation and stores the key, value, version (`V2`), and corresponding causal metadata (`["V1"]`) in the local store. It also responds to the client with a message containing the generated version and the causal metadata that the client should use in the next request (`["V1", "V2"]`). The replica also broadcasts the request and the assigned version to the other replicas. Otherwise, if the replica is not aware of `V1` it waits until it receives the PUT operation with version `V1` (`PUT(x, 10)`) from another replica. After receiving that request, it first applies `PUT(x, 10)` and then `PUT(y, 20)` to respect the causal dependency between the PUT operations.

The other replicas that receive the PUT operation (with version `V2` and causal metadata `V1`) from the replica that performed it must also check their history of operations to make sure that they first apply `V1` and then `V2`. They respond to the replica forwarding the request with status code 200 after successfully performing the write.

3. client2 sends `GET(y)` to a replica. The request does not contain any dependency list.

```
$ curl \
  --request GET \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {"dependency-list": []}}' \
  http://<replica-address:port>/kvs/data/y

{"val": "2", "version": "V2",
 "causal-metadata": {"dependency-list": ["V1", "V2"]}}
200
```

The replica responds with the latest value and version of the key `y` (20 and `V2`) and the corresponding causal metadata that the client can use in its next request (`["V1", "V2"]`). There is no need to forward the GET request to the other replicas. Please note that in our example, the `GET(y)` request reads the version written by `PUT(y, 20)` 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, 20)`, in which case it should return an error message indicating that the key does not exist.

Moreover, if the other clients sent other PUT requests for key `y` before the `GET(y)` request from client1, the value, version, and causal metadata returned to the client could be different.

4. client2 sends `PUT(x, 4)` to a replica. The causal metadata of the request is `["V1", "V2"]` — the causal metadata returned after the client's latest operation (`GET(y)`).

```
$ curl \
  --request PUT \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"val": "40", "causal-metadata": {"dependency-list": ["V1", "V2"]}}' \
  http://<replica-address:port>/kvs/data/x

{"replaced": true, "version": "V3",
 "causal-metadata": {"dependency-list": ["V1", "V2", "V3"]}}
200
```

The replica receiving the request first makes sure that it has already done the operations associated with the versions in the causal metadata. If so, it generates the new version `V4` and adds the version, value, and corresponding causal metadata (`["V1", "V2"]`) to its local store. Otherwise, it waits for the operations corresponding to the versions in the causal dependency list (`V1: PUT(x, 10)`, `V2: PUT(y, 20)`) and does those operations first. Finally, it broadcasts the request and the version so that other replicas can apply it.

Note that, in this example, the replica does not replace the value of the key; it maintains all the versions and values of the key.

DELETE operations. A DELETE operation can be considered a PUT with value `NULL`. For example, consider the case that client2 sends a DELETE request to a replica to remove key `y` after sending the request `PUT(x, 4)`. The

causal dependency list for the DELETE operation is therefore `["V1", "V2", "V3"]` (as received after the `PUT(x, 4)` operation).

5. client2 sends `DELETE(y)` to a replica.

```
$ curl \
  --request DELETE \
  --header "Content-Type: application/json" \
  --write-out "%{http_code}\n" \
  --data '{"causal-metadata": {"dependency-list": ["V1", "V2", "V3"]}}' \
  http://<node-address:port>/kvs/data/y

{"version": "V4",
 "causal-metadata": {"dependency-list": ["V1", "V2", "V3", "V4"]}}
200
```

The replica that receives the DELETE request checks the causal metadata and makes sure that it has completed the operations corresponding to the versions in the list before doing the DELETE. To do the DELETE operation, the replica generates a new (and unique) version `V4` and adds the version, NULL value, and casual metadata (`["V1", "V2", "V3"]`) to the store. It also broadcasts the DELETE request and the version `V4` to the other replicas.