CSC469 Assignment 3 Report

Jessica Hypatia Boritz, Wing Yan Grace Li, Yuchen Zeng

Section 1: Introduction

To understand fault tolerance in distributed applications, we implemented a distributed KV-store with replication, distributed servers, that can serve multiple clients and their requests. This report explains in detail any design decisions, implementation details, and discusses conceptual questions raised.

Section 2: Implementation & Design Choices

2.1 Basic operations

Operations performed by the client starts from obtaining the configurations from the coordinator. The coordinator provides the client ports for each server, and the client would send its requests to the corresponding server.

Servers each hold separate storage hash-tables, one for the primary replica and one for the secondary replica, and each server serves a specific key-range. The client sends their operation request to the corresponding server responsible for the specified key. A number of client-initiated operations are supported, including `GET`, `PUT` and `VERIFY`.

`OP_GET` and `OP_VERIFY`

The two work in a similar manner. Once the server receives the operation request from a client, it first reads the key requested, and determines if it is within the key-range the server is responsible for. If not, the server sends a `INVALID_REQUEST` response, indicating the request is invalid, rejecting the request. At the same time, the server also determines if the key belongs to the primary replica or the secondary replica, inserts it into the proper table, and returns the queried value to the client, in which a `VERIFY` request would return if the value provided matched the one in storage. If the key belongs to the secondary replica but recovery is not in progress, the server also rejects the request, though for a `VERIFY`, it would operate and return as normal.

`OP_PUT`

This is where replication happens. `OP_PUT` can be sent by either a client or a server, to a server. When a client sends a `PUT` request to the server, it first checks if it is responsible for the specified key. Similar to `GET` and `VERIFY`, it rejects any keys that are out of its range, or if the key belongs to the secondary replica it holds, if recovery is not in progress.

Once it is verified to be a valid request, the server writes the new key to the hash table, or, updates the existing key with the newly provided value. Then, the server forwards this exact request to the server that holds the secondary replica, we will call this server B.

To avoid a “deadlock” situation, where all servers are waiting on one another to respond to their `PUT` request, a thread is created with `pthread` to run on the background, specifically for listening and handling incoming messages from other servers. After server A, the server that received the client's request, relays the `PUT` request to server B, it blocks, waiting for server B to respond. Now, server B receives the request, and handles it in the background, while the main thread is responding to other requests from clients or the coordinator, not blocking server A indefinitely. Once server B sends out a response, server A unblocks, and depending on the status of such response, relays the correct response to the client.

Due to the multi-threaded approach for handling server-to-server communication, including replicating `PUT` requests, and heartbeat messages (described in Section 2.2, under *Heartbeat and Failure Detection*), all stored variables that are open file descriptors that are intended for outgoing messages, including `primary_fd`, `secondary_fd` and `coord_fd_out`, is protected by a mutex. As such, the `send_msg` action is in a critical section, and none of the threads would attempt to use the same open file descriptor to send a message at the same time, causing an invalid message being written into the pipe. Any operations that are written to the hash table, also are wrapped in a critical section using the provided `hash_lock` and `hash_unlock` functions, to avoid a race condition.

A successful request would require successful writes to the hash table of both server A and server B, suggesting that replication has succeeded and there are 2 copies stored. A failed request could be due to an invalid request, a request sent to a server of the wrong key-range, failing to write to the secondary replica, or an `OUT_OF_SPACE` error when writing to the hash table.

`DUMP_PRIMARY` and `DUMP_SECONDARY`
The two are also implemented, mainly for maintenance and debugging purposes. They are typically sent from the coordinator to a server, asking the server to dump all its stored kv-pairs in either the primary or secondary replica. The data dumped is written to the file `server_n.primary` for primary replica, and `server_n.secondary` for the secondary replica, where `n` is the server ID. Maintainers and admins can use this dump as a tool for debugging any errors that might have happened.

2.2 Recovery Protocol Implementation

Heartbeat and Failure Detection

To detect server failures, servers send periodical heartbeat messages to the coordinator. Since it is unwise to hog the main thread with the periodical heartbeats, a backend thread created using `pthread` is used for sending these heartbeat messages to the coordinator, consisting of the identification of the sending server.

The coordinator reads these heartbeat messages, and records the timestamp of the last received heartbeat message in the `last_heartbeat` field in the `server_node` structure. After processing all messages from servers, the coordinator goes through all server nodes, compares `last_heartbeat` with current time. If it exceeds the maximum timeout, it has identified a missed heartbeat message that was supposed to be sent. The coordinator moves on to record this missed heartbeat message, which is stored in the `server_node` structure under the field `missed_heartbeat`. Once the `missed_heartbeat` count reaches 5, the server is deemed to have failed, and the coordinator triggers the recovery process.

The recovery process is only triggered when there is no recovery in progress, aligning with the assumption of only at most 1 faulty server. It is also important to note the 5 time limit, instead of deeming a server failed after merely one missed heartbeat message. This is to avoid the possibility that there are unexpected delays, or other workload that would delay the delivery of such heartbeat messages. It is to also reduce false positives by setting a threshold higher than 1. Note also that the `missed_heartbeat` resets to 0 once a new heartbeat message is received. We also detect server failures when the coordinator breaks out of `select` due to timeout.

Coordinator

There are a few new global variables used by the coordinator to support the recovery process:

- `recovery_node`: a pointer of type `server_node`, which is `NULL` by default;
- `recovery_node_sid`: initially set to `-1`;
- `recovery_process_status`: can be `INACTIVE`, `ONGOING`, `ONE_SET_COMPLETED`, `USE_SECONDARY`, `DONE`, or `FAILED`.

To initiate the recovery process, once the coordinator detects a failed server, it calls the helper function `recover_server` with the `sid` of the failed KV server. This function then sets the global variable `recovery_node_sid` from `-1` to `sid`, changes the global status `recovery_process_status` to `ONGOING`, and creates a thread for the recovery process. This allows the recovery to be performed in the background, without interfering with how the coordinator handles client and server requests in the standard operational flows.

Initially, the thread cleans up resources utilized by the failed KV server, including the ports it was using. Then, the coordinator initializes a new server to replace the failed one, using arbitrary ports to maximize the chance of success and minimize the risk of port conflicts. The `spawn_server` and `send_set_secondary` functions have been modified to accept the memory address of the node to be spawned instead of `sid`, allowing them to be reused during the recovery process.

Then, the coordinator sends an `UPDATE-PRIMARY` message with the `sid` and port number of the recovery server, to the KV server that holds the secondary replica of the primary key set of the failed server. Upon receiving confirmation from that server, indicating it has received the message, the coordinator changes the client port and host name in the configuration for clients to match those of the server that holds the replica. The coordinator also change the `recovery_process_status` to `USE_SECONDARY`.

Similarly, the coordinator sends an `UPDATE-SECONDARY` message to the KV server that holds the primary key set corresponding to the secondary key set of the failed server. The thread then returns upon receiving confirmation from that server, indicating that it has received the message.

Now the coordinator listens for responses from both KV servers in its loop. When it receives either an `UPDATED_PRIMARY` or `UPDATED_SECONDARY` message from a server, if `recovery_process_status` is not equal to `ONE_SET_COMPLETED`, the status will be set to `ONE_SET_COMPLETED`. Otherwise, it will trigger the `SWITCH-PRIMARY` process. Since the coordinator handles server messages sequentially in our implementation, there are no synchronization concerns when updating the status.

In the `SWITCH-PRIMARY` process, the coordinator sends a `SWITCH-PRIMARY` message to the KV server that holds the secondary replica of the primary key set of the failed server and waits for confirmation, which indicates that the `SWITCH-PRIMARY` operation was handled. Following this, the coordinator updates the configuration, replacing the failed server with the recovery server. It then clears the memory allocated for the recovery node, sets `recovery_node_sid` to `-1`, and changes `recovery_process_status` to `DONE`, indicating the recovery process has been completed.

Handle Client Requests

Note that the `prepare_config_response` function, which copies the prepared configuration response into a fresh message buffer, resets `config_msg`. This occurs when `recovery_process_status` is set to `USE_SECONDARY`, where `recovery_process_status` will then be set to `ONGOING`. Similarly, this occurs when it equals `DONE`, where `recovery_process_status` will then be set to `INACTIVE`. This allows the configuration message to be updated with the most recent information for the client. This ensures that clients receive the most updated configuration when changes occur.

An alternative approach could be to update the client message in the recovery thread just once, or update hostname and port in `server_nodes` in the `prepare_config_response` function. However, since the recovery thread runs concurrently with the coordinator loop, this would necessitate extra synchronization to protect the configuration message. This contrasts with the current design, which reduces synchronization needs because `prepare_config_response` is executed sequentially. Another possible alternative is to introduce more fine-grained types for `recovery_process_status`, allowing the configuration message to be set only once. However, this could increase the code complexity and make it more susceptible to bugs during implementation.

Server

A new global variable, `recovery_process_status`, is used by servers to support the recovery process. It can be `INACTIVE`, `ONGOING`, or `SWITCH_PRIMARY_RECEIVED`. There are three messages from the coordinator that a KV server needs to handle in order to recover a failed KV server: `UPDATE-PRIMARY`, `UPDATE-SECONDARY`, and `SWITCH-PRIMARY`.

When the KV server receives the `UPDATE-PRIMARY` message from the coordinator, it will attempt to connect to the recovery server and save the connected socket fd in `primary_fd`. It will set `recovery_process_status` to `ONGOING`, and create a thread to asynchronously send its replica of the primary key set of the failed server to the recovery server. After initiating the thread, it will respond to the coordinator, indicating whether it has successfully started to recover the failed server. The thread will activate an iterator that sends the Key-Value pairs from the secondary replica one by one to the recovery server. Once this is done, it will send an `UPDATED_PRIMARY` or `UPDATE_PRIMARY_FAILED` message to the coordinator, depending on whether the recovery was successful or not.

The servers handle `UPDATE-SECONDARY` in the same manner as `UPDATE-PRIMARY`. However, they save the connected socket fd in `secondary_fd`, and send the primary key set of the current server instead. Once this process is completed, the server will send an `UPDATED_SECONDARY` or `UPDATE_SECONDARY_FAILED` message to the coordinator.

When a server receives a `SWITCH-PRIMARY` message from the coordinator, it simply sets `recovery_process_status` to `SWITCH_PRIMARY_RECEIVED`.

Handle Client Requests

Client requests are processed differently based on the `recovery_process_status`. When the status is `INACTIVE`, client requests are handled according to the standard operation flow, which allows only `OP_PUT` and `OP_GET` operations on the primary key set.

During `ONGOING` and `SWITCH_PRIMARY_RECEIVED` status, the server should also handle `OP_PUT` and `OP_GET` operations on the secondary key set, as the primary owner of these keys is undergoing recovery. It is important to note that in the server loop, messages from the coordinator are processed before client messages, and both are handled sequentially.

Thus, after the server receives a `SWITCH-PRIMARY` request and sets the status to `SWITCH_PRIMARY_RECEIVED`, all currently received client requests for secondary keys will be handled. At the end of the loop, if `SWITCH_PRIMARY_RECEIVED` is set, the status will change to `INACTIVE`. Consequently, any future client requests for `OP_PUT` and `OP_GET` on secondary keys will receive an 'invalid request' response.

Since in-flight messages from the current server to the recovery server for updated keys are synchronized while handling client requests, and because client and coordinator requests are processed sequentially, no extra synchronization is necessary for `SWITCH-PRIMARY`. An alternative to this design is to allow concurrent handling of coordinator and client requests, which could improve the system's performance. However, this approach would require additional synchronization, thereby increasing the complexity of the system.

Section 3: Conceptual Questions

In this section, we discuss the six conceptual questions raised in assignment 3.

1: If a server could run out of storage space for its keys, and we were not allowed to reject client requests, we would need to an additional server when a server became full, and then update the primary key ranges allotted to each server such that data could be moved off of the full server(s) and onto the new server. Because our primary keys are stored as hashes, we would expect a roughly uniform distribution of data across all of the servers in our system, and thus instead of only moving data from the server that has reached capacity, it would likely make more sense to redistribute the data across the entire system. We would also need to choose a server as our new server's secondary replica, and choose a server that our new server will be the secondary replica for. The easiest way to do this would be to choose an arbitrary server s, make our new server hold the secondary replicas held on s (and update the system accordingly), and then make s be the secondary replica for our new server.
Alternatively, in the case of non-uniform key distribution, we could divide the key range of the full server into two portions of roughly equal size, and move half of the data over to our newly spawned server. We could then update the secondary replicas as described above.

2: If the items stored in one server's primary replica were replicated round-robin (or otherwise in a non-deterministic process) across several secondary servers, we would have two major concerns. The first problem concerns PUT requests for keys which already exist in the system. In this case, the expected behavior is to overwrite the current value for the key in the system with the new system. However, when propagating this request to a secondary replica, we would need to ensure the request is sent to the same replica as all prior PUT requests for the same key. This is because we need to ensure we overwrite the old, no-longer valid value associated with this key with the new value: otherwise, on recovery, we may have multiple values for the same key, and be unable to determine which is the correct one. This problem could alternatively be alleviated with version numbers or timestamps, but this would be unideal (because we would be wasting space holding outdated secondary replicas for a key-value pair). Instead, we could store the sid of the server holding the secondary replica alongside the key-value pair entered in our primary hash table, and when a PUT is received for a key already in the system, we could forward the PUT request to the same replica as the prior version of the key-value pair.
When a server s_1 fails and a new server s_2 is created to replace it, the coordinator can tell each other server that it needs to send the keys in its secondary replica that are in the primary range for s_1 to s_2, and because each key is only replicated on one secondary server, we will not have any confusion or loss of data.

3: If failures can happen during recovery, what we need to do depends on the relation between the failing server and the server(s) currently being recovered.
If the failing server can be the server that was being recovered, we simply need to restart the recovery: no changes are needed to our process.
If the failing server is not the primary replica nor the secondary replica for a server being recovered, there is no concern regarding the additional failure, as it concerns data distinct from any server being recovered and thus this case can be handled alike the case of a single failure.
If the failing server is the primary or secondary replica for a server being recovered, we need more replicas: otherwise, we will lose data that can not be recovered.

4: To tolerate f failures, we need f + 1 replicas. On paper, this is because our key-value store is a replicated state machine using the fail-stop model. In practice for our system (which supports 1 failure and has 2 replicas), this is because when a server fails, we redirect requests for its primary key range to the secondary replica, and then recover the failed server from its secondary replica (and repopulate the failed servers secondary replica from the server it is a replica for). If we imagined supporting 2 failures, we would clearly need 3 replicas: on failure, we could restore from either replica. Because we are assuming fail-stop, and we do not deem a PUT successful until it has been replicated onto our replicas, we can assume that both of our replicas will contain identical data, so it does not matter which one we restore from.

5: If our key-value store was changed to an eventual consistency model, we would need to accept that the system could lose some data on failure, as a PUT request could return success to a client but the server that sent the request could fail before the PUT was propagated to the secondary replica. With this in mind, we would want to reduce how much potential data loss could occur on a failure. One potential method of achieving this would be to perform checkpointing, either after a number of operations have been performed or after a period of time has passed. In this approach, when we checkpoint we would ensure that all of the data currently input was replicated before we proceeded to accept more `PUT`s. Another potential method would be to allocate a fixed period of time where the `PUT` could be eventually replicated, and if this period was exceeded before the `PUT` was replicated, we could then actively replicate the `PUT` on the critical path, before performing more operations.

6: If we allowed the number of nodes in our key-value store to scale upwards and downwards with system load, we would need to implement processes that redistributed the primary key space and secondary replicas when nodes were added or removed. This could be done via the process described in question 1, extended to both scale upwards and downwards: when increasing the number of servers, we shrink the primary key space allocated to each server so that we can assign key space to the newly added node, and then must have the servers move the key-value pairs associated with the space that they have given up to the new node. When decreasing the number of servers, we pick a server to remove, and distribute its primary key space (and any key-value pairs stored in the server) across the other servers in the system.
To ensure that a newly created server has a secondary replica, we could pick a server `s`, and have our newly created server become the secondary replica for the server that `s` was currently the secondary replica for (and move over the secondary key-value pairs). We would then have `s` act as the secondary replica for our newly created server, and migrate all of the secondary replicas for key-value pairs that belong to the primary key space of our new server to `s`.