CS 6210: Advanced Operating Systems Apr 23, 2023

Project 4 Report

Instructor: Ada Gavrilovska *Aditya Vikram*

1 Abstract

This report provides an overview of the design and implementation of GTStore, a distributed key-value store where RPC is used to store and replicate key-values over multiple storage nodes. I also present the results for throughput and load-balancing capabilities of the system and discuss the performance results.

2 Introduction

The objective of GTStore is to create a key-value store that is distributed over multiple servers. This ensures that when an assigned primary node for a key goes down, the clients can contact one of the secondary nodes with the replicated value, and the system continues to be up. There is a single server (called manager) that manages the control flow from clients and handles failure of any of the storage nodes.

The manager, storage nodes and client communicate via gRPC ([1]) calls. This means that even though the servers are hosted on a single machine for the tests, GTStore is easily extensible to run the manager and storage nodes on different machines.

The figure below shows an overview of the various API interactions between different GTStore components.

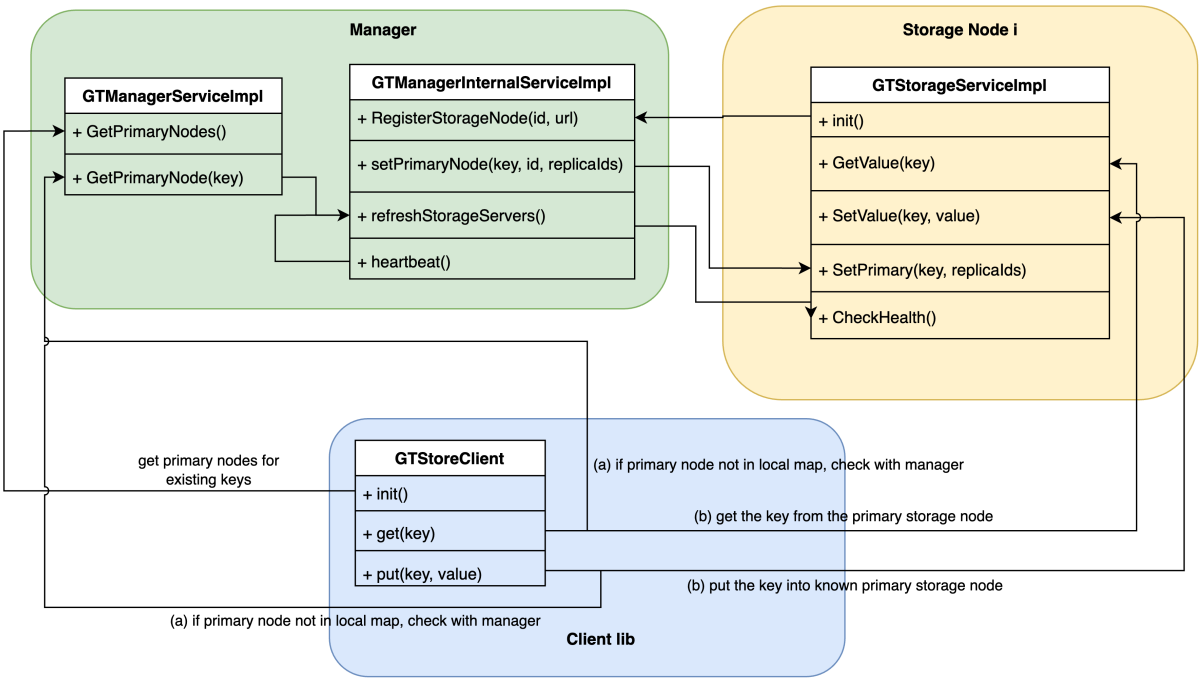

Figure 1: An overview of main GTStore components

The design is described in the next section.

3 Design

There are three types of processes in the system:

1. Manager: this is the centralized manager process which runs two gRPC servers waiting for requests on individual threads:
   - `GTManagerServiceImpl` for public facing client lib APIs
   - `GTManagerInternalServiceImpl` for APIs used only by other storage nodes
2. Storage: there is one storage process per storage node required. Each process runs a gRPC server implemented by `GTStorageServiceImpl` that exposes the get/put APIs
3. Client app: this statically links the client lib which interacts with the manager and storage servers to get/put key-values required by the client app

Now we go over each of these processes and their services in detail.

3.a Manager

When the manager process is started, we first create an instance of `GTManagerInternalServiceImpl`, and use that to initialize an instance of `GTManagerServiceImpl` because the storage nodes information is stored by the internal service. Then both servers wait on their individual threads and the manager process waits indefinitely by joining both threads.

3.a.1 GTManagerInternalServiceImpl

This is the internal gRPC service running at "0.0.0.0:50050" (unknown to client lib, there would be more security in a real application like checks if caller is indeed a storage node) used by other storage nodes. The class stores the information about the id and URL of each registered storage service. When a storage process is spawned, the initializer registers the storage node using the `RegisterStorageNode()` API exposed by this gRPC service. This in turn creates a `GTStorageServiceClient` object (discussed in 3.a.2) and stores that in a map corresponding to the id of the calling storage node.

Additionally, this service also has a heartbeat thread that periodically (every 250ms) iterates through all known alive storage nodes, and checks if they are still alive using the storage service's `CheckHealth()` API. If any storage node is down (gRPC error code `UNAVAILABLE`), then a load-balancing algorithm is invoked (discussed in 3.a.5) to reassign primary nodes for all the keys that were managed by the failed node.

3.a.2 GTStorageServiceClient

This is a gRPC client class that encapsulates the functionality required to contact the storage service implemented by `GTStorageServiceImpl`. It simplifies the code by exposing function calls that abstract out all the gRPC request-response and deadline handling. This common class is used both in the manager process and the client lib, since both are clients for the storage service.

3.a.3 GTManagerServiceImpl

This class implements the client-facing gRPC manager service which has APIs to either get the primary storage node for all existing keys (`GetPrimaryNodes()`) or for a specific key (`GetPrimaryNode()`). The former is used by the client lib on initialization, and the latter is mainly used when the client lib doesn't have the primary storage node information locally, or the known primary storage is unavailable due to a failure.

To do this, this class maintains a map of all known keys to a struct that contains the primary node id, and a vector of all replica node ids (nodes where the key-value pair is replicated). Note that we only maintain the ids in this class to avoid data structure duplications. The client object `GTStorageServiceClient` is only stored in the map inside `GTManagerInternalServiceImpl`, and `GTManagerServiceImpl` holds a reference to the internal service.

When a new key needs to be assigned a primary and replica nodes, the data partitioning algorithm described in 3.a.4 is invoked to determine the node ids. Then using the `GTManagerInternalServiceImpl` reference, we call the `SetPrimary()` API on the storage service corresponding to the decided primary node. Then we return the decided primary node id as a response to the client lib's gRPC call.

3.a.4 Data partitioning

The goal of our data partitioning scheme is to evenly distribute the load for storing and replicating keys over the given storage nodes. Since we don't have any prerequisite information on the probability distribution of keys used by the clients, it makes sense to not rely on the lengths of keys to distribute the key responsibilities. Another possibility was to rely on the hash of the keys to map them to a storage node, but that requires additional hash computation. We can achieve the same level of load-balancing by using a Round-Robin assignment of primary storage nodes, which is used in this implementation of GTStore.

For the first key, we assign it to the first storage node (in the order of storage node registrations) as primary and nodes $2, \cdots, k$ as replicas where k is the number of replicas specified. For the next key, we assign node 2 as the primary and nodes $3, \cdots, k+1$ as replicas, and so on. With this data partitioning scheme, with K keys, each storage node manages either $\lceil \frac{K}{n} \rceil$ or $(\lceil \frac{K}{n} \rceil - 1)$ keys.

This is as even a load-distribution as one could hope to achieve without knowing the probability distribution of the keys in advance. A scheme using hashing would also achieve this level of load balancing, if the keys used were chosen at random.

3.a.5 Load-balancing

When a given storage node i goes down, we iterate through the map of all keys, and:

- for each key that had node i as a replica, remove it from the key's replica vector and update the primary storage about the replica update
- for each key that had node i as a primary: iterate through the vector of replica nodes and find the node that is the primary for least number of keys. That node gets assigned as a primary for this key now.

As we iterate through the map of keys, this greedy policy allows us to:

- efficiently reassign primary nodes for the keys managed by the lost storage node, leading to quicker update when client lib asks for a new primary node; and
- maintain an almost-even load-distribution among the storage nodes

A minor drawback of this scheme is that it is slightly slower than blindly assigning the first replica node for each key that needs reassignment, because we need to iterate through the list of replicas to find the one with minimum managed keys. But that trivial scheme would lead to a load imbalance because all keys managed by the failed node would potentially end up on the same replica storage, thereby putting more load on that specific storage node. Furthermore, to implement our scheme efficiently, a map of node ids to the number of keys managed is maintained throughout the lifecycle in `GTManagerInternalServiceImpl`. This takes a few cycles per every new key, but helps in finding the least-loaded replica during load-balancing.

3.b Storage

When the storage node starts with an id and service URL as arguments (we assume these are unique and controlled by the test or deployment scripts), we create an instance of `GTStorageServiceImpl` which implements the storage service. This class has a `GTManagerInternalServiceClient` object (similar to `GTStorageServiceClient` from 3.a.2 used for abstracting the API calls to internal manager service) that is used to call `RegisterStorageNode()` API. This is an async gRPC API call which blocks until all storage nodes (equal to `--nodes` argument for manager) have registered with the manager. Then the API's response contains a list of the id and URL for all storage node peers. This is stored in a local map from id to `GTStorageServiceClient` for later use.

For storing the key-values, `GTStorageServiceImpl` uses a map of key to a struct `GTStoreValue` which contains:

- value
- `isPrimary`: whether this node is the primary for this key. Multiple storage nodes replicate a key-value pair as discussed in 3.b.1
- `replicaNodeIds`: vector of ids of all the replica nodes, if this node is indeed the primary node

For a `GetValue()` call, we just return the value corresponding to request key, if it exists in the map. When `SetPrimary()` gets called, we create an entry in the map with empty value, `isPrimary = true` and `replicaNodeIds` initialized with the replica nodes in the request. *Here I acknowledge the security concern that a malicious client could call this storage API if they knew the service URL and mess with the primary nodes for a key. If security was a concern, we'd have to separate this call into an internal storage service and have more checks. I chose to skip that given the time limitations for this project.*

For a `SetValue()` call:

- if the key already exists in the map, then we just update the value in local map. Then if this node is the primary for the given key, iterate through `replicaNodeIds` and call `SetValue()` on each replica storage node to update their copies
- if the key doesn't exist, we create an entry in the map with `isPrimary = false` and `replicaNodeIds` as empty. This would be the case when a primary storage node would call `SetValue()` on a replica node, because otherwise the manager would've already called `SetPrimary()` for the key and it would exist in the map.

3.b.1 Data replication and consistency

As mentioned in the previous section, we use an immediate replication scheme. Whenever a value is set for a key, we immediately put/update that value on all the replica nodes also. This leads to a strict consistency model as there is a single primary node used by the clients, and before the `SetValue` call completes, all replicas have been updated "atomically". This is because `SetValue` and many other API implementations in `GTStorageServiceImpl` lock a mutex to denote control over the key-`GTStoreValue` map. So only one of multiple `SetValue` calls can complete at a time.

3.c Client lib

The `GTStoreClient` class exposed from our client lib (statically linked in any client app) has the following functions:

- `init()`: meant to be called when the app wants to initialize the connection to GTStore. This creates a `GTManagerServiceClient` instance (used to call manager's service APIs) and calls `GetPrimaryNodes()` to get the node ids and URLs corresponding to existing keys in the store. A key-nodeId map is initialized with this data. We also store a similar map from node id to `GTStorageServiceClient` so that we can call the corresponding storage node's APIs when needed.
- `get(key)`: If the primary node id exists for the given key in the local map, we use the corresponding `GTStorageServiceClient` instance to call the primary node's `GetValue()` API and return the result. If the key doesn't exists, then we call the manager's `GetPrimaryNode()` API with the given key to fetch the assigned primary node id. This logic is wrapped in a retry loop (max retries configurable by client, defaults to 3) that erases the primary node id in the map if `GetValue()` call fails within a specified deadline (500ms), assuming that the storage node has failed. In the next retry, because the key doesn't exist in local map, we'd end up contacting the manager to get the updated primary node.
- `put(key, value)`: Similar to the get call, we first ensure that the primary node for the given key is known, either from the local map or by contacting the manager. Then we call `SetValue()` API for the

primary node and return the primary node id for the key, -1 if failed. This logic is again wrapped in a retry loop (max retries configurable by client, defaults to 3) that erases the primary node id in the map if `SetValue()` call fails, assuming that the storage node has failed. In the next retry, because the key doesn't exist in local map, we'd end up contacting the manager to get the updated primary node.

- `finalize()`: this is a no-op, all classes cleanup automatically in their destructor so we don't need to explicitly cleanup. There is a use case if client app wants to close the connection to GTStore services early, but I chose to skip this scenario in the interest of time as all demo tests have single get-set per test app invocation.

4 Performance Results

We have two test apps in the submitted code: `test_app` for demo purposes which takes either `--get <key>` or `--put <key> --val <value` arguments to interact with GTStore. For performance results, we have `perf_test_app` that has two options:

- `--throughput`: Initializes `GTStoreClient` and performs 100k `put()` calls, each followed by a `get()` call, so total 200k read-writes. Then we measure the total time taken and calculate the throughput
- `--lb`: Initializes `GTStoreClient`, performs 100k `put()` calls, maintains the number of keys managed by each storage node and prints that finally.

Firstly for load-balancing, as seen in Figure 2 we see an even distribution of the number of keys managed by each storage node. This is as expected, for the reasons given in Section 3.a.5.

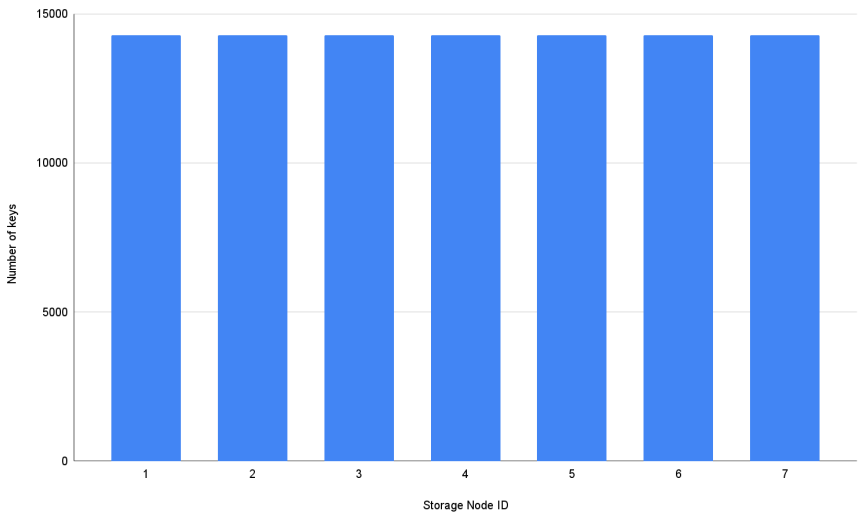

Figure 2: Load across storage nodes for 100k keys

The actual numbers of the number of keys managed by storage nodes are

$$14286, 14286, 14286, 14286, 14285, 14286, 14285$$

which depends on the order in which the 7 storage processes got spawned and registered with the manager. But there is an optimal distribution of keys across the seven storage nodes.

A bar chart of throughput as the number of replicas varies is shown in Figure 3 (left). First and foremost, we see an incredibly high value of throughput: ~ 659 ops/s for 1 replica on the higher end. Even for 5 replicas, we have a throughput of ~ 458 ops/s which is great (2ms per operation!) considering that we have fault tolerance of up to 4 storage node failures (for 5 replicas). This is a classic *trade-off of efficiency of the system*

for more reliability via replication. Throughput of the system decreases as the number of replicas increases, because for each put call, the primary storage node has to call an increasing number of replica storage nodes, which adds to the operation time.

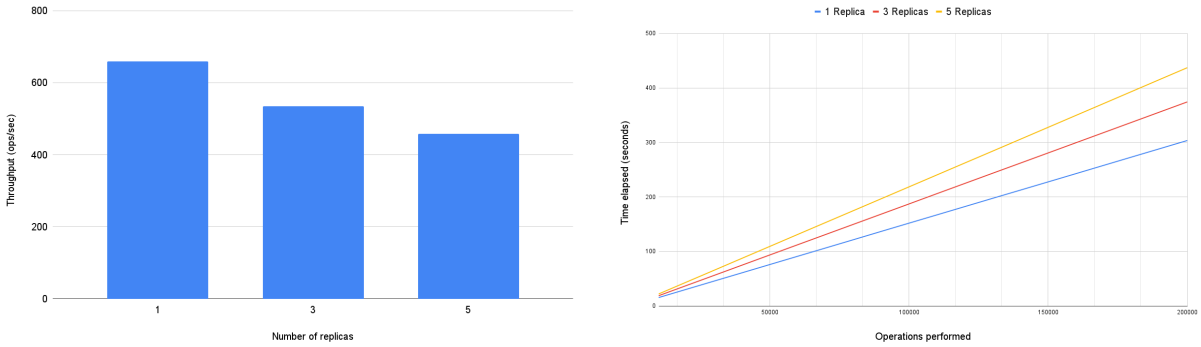

Figure 3: Left: throughput for 7 storage nodes vs number of replicas
Right: time elapsed vs number of operations performed for 1, 3 and 5 replicas

To get a closer look, the right plot in Figure 3 shows how the elapsed time varies linearly with the number of operations performed as we vary the number of replica nodes. And with increasing number of replicas, the total time taken goes from around 303 seconds for 1 replica to 374 seconds for 3 replicas and on to 437 seconds for 5 replicas. As expected, the time taken scales linearly with the number of replicas, adding around 0.18 ms per operation per new replica added. So we would expect an hyperbolic decrease in throughput ($\frac{1}{\text{time per operation}}$) as number of replicas increase further beyond 5.

5 Conclusion

We have described a distributed key-value store in this report, which can efficiently respond to requests and maintain a balanced load of the keys managed by its storage nodes. We also showed how the system handles failures of nodes by polling periodically and load-balancing if any of the storage nodes fails to respond. The performance results show that our system is lightning fast (2ms per operation for 7 storage nodes and 5 replicas) and the throughput scales inversely with the number of replicas. The sweet spot would depend on the requirement for fault-tolerance: if we expect only up to 2 storage nodes to fail, having 3 replicas is sufficient and faster.

What more could be done? Firstly there are some security flaws for deployment in the real world. As mentioned in the design sections, there are no checks that the calls are indeed coming from storage nodes and not a malicious endpoint. Another improvement would be an implementation of `finalize()` that closes all connections to GTStore early and cleans up the local maps, in case the client app wants to stop using GTStore and focus its resources elsewhere. Another avenue of improvement could be in manager-storage heartbeat mechanism: using gRPC health checking protocol that allows a streaming-type API where the manager can get notifications about the change of the storage server health status. But this requires allocating potentially 1 thread per storage node in the manager process to observe each storage node. This would work well for systems with lesser number of storage nodes but would cause resource contention when deployed on a larger scale.

References

[1] "grpc homepage." https://grpc.io/.