Project 4: GT Key Store

CS 6210: Advanced Operating Systems

Akshat Deo (adeo37)
903944449

Dhruv Rauthan (drauthan3)
903943882

Introduction

This report describes the design and implementation of GTStore, a distributed key-value store. GTStore is designed to provide a scalable, reliable and consistent key-value storage system while keeping principles of data partitioning, data replication and data consistency in mind. This report details the design, implementation, and performance results of our implementation.

System Components

Our implementation of GT Store comprises of three main components: a Manager, which oversees storage nodes and ensures load balancing and fault tolerance; Storage Nodes, which handle the actual key-value storage and replication and a client application, which allows users to perform Put and Get operations on the key-value store. Communication between these components is implemented via gRPC.

Manager

When the GTStore service is spawned, a manager process is created that is responsible for handling all the storage nodes. It takes the number of storage nodes and replicas for each key as its command line arguments. Its member variables and functions are described below:
Variables:

- **unordered_map<string, StorageNode> key_to_primary_node:** Maps the key to the primary node ID and address (saved together in the StorageNode struct)
- **unordered_map<string, vector<StorageNode>> key_to_replica_nodes:** Maps the key to a list of replica nodes

- **unordered_map<int, string> node_id_to_address:** Maps the node ID to the URL address of the node, where the storage gRPC server is running
- **unordered_map<int, int> node_id_to_key_count:** Tracks the count of keys for which a node is the primary, helps in load balancing

Functions:

- **void init():** This is the base initialization function. A gRPC service GTManagerServiceGRPC is started, which listens on the localhost port 4444 ("0.0.0.0:4444") for any storage node registration or client put/get requests. It also spawns another thread, the PollNodes(), which keeps running in the background.
- **void SetPrimary(string key, int primary_id, vector<int> replica_ids):** Responsible for mapping the primary node ID and the replica node IDs to the key and sending this information in a gRPC message to the primary storage node via the PrimaryAllocation() function which is described in the Storage section.
- **void PollNodes():** A background thread which sends an empty gRPC message to all the nodes to check if they are alive or not. This is done every 100ms. If a node is down, it appropriately handles the failure.
- **void HandleNodeFailure(int node_id):** Goes through all the keys for which the node was the primary node and reassigns the primary. This is done by choosing the node which is the primary for the least number of keys, ensuring proper load balancing. It also goes through the keys for which the node was a replica and chooses another replica node for that key. Afterwards, the SetPrimary() function is called to communicate updates to the storage nodes. The failed node is then removed from all the maps and lists in the manager program so that it cannot be used for further operations.
- **Status GetPrimaryNodeForGet(ServerContext* context, const KeyRequest* request, NodeResponse* response):** This is a gRPC service function which returns the storage node details (the node ID and its address) to the client requesting the value for a particular key. If the key does not exist, it responds accordingly.
- **Status GetPrimaryNodeForPut(ServerContext* context, const KeyRequest* request, NodeResponse* response):** If the request key already exists, then the manager returns the storage node details of the corresponding primary storage node for that key. Otherwise, it assigns a new primary node in a round-robin fashion, which again, ensures proper load balancing. The replica nodes are also chosen as the next k nodes after the primary node, where k is the number of replicas. SetPrimary() sends this

information to the storage nodes and then the primary node details are sent to the client. Since this is happening in a synchronous blocking manner, the client can only connect to the node after it sends back an acknowledgement after learning it is the primary node for that particular key.
- **Status RegisterStorageNode(ServerContext* context, const StorageNodeRequest* request, RegistrationResponse* response):** Used to save the storage node information sent in the request message in the manager's local maps.

Storage

The storage node program takes the port it is supposed to run on as a command line argument. This should not be port 4444, since the server is running on that port. Its member variables and functions are as follows:
Variables:

- **unordered_map<string, string> key_value_store:** The actual key value storage system
- **int node_id:** A unique node ID used to identify the storage node, it is equal to the process ID
- **string address:** The URL address on which the gRPC service is running
- **unordered_map<string, bool> key_is_primary:** Used to track which keys the current node is a primary for and to notify the replicas accordingly
- **unordered_map<string, vector<StorageNode>> key_to_replicas:** Maps the key to the appropriate replica storage nodes

Functions:

- **init(int port):** Initializes the address and node ID and calls the RunStorageServer() function
- **void RunStorageServer(string address, string manager_address, int node_id):** FIrst calls the RegisterWithManager() function which is described below. This function also starts the gRPC service listening on the earlier specified port. This service waits and listens for manager or client gRPC messages
- **RegisterWithManager(string manager_address, int node_id, string storage_address):** Creates a new gRPC channel which connects to the service running on the manager. It calls the RegisterStorageNode() gRPC function and sends its own node ID and address to the manager.

- **Status GetValue(ServerContext* context, const KeyRequest* request, ValueResponse* response):** A client side function, it returns the value associated with a particular key. The key will always be found since the manager will only give the address of the primary node which contains that key
- **Status SetValue(ServerContext* context, const KeyValueRequest* request, AckResponse* response):** This function first sets the value of the key in the local map. Then, since it is the primary node for that particular key it also has to update all the other replica nodes for that key. It iterates through these replica nodes and calls the same SetValue() function on the other replica nodes. This happens in a blocking manner, so when this function returns an OK status to the client, all the other replicas already have the updated value for that key.
- **Status Poll(ServerContext* context, const EmptyRequest* request, PollResponse* response):** Just returns a response message to the manager to let it know that it is still alive
- **Status PrimaryAllocation(ServerContext* context, const PrimaryRequest* request, AckResponse* response):** Sets the member variables accordingly for that key (true for key_is_primary and clears the previous replicas for that key, if any). It also adds the replica StorageNode structures in the storage's local map, which are referenced during the SetValue() function, when the primary node needs to update the replica nodes.

Client

The client application is used to set, update or get values from the GTStore application (the manager and storage nodes combined). We assume that it already knows the port on which the manager is listening for new gRPC requests (4444). The put/get commands are taken via the command line. The member variables and functions are described as:
Variables:

- **int client_id:** The unique client ID (same as the process ID)
- **unique_ptr<GTManagerService::Stub> manager_stub_:** A gRPC stub used to connect to the manager service. This is used to call the functions defined on the manager.

Functions:

- **void init():** Sets the client ID

- **bool Put(string key, string value):** First, this function gets the primary node for the particular key from the manager using the GetPrimaryNodeForPut() function from the stub. Subsequently, it connects to the primary node using the returned node address and calls the SetValue() function on the storage stub to save the value in GTStore. On success, it returns true
- **string Get(string key):** Similar to the Put() function, it gets the primary node from the manager. However, if the key does not exist in GTStore, it immediately returns false. Otherwise, it connects to the primary node and gets the value for the particular key from the primary node.

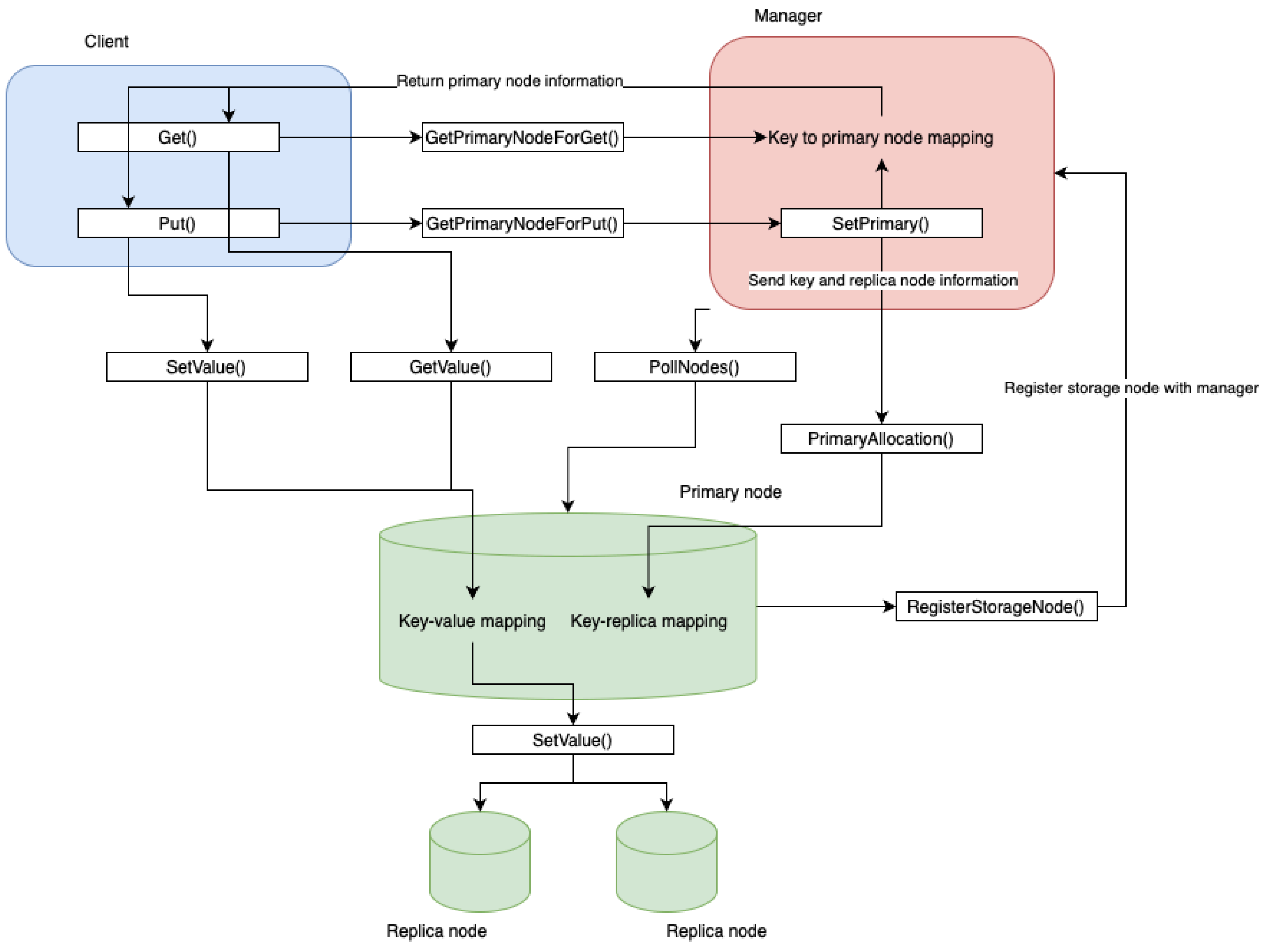

Figure 1: Process flow for GTStore

Design Principles

Data Partitioning

Our GTStore system partitions data across multiple storage nodes to ensure scalability and load balancing. The manager assigns each key a primary storage node using a round-robin approach, which helps us evenly distribute the data across nodes. The manager maintains a mapping of keys to their respective primary storage nodes and tracks the number of keys assigned to each node. This approach ensures that no single node gets overloaded, preventing any performance bottlenecks. As shown in figure 3 we observe that our data gets evenly balanced across all available storage nodes.

Data Insert/Lookup Protocol

1) **Put():** When a client sends a Put() request, the manager assigns a primary node for the key using the round-robin scheme. The manager then updates its internal mapping of keys to the assigned primary storage node and communicates the assignment to the selected primary node. If multiple replicas are enabled, the manager also selects replica nodes in a round-robin fashion and informs the primary of their details. This information about the primary gets sent to the client, which transmits the value to the primary node. The primary node then propagates the data to the replicas to ensure consistency.
2) **Get():** For a Get() request, the manager checks its key-to-node mappings to identify the primary node responsible for the key. It communicates information about this primary node to the client, which allows the client to directly retrieve the value from the storage node.

This round-robin algorithm allows our system to scale easily, adding more storage nodes is trivial. It also allows us a simple way to load balance our keys, without having to perform potentially expensive hashing operations every time a key is added. This system may face performance challenges if many frequently accessed keys get assigned the same primary node, potentially causing a node to face higher load than others.

Data Replication

Our GT Store system replicates each key-value pair across multiple storage nodes to ensure reliability and fault tolerance. Each key-value pair is stored on a primary node and replicated across a configurable number of replica nodes. The manager, when assigning a primary node for a key, also selects a set of replica nodes in a round-robin fashion to hold copies of the data. This replication protects our GTStore service against node failures. If a primary node fails, the manager can quickly promote one of the replicas to become the new primary, ensuring uninterrupted service.

Data Insert/Lookup Replication Protocol:

1) **Put()**: Following the description in the above section. When a client sends a Put request to the manager, the manager checks if the key already exists. If yes, it retrieves the primary node and forwards the client's request to it. If not, it assigns a primary node and replica nodes for the key, and communicates this information to the primary node.
   The primary node then stores the key-value pair in its local data store and propagates the data to the replica nodes. Each replica node then updates its local storage and acknowledges the update back to the primary node. Once all replicas confirm, the primary node responds to the client, indicating the operation was successful. Note here that a Put() operation is not successful until all the nodes are updated with the correct value.
2) **Get()**: For a Get() request, the manager checks its key-to-node mappings to identify the primary node responsible for the key. It communicates information about this primary node to the client, which allows the client to directly retrieve the value from the storage node.

Our GT Store effectively handles node failures within the selected round-robin partitioning approach. The manager continuously monitors storage nodes using the PollNodes() thread, which sends a message to all nodes every 100ms and expects an OK response. If a primary Node fails, the manager identifies the failure, retrieves the associated key-value pairs, and reassigns the keys to the storage node with the fewest keys to maintain load balancing. After reassigning a new primary node, the manager selects a new set of replica nodes for the affected keys. This process is similar to a basic Put() operation for each key managed by the failed

primary node. The manager then updates its internal key-to-node mappings to reflect the new primary and replica assignment.
For each replica previously held by the failed node, the manager assigns a new replica to ensure the total number of replicas remains consistent. The manager then communicates this updated replica assignment to the primary node, ensuring future updates can be propagated accordingly.

Data Consistency

Our implementation follows a strong consistency model, meaning all clients will see the same version of data at any given time. Consistency is enforced by blocking the client until the value has been successfully updated on all nodes. When a client issues a Put() request, the manager assigns a primary node and replica nodes for the key. The primary node stores the key-value pair locally and propagates the update to all replica nodes. The Put() operation is only acknowledged to the client after all replicas have successfully stored the key-value pair, ensuring that the data is consistent across all nodes.
We employ an immediate replication scheme to uphold strong consistency. This ensures that all inserted objects are replicated to the designated nodes within a bounded timeframe. If a failure occurs during the replication process, the operation is marked as unsuccessful, and the value remains unchanged. This approach guarantees that clients always access the latest value, maintaining data consistency.

Design Tradeoffs

1) Consistency vs Latency: Our implementation enforces a strong consistency model, ensuring all clients see the same version of data. Put() operations are only acknowledged when all replicas have been successfully updated. While this guarantees data consistency, it introduces additional latency (especially for write-heavy workloads). We believe this is a fair tradeoff to make, given the unknown nature of the workload.
2) Round-Robin Partitioning: We implement key distribution using a round-robin approach. This allows us to keep the Put() operations relatively simple, however, this may cause more load on a particular node if its keys are heavily accessed.
3) Polling Frequency: The manager's PollNodes() function polls all the storage nodes every 100ms to detect failures. This allows us to perform recovery relatively quickly, however it

adds some network overhead. The polling rate should ideally be set based on the frequency of failures.

Implementation Issues

Node failures are handled by the HandleNodeFailure() function, which is called by the PollNodes() thread upon detection of a failure. If a node hosting a primary replica for a given key goes down while a simultaneous put operation arrives, it could lead to an undefined state. However, we believe 100ms is a reasonable enough time frame to assume that no Put() operations will occur between the node's failure and recovery

Performance Tests & Graphs

We tested our code on the Advos VM provided, and it successfully compiled and ran without any issues. However, compiling the code on the VM requires a modified 'CMakeLists.txt' file to ensure it correctly finds gRPC and protoc libraries on the system. We have included both files - CMakeLists.txt (for compiling code on the Advos VM) and CMakeLists_Local.txt (for our local machines). The README contains additional information that may be required to run the code.

For the performance tests, we executed our code on a system equipped with 8 CPU cores 32 GB of memory. However, these tests were run on Windows Subsystem for Linux (WSL), which may not offer native performance. The results obtained are as follows.

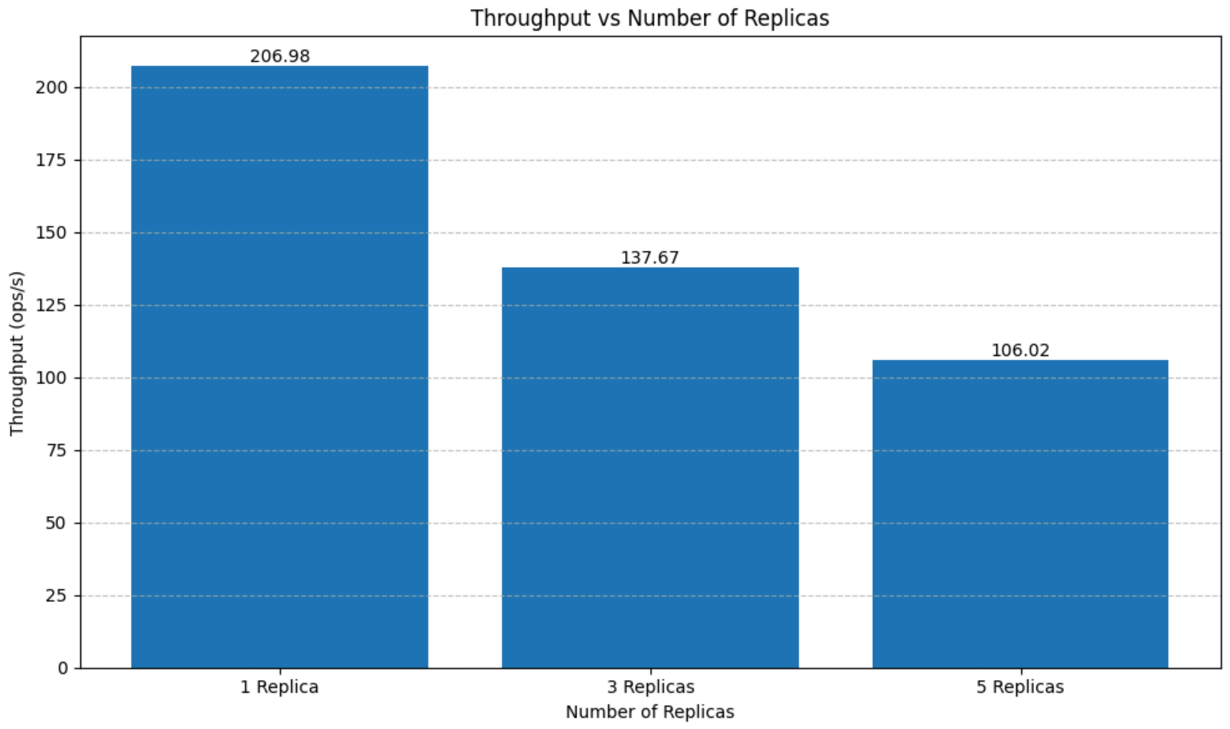

Figure 2: Throughput as number of replicas increase

The above bar graph shows us how our throughput scales across multiple replica nodes. The tests were conducted with 7 Storage nodes and varying number of replicas (1,3 and 5). Each test executed a total of 200,000 operations with a mix of 100,000 PUT and 100,000 GET operations.

The results demonstrate a throughput of ~207 operations per second in the case of a single replica. As expected, throughput decreases as the number of replicas increase, with 3 replicas netting a throughput of ~138 ops/s and 5 replicas yielding 106 ops/s. Adding more replicas increases the communication overhead for the manager, which now has to coordinate with multiple storage nodes to store keys. Notably, Get operations take significantly less time than Put operations, since Put requires a new node to be assigned and data to be replicated across multiple nodes.

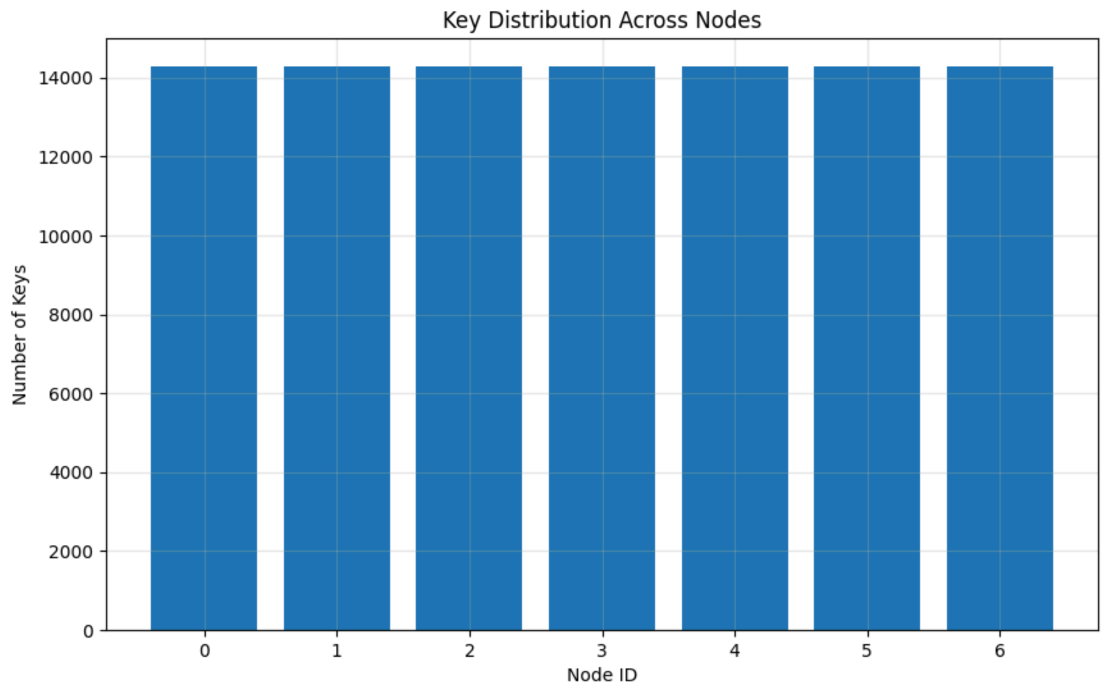

Figure 3: Distribution of keys across the different storage nodes

The Bar graph above shows the distribution of keys across the 7 storage nodes. We observe a nearly even allocation, with nodes 0 and 4 storing 14285 keys each, while the remaining nodes store 14286 keys each, totalling up to 100,000 keys. This even distribution demonstrates that our system load-balances keys as intended, ensuring that the keys are all uniformly distributed across nodes.

Task Division

Both of us worked together on the GTStore implementation. There was no specific task division.

References

1) https://medium.com/@shradhasehgal/get-started-with-grpc-in-c-36f1f39367f4
2) https://grpc.io/docs/languages/cpp/quickstart/