

Project 4: GT Store

Mohit Gupta (#903762237), Himanshu Goyal (#903843623)

April 2023

1 GT Store Overview

Our team designed a fault-tolerant replicated GT Store, a distributed system that stores key-value pairs. To build this system, we utilized Google's Remote Procedure Call (gRPC) framework, which provides a way for different components of the system to communicate with each other efficiently.

Our primary goal during the design process was to ensure that the system could handle a high volume of requests and provide reliable access to data. To achieve this, we focused on three critical factors: scalability, availability, and reliability. We believe that our resulting system can handle large amounts of data traffic while remaining operational in the face of failures and providing accurate and consistent data to users.

2 System Components

2.1 GTStore Centralized Manager:

In our design, the centralized manager acts as an orchestrator. The manager process is initiated first with two parameters - the number of storage nodes and the replication factor. Based on the provided parameters, the manager forks the required number of storage node processes and sets up a communication path with each.

The manager process exposes two RPC interfaces for communication. One is for the client, and the other is for the storage nodes. It functions as both a client and server process simultaneously. When a GT store client requires control information, it connects to the manager process and requests it. Therefore, the manager process acts as a server for the GT Store client. On the other hand, to maintain the active state of all the storage nodes, the manager process sends keep-alive ping messages at a periodic intervals. In this case, the manager process acts as a client.

1. **KeepAliveCheck():** To ensure the continuous operational state of the storage cluster nodes, the manager periodically sends a keep-alive ping message to all of the nodes and updates the relevant information based on the responses received. The manager always handles GET and PUT requests from the client in accordance with this updated information.
2. **Provide_K_Nodes(key):** Whenever a client wants to perform a PUT operation, it connects to the manager process to obtain control information. Based on the health status of the storage cluster, the manager process determines the required number of storage node IDs using our deterministic data partitioning scheme (described later) and returns the list to the client. The client then connects to the specified storage nodes and uploads the data.
3. **ProvideNextPrimaryID(key):** In our design, when fulfilling a GET request, the client first attempts to connect to the primary replica. However, if the connection fails due to a primary replica failure, the client calls this API as a protective measure to obtain updated information about the particular key. The manager process serves the request by providing the new primary storage node responsible for the key.

2.2 GTStore Storage Node:

Our storage node process always functions as a server process, providing GET and PUT Remote Procedure Call (RPC) interfaces to the GT store client. The storage node initializes a hash map data structure and uses it to store key-value pairs. We chose the hash map data structure as it provides constant time lookup, i.e., $O(1)$.

In our design, when storing or fetching a key-value pair, the client connects directly to the storage nodes based on the control information provided by the manager. Hence, the manager process never acts as an intermediary for any data retrieval or storage operations. It resulted in better efficiency of the overall system.

1. **GetReply(key, Value):** The storage node searches for the requested key in its hash-map data structure. If the key is found in the hash map, the storage node returns the corresponding value. If the key is not found in the hash map, the storage node returns a lookup failure code.
2. **PutReply(key, Value):** If the storage node encounters the key for the first time, it adds a key-value entry to the local storage. Otherwise, if the key already exists in the local storage, the storage node updates the value of the existing key with the new value.

2.3 Client API calls (one-by-one):

Our design assumes that the client process is a stateless process. Therefore, whenever the client process is launched, it connects to the manager process to obtain the necessary information required to communicate with the storage nodes. The overall working of the client process for GET and PUT requests can be understood through the control flow shown in Fig. 1.

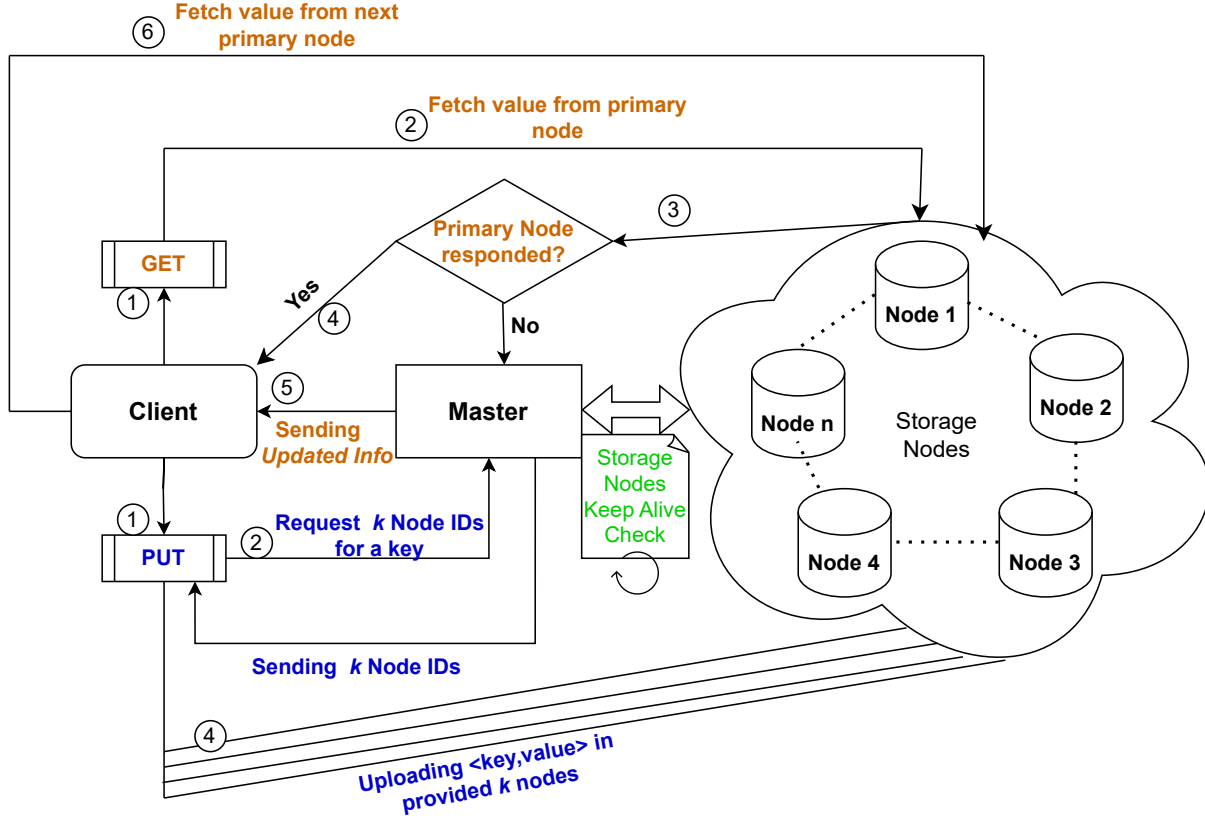


Figure 1: End-to-End workflow of GET and PUT requests from the Client

1. **Init():** The client process initially makes an RPC call to the manager, assuming it knows the manager's IP address. Consequently, the manager sends the necessary metadata information regarding the cluster, i.e., Number of Storage Nodes, Replication factor, etc., to the client as an RPC response.
2. **PutRequest(key, Value):** The client makes an RPC call via *Provide_K_Nodes(key)*: to the manager to obtain control information related to the appropriate k (replicated factor) storage nodes. Upon receiving the requested information from the manager, the client proceeds to independently establish connections with the corresponding storage

nodes and upload the $\langle \text{key}, \text{value} \rangle$ pair to each of storage node. The client uploads the value simultaneously to storage nodes with the help of multi-threading. The client interacts with the manager solely to obtain control information and subsequently interacts directly with the storage nodes for the data path without involving the manager again.

3. **GetRequest(key):** Using our deterministic hash algorithm, the client calculates the primary node for the given key locally and attempts to retrieve its corresponding value directly from it via RPC call. If the value is successfully retrieved, it exists. However, if the connection fails, indicating that the intended primary node has failed, the client contacts the manager via *ProvideNextPrimaryID(key)* and requests updated information on the next primary node. The client then retrieves the value for the requested key from the new primary node and returns it to the application.

3 Design Principles

3.1 Data Partitioning:

Hashing Algorithm(key): We design a deterministic and simple hashing algorithm. Given a key value, we calculate the sum of ASCII values of the characters present in the provided key and finally do $\text{Sum} \% \text{Number of Storage Nodes}$. It returns a unique value between $0 - \text{Number of Storage Nodes} - 1$. We consider the returned index to be the primary replica for handling the given key. Given a random set of keys, we believe this hashing would help distribute the data traffic nearly equally on all the storage nodes. Since both the client and the manager use this algorithm to find the primary node id, we decided to make it deterministic.

3.2 Data Replication:

Based on the primary node index returned by the hashing algorithm, we replicate the data on the k successive storage nodes, assuming the nodes are virtually aligned circularly. To ensure that the data is replicated even in the event of intermediate node failures, the manager process combines the information from the *KeepAliveCheck()* API and starts looking for K storage nodes starting from the primary node index when serving the *Provide_K_Nodes(key)* API. The constraint that there could be at most $k - 1$ failures in the system guarantees that the data will be stored in at least one storage node using this approach. In case of the primary replica failure, the operations workflow can be learned with the description provided earlier in Sec. 2.3 and Fig. 1.

3.3 Data Consistency:

For the PUT request, once the client learns the storage node IDs via the *Provide_K_Nodes(key)* API, it uploads the data synchronously and waits until the storage nodes have confirmed that they have applied the requested write. Consequently, the client does not proceed until it receives write acknowledgments from all the concerned storage nodes. We believe that the advantages of synchronous writing are that the client is guaranteed to have an up-to-date version of the data while performing the GET(key) request since the data is consistent across all the storage nodes.

Overall, Synchronous writing ensures that the client receives an acknowledgment from all the storage nodes, which implies that the data has been successfully written to all the nodes. Therefore, the client is guaranteed to have an up-to-date version of the data. Additionally, the synchronous nature of writing ensures that the data is consistent across all the storage nodes. This is because, during the write operation, all the concerned storage nodes apply the write operation in the same order, thereby ensuring that the data is consistent across all the nodes.

4 Performance Testing

4.1 Throughput

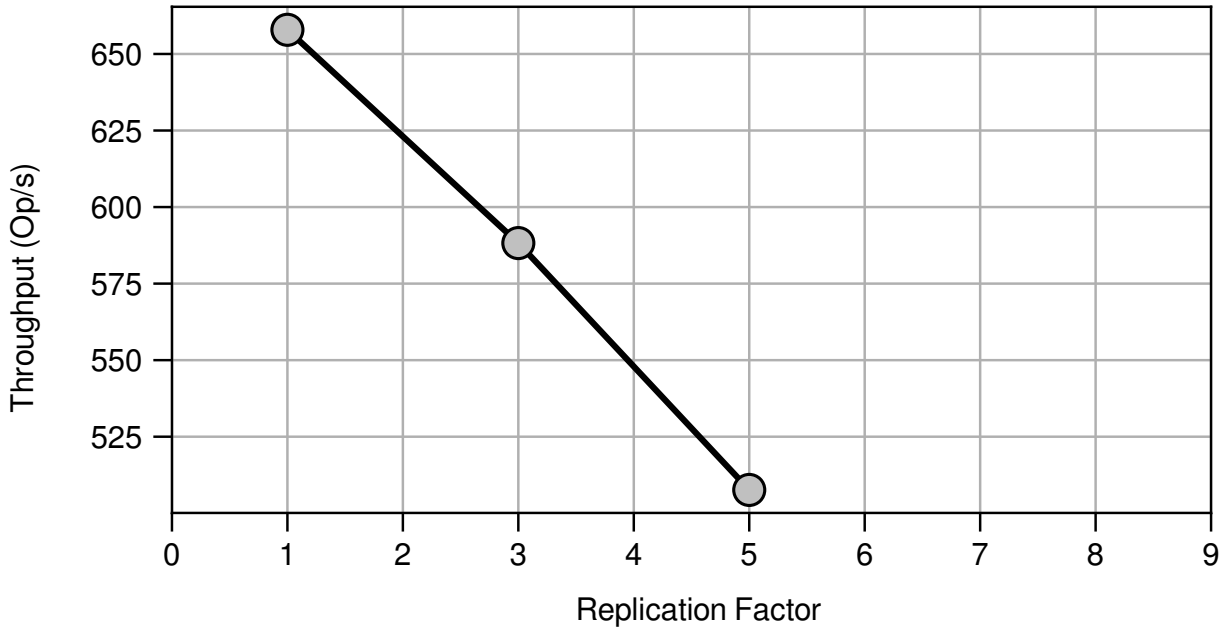


Figure 2: GTStore performance with variation in the number of replica nodes

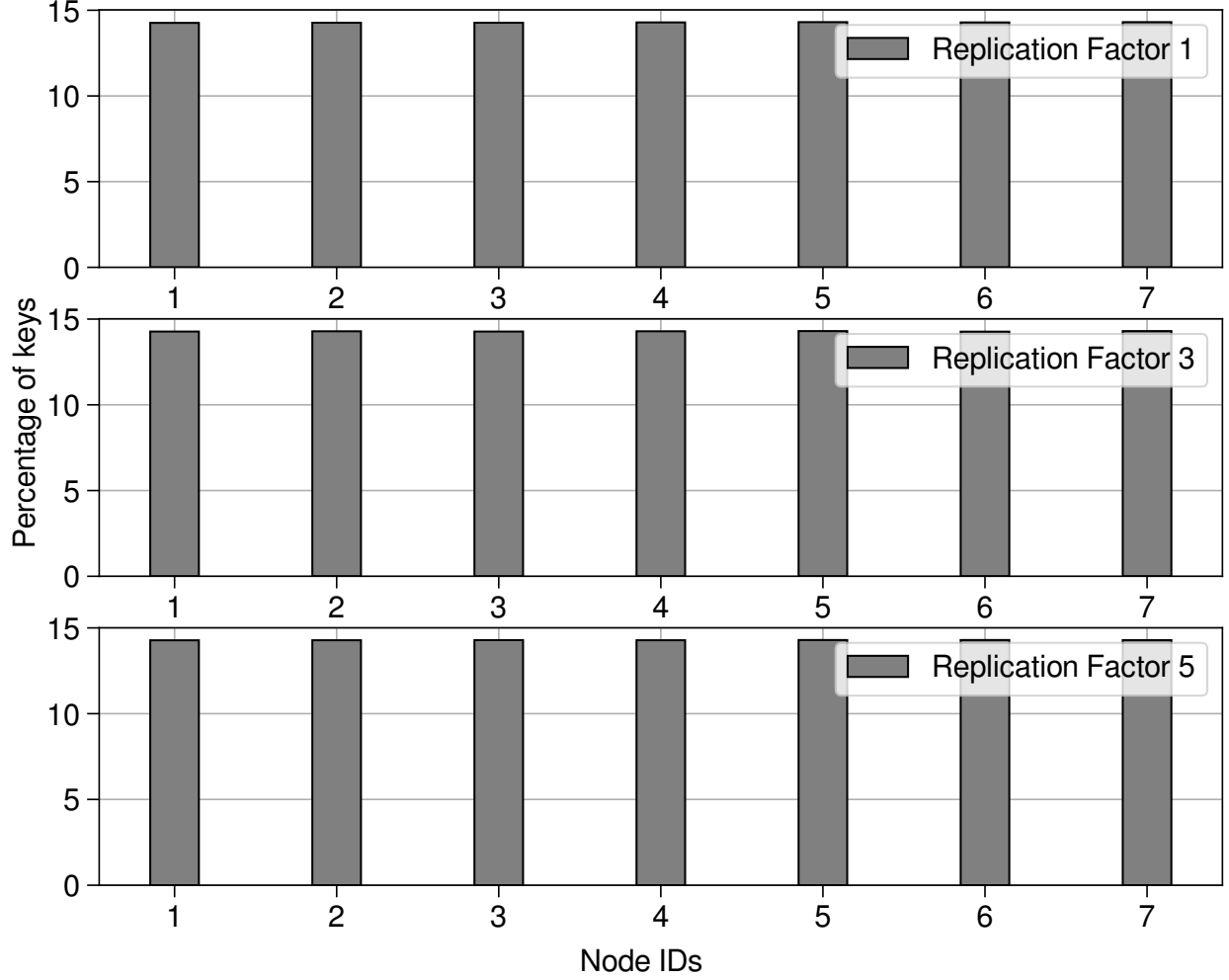


Figure 3: Key distribution in GTStore with respect to replication factor

To measure the throughput of our designed GTStore system, a total of 200,000 operations were performed on 7 nodes, with 1, 3, and 5 as the replication factor. These operations included a combination of 50% GET and 50% PUT operations. The performance of the system is illustrated in Figure 2. Despite an increase in the number of replica nodes, we did not observe a significant decrease in the throughput of the operations due to the efficiency of our design when serving GET/PUT requests. For PUT requests, the client receives control information regarding the replica nodes and then simultaneously uploads data using multi-threading to all nodes. For GET requests, the client node can compute the primary replica locally and connect directly to it to retrieve the data, eliminating the need for any communication with the manager. We believe that the optimizations behind both operations have made our system efficient and scalable.

4.2 Key(s) distribution

During performance testing of the GTStore system, we also analyzed key distribution over the 7 nodes while using 1, 3, and 5 replicas. To ensure robust testing, we selected keys using the random function *rand()* provided by the C library. Figure 3 shows that in all three cases, the keys were distributed almost equally across all seven nodes. We believe that our designed hashing algorithm for data partitioning is effective and guarantees efficient storage space utilization across replica nodes.

5 Pros and Cons of the design (Design Tradeoffs)

5.1 Pros

- The lightning-fast response property is favored in our designed GTStore system. Whenever the client has a GET request, it can locally compute the responsible primary replica in our design. Later, it directly connects to the primary replica and fetches the corresponding value if it exists, without any interaction with the manager node. It results in saving the cost of an RPC call. The manager service only provides control information initially and is never involved in actual data fetch.
- Upon receiving control information from the manager for a PUT request, the client uploads data to the replica node in parallel using multi-threading, resulting in less overall time compared to a sequential connection with the replica nodes.
- By combining synchronous data consistency and the constraint of having at most $k - 1$ replica nodes, our designed GTStore guarantees that at least one storage node for a given key will always be found, and the client will be able to retrieve the value, making it fault-tolerant.

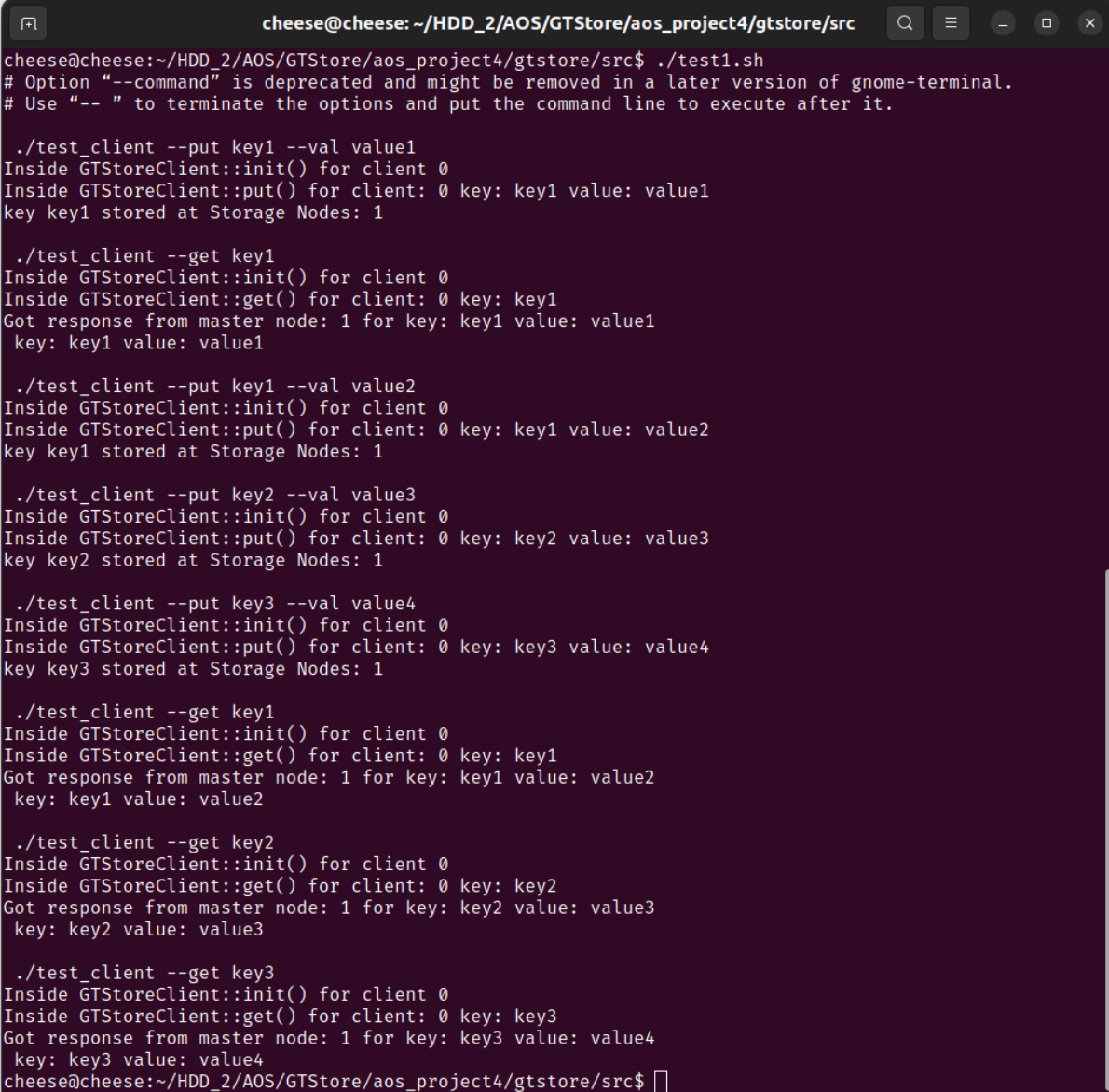
5.2 Cons

- Our design won't work consistently in the case when multiple clients interact with the GTStore service simultaneously.
- Addition of a new storage node to the system won't be easy as we follow a static data partitioning scheme dependent on the initial storage nodes count.
- Append operation can corrupt the data in case of a node failure since we are always replicating a key-value pair to k consecutive nodes starting from the primary replica node; hence any intermediate failure will result in data loss.

6 Tests Logging

6.1 Test 1: Basic Single Server GET/PUT

Client:

A terminal window with a dark purple background and white text. The window title is "cheese@cheese: ~/HDD_2/AOS/GTStore/aos_project4/gtstore/src". The terminal shows the execution of a script named test1.sh. The script performs several operations: it puts key1 with value1, gets key1 (returns value1), puts key1 with value2, puts key2 with value3, puts key3 with value4, gets key1 (returns value2), gets key2 (returns value3), and gets key3 (returns value4). Each operation is logged with its internal state and the response from the master node.

```
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test1.sh
# Option "--command" is deprecated and might be removed in a later version of gnome-terminal.
# Use "-- " to terminate the options and put the command line to execute after it.

./test_client --put key1 --val value1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value1
key key1 stored at Storage Nodes: 1

./test_client --get key1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key1
Got response from master node: 1 for key: key1 value: value1
key: key1 value: value1

./test_client --put key1 --val value2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value2
key key1 stored at Storage Nodes: 1

./test_client --put key2 --val value3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key2 value: value3
key key2 stored at Storage Nodes: 1

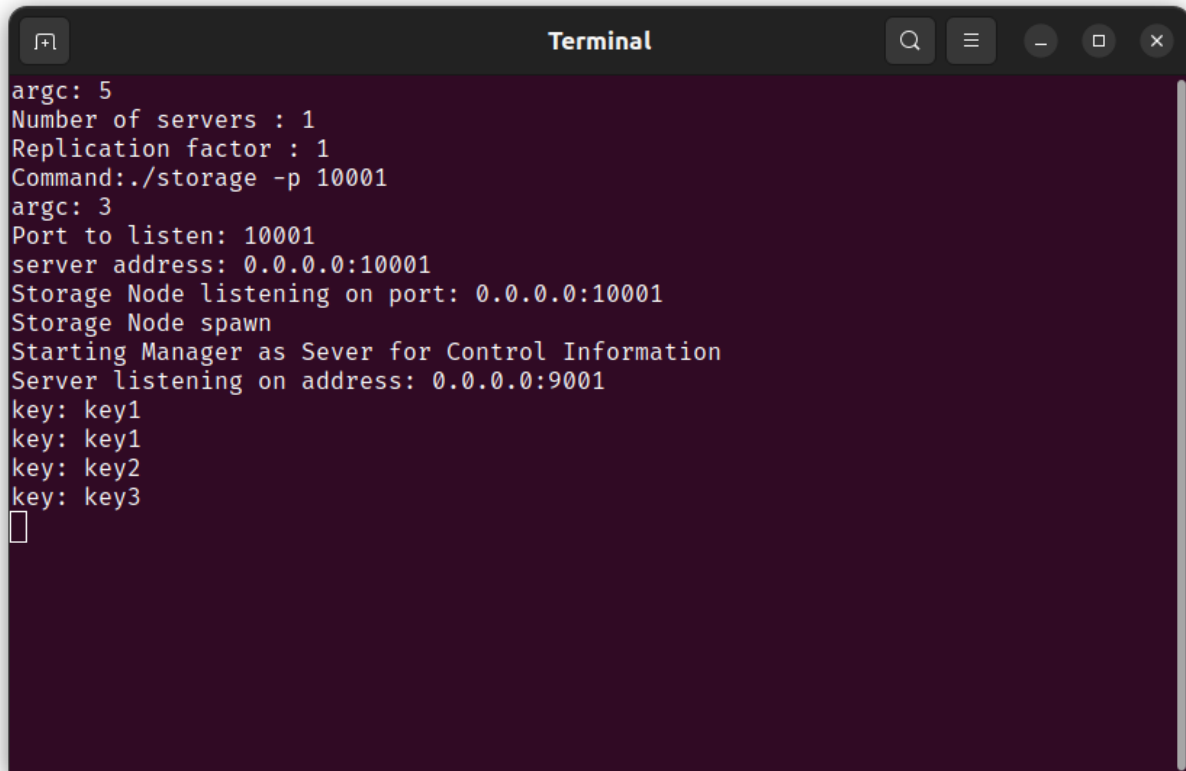
./test_client --put key3 --val value4
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key3 value: value4
key key3 stored at Storage Nodes: 1

./test_client --get key1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key1
Got response from master node: 1 for key: key1 value: value2
key: key1 value: value2

./test_client --get key2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key2
Got response from master node: 1 for key: key2 value: value3
key: key2 value: value3

./test_client --get key3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key3
Got response from master node: 1 for key: key3 value: value4
key: key3 value: value4
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$
```

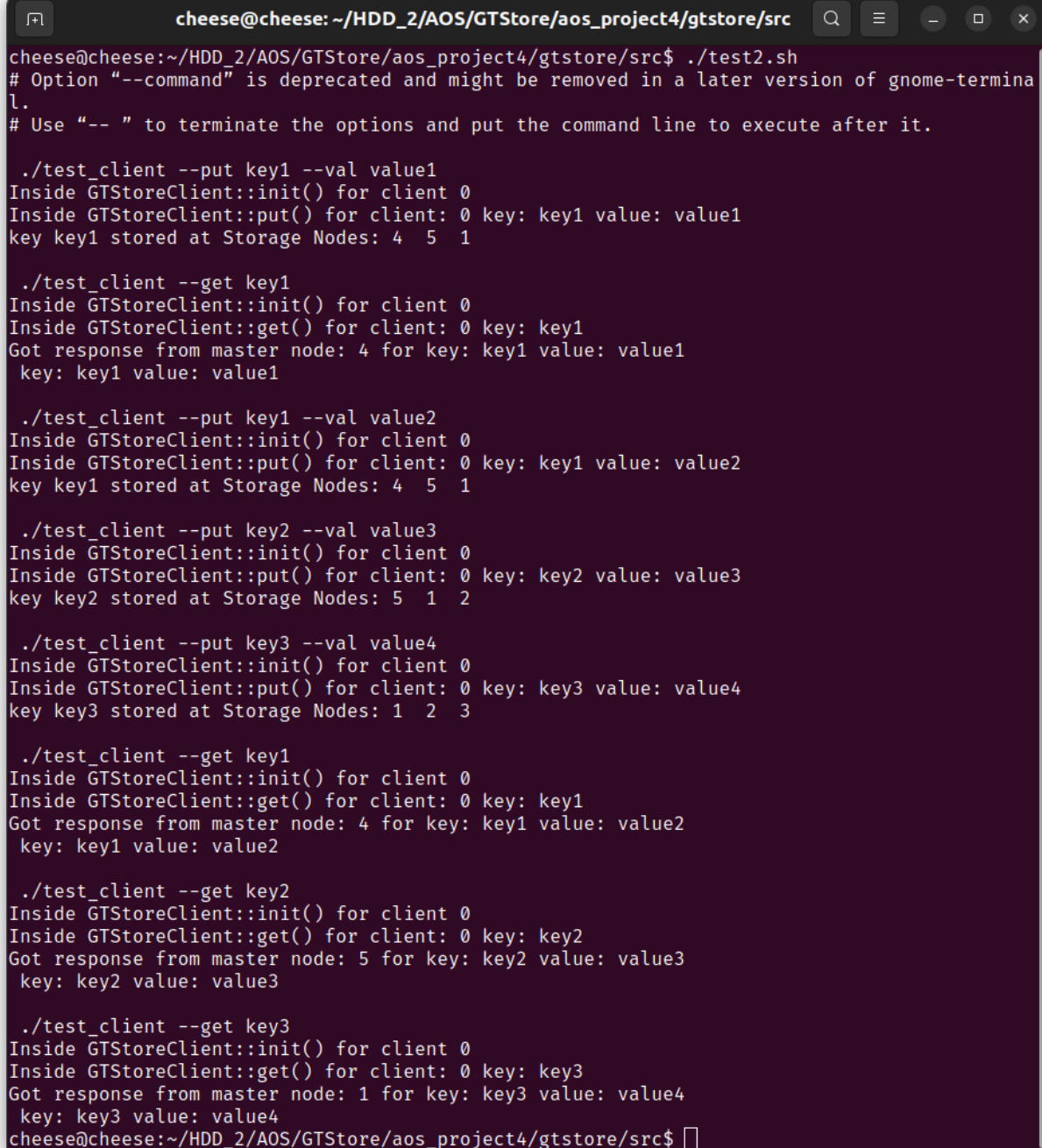

Manager:

A terminal window titled "Terminal" with a dark background and light text. The window has standard macOS window controls (red, yellow, green buttons) in the top-left corner and search, menu, and window management icons in the top-right corner. The terminal output shows the execution of a command with various arguments and the resulting output from a program. The output includes information about the number of servers, replication factor, command used, port to listen, server address, storage node listening port, storage node spawn, starting manager as server for control information, server listening address, and a list of keys.

```
argc: 5
Number of servers : 1
Replication factor : 1
Command:./storage -p 10001
argc: 3
Port to listen: 10001
server address: 0.0.0.0:10001
Storage Node listening on port: 0.0.0.0:10001
Storage Node spawn
Starting Manager as Sever for Control Information
Server listening on address: 0.0.0.0:9001
key: key1
key: key1
key: key2
key: key3
█
```

6.2 Test 2: Basic Multi-Server GET/PUT

Client:

A terminal window titled 'cheese@cheese: ~/HDD_2/AOS/GTStore/aos_project4/gtstore/src' showing the execution of a test script. The script performs several PUT and GET operations on a distributed storage system. The output shows the internal state of the client and the storage nodes for each operation.

```
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test2.sh
# Option "--command" is deprecated and might be removed in a later version of gnome-terminal.
# Use "-- " to terminate the options and put the command line to execute after it.

./test_client --put key1 --val value1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value1
key key1 stored at Storage Nodes: 4 5 1

./test_client --get key1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key1
Got response from master node: 4 for key: key1 value: value1
key: key1 value: value1

./test_client --put key1 --val value2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value2
key key1 stored at Storage Nodes: 4 5 1

./test_client --put key2 --val value3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key2 value: value3
key key2 stored at Storage Nodes: 5 1 2

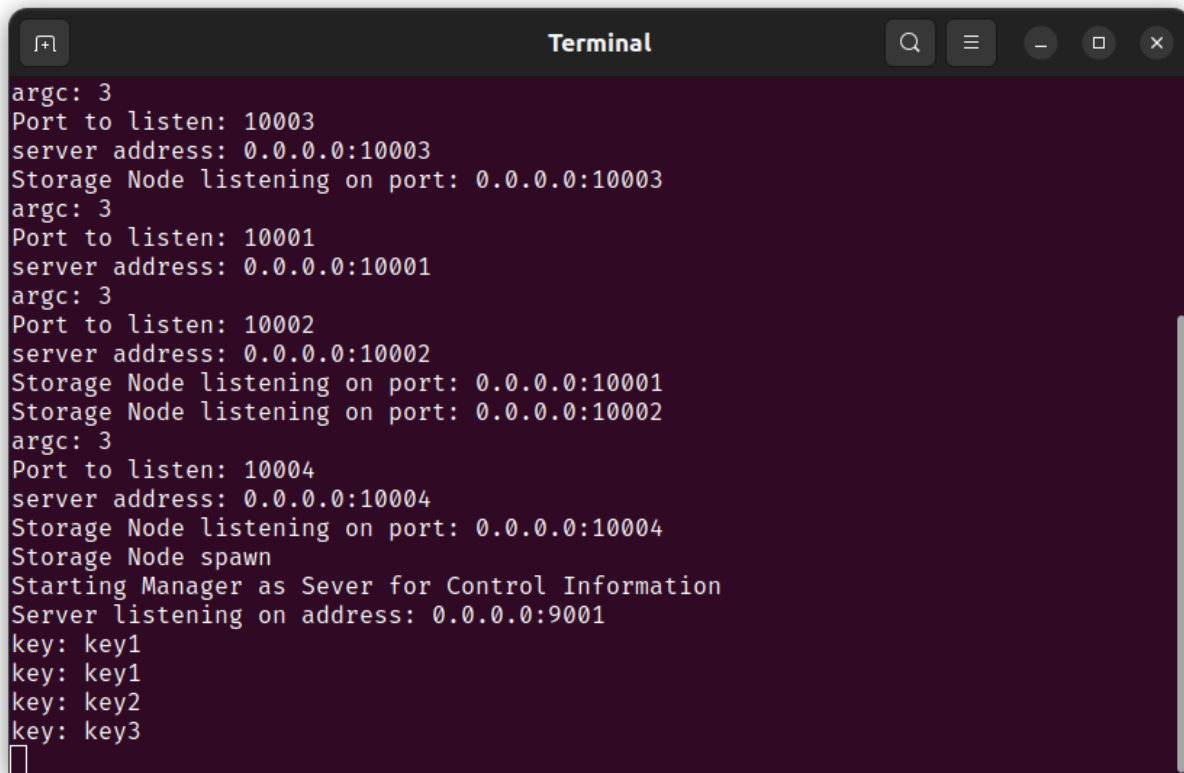
./test_client --put key3 --val value4
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key3 value: value4
key key3 stored at Storage Nodes: 1 2 3

./test_client --get key1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key1
Got response from master node: 4 for key: key1 value: value2
key: key1 value: value2

./test_client --get key2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key2
Got response from master node: 5 for key: key2 value: value3
key: key2 value: value3

./test_client --get key3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key3
Got response from master node: 1 for key: key3 value: value4
key: key3 value: value4
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$
```

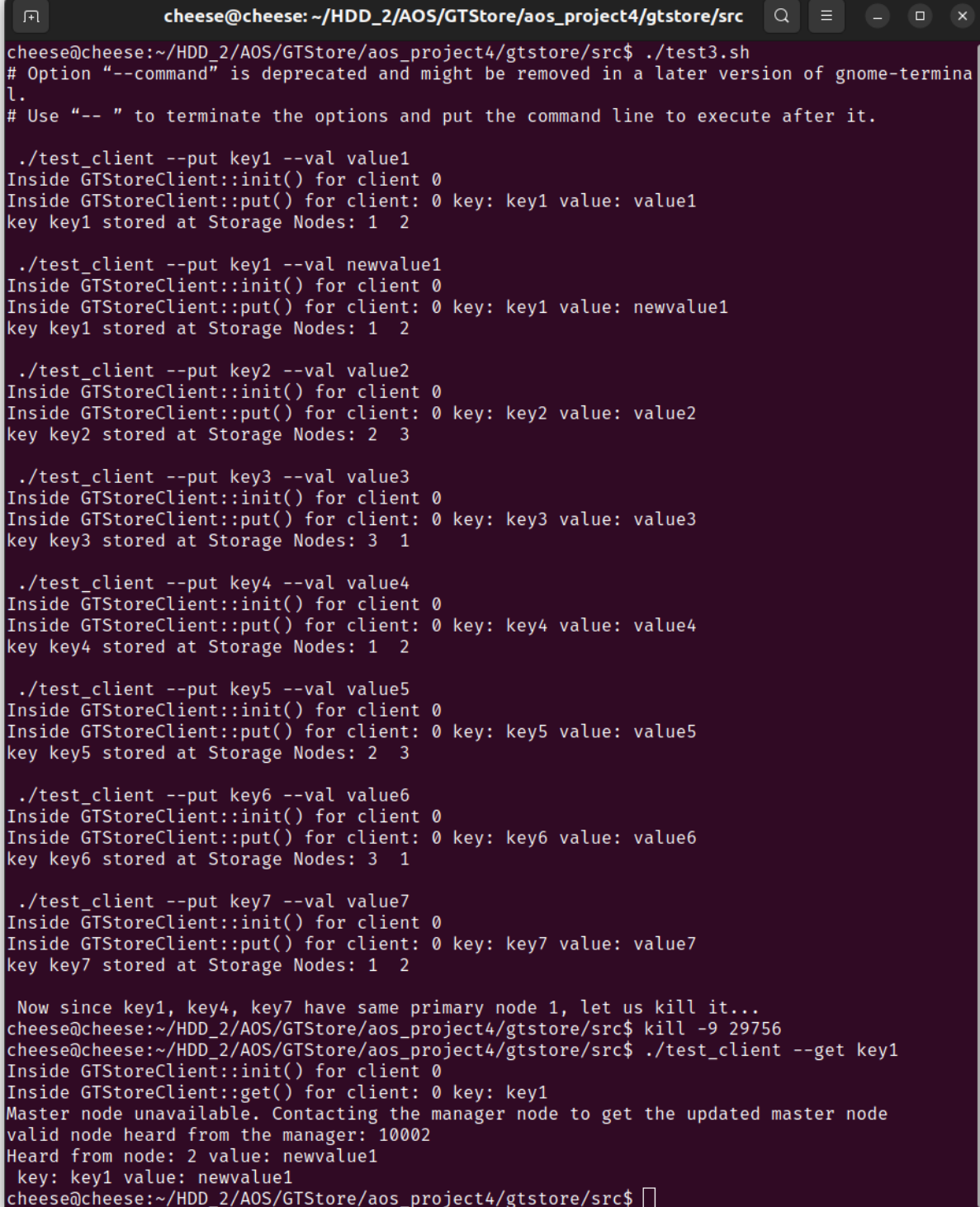
Manager:

A terminal window titled "Terminal" with a dark background and light text. It shows the output of a program. The output consists of several lines of text, including arguments, port numbers, server addresses, and status messages. The terminal has standard window controls (minimize, maximize, close) and a search icon in the title bar. A vertical scrollbar is visible on the right side of the terminal content area.

```
argc: 3
Port to listen: 10003
server address: 0.0.0.0:10003
Storage Node listening on port: 0.0.0.0:10003
argc: 3
Port to listen: 10001
server address: 0.0.0.0:10001
argc: 3
Port to listen: 10002
server address: 0.0.0.0:10002
Storage Node listening on port: 0.0.0.0:10001
Storage Node listening on port: 0.0.0.0:10002
argc: 3
Port to listen: 10004
server address: 0.0.0.0:10004
Storage Node listening on port: 0.0.0.0:10004
Storage Node spawn
Starting Manager as Sever for Control Information
Server listening on address: 0.0.0.0:9001
key: key1
key: key1
key: key2
key: key3
█
```

6.3 Test 3: Availability through a single node failure

Client:



```
cheese@cheese: ~/HDD_2/AOS/GTStore/aos_project4/gtstore/src
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test3.sh
# Option "--command" is deprecated and might be removed in a later version of gnome-terminal.
# Use "-- " to terminate the options and put the command line to execute after it.

./test_client --put key1 --val value1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value1
key key1 stored at Storage Nodes: 1 2

./test_client --put key1 --val newvalue1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: newvalue1
key key1 stored at Storage Nodes: 1 2

./test_client --put key2 --val value2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key2 value: value2
key key2 stored at Storage Nodes: 2 3

./test_client --put key3 --val value3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key3 value: value3
key key3 stored at Storage Nodes: 3 1

./test_client --put key4 --val value4
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key4 value: value4
key key4 stored at Storage Nodes: 1 2

./test_client --put key5 --val value5
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key5 value: value5
key key5 stored at Storage Nodes: 2 3

./test_client --put key6 --val value6
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key6 value: value6
key key6 stored at Storage Nodes: 3 1

./test_client --put key7 --val value7
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key7 value: value7
key key7 stored at Storage Nodes: 1 2

Now since key1, key4, key7 have same primary node 1, let us kill it...
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ kill -9 29756
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test_client --get key1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key1
Master node unavailable. Contacting the manager node to get the updated master node
valid node heard from the manager: 10002
Heard from node: 2 value: newvalue1
key: key1 value: newvalue1
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$
```

[illegible]

6.4 Test 4: Availability through multiple node failures

Client:

```
cheese@cheese: ~/HDD_2/AOS/GTStore/aos_project4/gtstore/src
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test4.sh
# Option "--command" is deprecated and might be removed in a later version of gnome-terminal.
# Use "-- " to terminate the options and put the command line to execute after it.

./test_client --put key1 --val value1
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key1 value: value1
key key1 stored at Storage Nodes: 1 2 3

./test_client --put key2 --val value2
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key2 value: value2
key key2 stored at Storage Nodes: 2 3 4

./test_client --put key3 --val value3
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key3 value: value3
key key3 stored at Storage Nodes: 3 4 5

./test_client --put key4 --val value4
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key4 value: value4
key key4 stored at Storage Nodes: 4 5 6

./test_client --put key5 --val value5
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key5 value: value5
key key5 stored at Storage Nodes: 5 6 7

./test_client --put key6 --val value6
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key6 value: value6
key key6 stored at Storage Nodes: 6 7 1

./test_client --put key7 --val value7
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key7 value: value7
key key7 stored at Storage Nodes: 7 1 2

./test_client --put key8 --val value8
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key8 value: value8
key key8 stored at Storage Nodes: 1 2 3

./test_client --put key9 --val value9
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key9 value: value9
key key9 stored at Storage Nodes: 2 3 4

./test_client --put key10 --val value10
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key10 value: value10
key key10 stored at Storage Nodes: 7 1 2

./test_client --put key11 --val value11
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key11 value: value11
key key11 stored at Storage Nodes: 1 2 3

./test_client --put key12 --val value12
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key12 value: value12
key key12 stored at Storage Nodes: 1 2 3
```

```

./test_client --put key13 --val value13
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key13 value: value13
key key13 stored at Storage Nodes: 3 4 5

./test_client --put key14 --val value14
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key14 value: value14
key key14 stored at Storage Nodes: 4 5 6

./test_client --put key15 --val value15
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key15 value: value15
key key15 stored at Storage Nodes: 5 6 7

./test_client --put key16 --val value16
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key16 value: value16
key key16 stored at Storage Nodes: 6 7 1

./test_client --put key17 --val value17
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key17 value: value17
key key17 stored at Storage Nodes: 7 1 2

./test_client --put key18 --val value18
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key18 value: value18
key key18 stored at Storage Nodes: 1 2 3

./test_client --put key19 --val value19
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key19 value: value19
key key19 stored at Storage Nodes: 2 3 4

./test_client --put key20 --val value20
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key20 value: value20
key key20 stored at Storage Nodes: 1 2 3

Overwriting key5, key10, key15

./test_client --put key5 --val newvalue5
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key5 value: newvalue5
key key5 stored at Storage Nodes: 5 6 7

./test_client --put key10 --val newvalue10
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key10 value: newvalue10
key key10 stored at Storage Nodes: 7 1 2

./test_client --put key15 --val newvalue15
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::put() for client: 0 key: key15 value: newvalue15
key key15 stored at Storage Nodes: 5 6 7

let us kill it two nodes
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ kill -9 31283
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ kill -9 31286
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ ./test_client --get key5
Inside GTStoreClient::init() for client 0
Inside GTStoreClient::get() for client: 0 key: key5
Master node unavailable. Contacting the manager node to get the updated master node
valid node heard from the manager: 10006
Heard from node: 6 value: newvalue5
key: key5 value: newvalue5
cheese@cheese:~/HDD_2/AOS/GTStore/aos_project4/gtstore/src$ 

```


Manager:

```
Terminal
Command: ./storage -p 10003
Command: ./storage -p 10004
argc: 3
Port to listen: 10005
server address: 0.0.0.0:10005
argc: 3
Port to listen: 10002
server address: 0.0.0.0:10002
Storage Node listening on port: 0.0.0.0:10002
Storage Node listening on port: 0.0.0.0:10005
argc: 3
Port to listen: 10006
server address: 0.0.0.0:10006
Storage Node listening on port: 0.0.0.0:10006
argc: 3
Port to listen: 10007
server address: 0.0.0.0:10007
Storage Node listening on port: 0.0.0.0:10007
argc: 3
Port to listen: 10001
server address: 0.0.0.0:10001
argc: 3
Port to listen: 10003
server address: 0.0.0.0:10003
Storage Node listening on port: 0.0.0.0:10001
Storage Node listening on port: 0.0.0.0:10003
argc: 3
Port to listen: 10004
server address: 0.0.0.0:10004
Storage Node listening on port: 0.0.0.0:10004
Storage Node spawn
Starting Manager as Sever for Control Information
Server listening on address: 0.0.0.0:9001
key: key1
key: key2
key: key3
key: key4
key: key5
key: key6
key: key7
key: key8
key: key9
key: key10
key: key11
key: key12
key: key13
key: key14
key: key15
key: key16
key: key17
key: key18
key: key19
key: key20
key: key5
key: key10
key: key15
Killed
14: failed to connect to all addresses: last error: UNKNOWN: ip: 0.0.0.0:10003: Failed to c
```



```
key: key5
key: key10
key: key15
Killed
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
Killed
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10005: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10005: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10005: Failed to c
onnect to remote host: Connection refused
RPC FAILED
inside get_next_primary_node
key: key5
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10002: Failed to c
onnect to remote host: Connection refused
RPC FAILED
14: failed to connect to all addresses; last error: UNKNOWN: ipv4:0.0.0.0:10005: Failed to c
```