Project 4

I. System Components

GT Store

i. Centralized manager.

The centralized manager is responsible for adding, removing, and managing nodes and distributing client requests across them. XML-RPC was used to communicate with clients/servers in this project.

A centralized manager can handle the following requests:

```
virtual uint32_t JoinNode(const string& ip, const string& port) = 0;
virtual void LeaveNode(uint32_t token) = 0;
virtual bool Put(const string& key, const vector<string>& values) = 0;
virtual vector<string> Get(const string& key) = 0;
```

Centralized manager stores the following data:

```
vector <pair <uint32_t, string >> _real_node_index; // List of all real nodes.
vector <pair <uint32_t, string >> _virtual_node_index; // List of all virtual nodes.
map <uint32_t, string> _token_to_id_index; // Map to preserve the correspondence between the server token and its id, which is "ip: port".
```

Also, to check the health of the nodes, the manager maintains the thread _nodeCheckingThread and stores the _isNodeCheckingStop flag.

A consistent hashing method was chosen for data distribution and storage. So nodes are also managed by principles of consistent hashing. Upon receiving a request to add a node (METHOD_JOIN), the manager creates node_id = ip + ":" + port, and generates the URL of the node (url = string ("http: //") + node_id + "/ RPC2").Then the manager will calculate the token (uint32_t getToken()) from the hash value of "node_id" and store the correspondence between "node_id" and the token in _token_to_id_index. At the time of adding, the nodes are sorted into a hash ring by the token's value (a list _real_node_index is used). Finally, the manager notifies the neighboring nodes on the right and left about this new node (METHOD_CHANGE_LEFT / RIGHT_NODE). After this notification, all three nodes (added one and its neighbors) change the values of _leftNodeUrl and _rightNodeUrl, which are pointers. The next step is to create virtual nodes for each real node (VIRTUAL_NODE_COUNT 10), which are necessary for a more even distribution of requests from clients. The AddVirtualNode() function is used to generate virtual nodes. For virtual nodes, we hash the value of "node_id + "#" + to_string(r)" to get the token, it turns out something like this: 127.0.0.1:4564#1, 127.0.0.1:4564#2, 127.0.0.1:4564#3, etc. Further, using the token's value, each virtual node is distributed in a hash ring (for this, we go through the list _virtual_node_index). You can see this process more clearly in Figures 1 and 2.

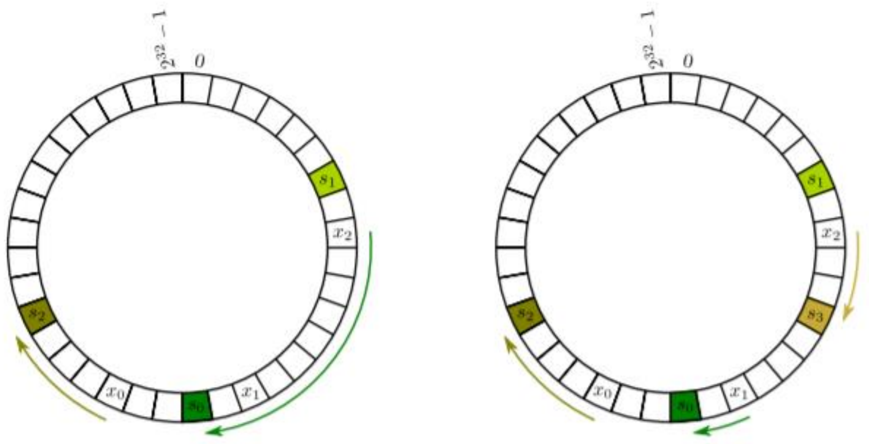


Figure 1. Objects are assigned to the server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last server. (Right) Adding a new server s3. Object x2 moves from s0 to s3.

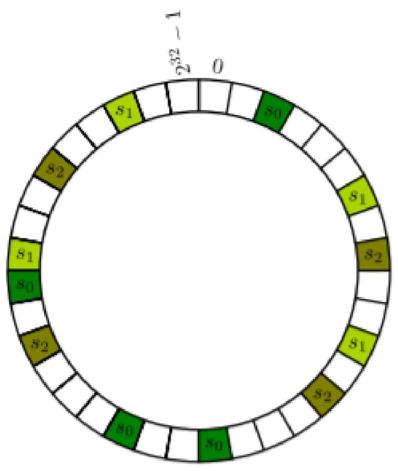


Figure 2. Decreasing the variance by assigning each server multiple hash values.

In deleting a node, the manager uses _token_to_id_index to find the node_id by token's value and then removes the real node and all virtual nodes from the ring. Simultaneously, the manager notifies the neighboring real nodes to change their pointers on the left/right neighbors (METHOD_CHANGE_LEFT / RIGHT_NODE).

The token is generated according to the following algorithm:

1) Get the hash of the "node_id" value. The program uses sha1_hash, not the best option from the security side, but since the project is educational, it doesn't hurt the system a lot;

2) Determine the maximum possible value for token, mod = pow (2.0, 32) - 1;

3) Calculate the token = hash % mod.

The description of the distribution of client requests will be described in the Data Partitioning part.

ii. **Storage/Data nodes**.

Storage nodes represent a key-value store. Each node can process the following requests:

```
virtual bool PutReplica(const DataItem& values, int replicaCounter) = 0;
virtual DataItem GetReplica(const string& key, int replicaCounter) = 0;
virtual void ChangeLeftNode(const string& url) = 0;
virtual void ChangeRightNode(const string& url) = 0;
virtual void Release() = 0;
virtual bool Put(const string& key, const vector<string>& values) = 0;
virtual vector<string> Get(const string& key) = 0;
```

Each node stores the following data:

```
map <string, DataItem> _storage; // Map to preserve the correspondence between the key and the DataItem for this node
map <string, DataItem> _storage_replica; // Map to preserve the correspondence between key and DataItem for neighboring nodes
string _token;
string _ip;
uint32_t _servicePort; // port on which the node is running
string _leftNodeUrl; // pointer to the left neighbor node
string _rightNodeUrl; // pointer to the right neighbor node
thread _syncThread; // the thread doing the synchronization
mutex _syncLock;
list <pair <DataItem, int >> _replicaQueue; // replication queue
bool _syncFlag; // flag for data synchronization
string _managerUrl;
thread _asyncCall; // thread connecting to manager
```

The DataItem structure includes the key value, the data itself, and a timestamp.

The node is started using the Run() function. The node sends a METHOD_JOIN request to connect to the manager using the ConnectToManager() function. The connection is made via a free port, defined in the range from 1024 to 65535 using the GetFreePort() function. A separate thread carries out each connection to the server.

In order to save client data, the node stores the data of the DataItem with a timestamp in _storage[key] using the Put() function. Then it adds this data to the replication queue (_replicaQueue.push_back (pair <DataItem, int> (_ storage[key], static_cast <int64_t> (REPLICA_COUNT)))) and starts synchronization with other nodes(_syncLock.unlock ()). The number of replications depends on REPLICA_COUNT. The replication process is described in more detail in Data replication.

The Get() function is used to retrieve data from the nodes. To find the DataItem value by key, we go through the _storage array. If not found in the local storage, we search on neighboring nodes using the GetReplica() function (_storage_replica).

Client API calls (one-by-one).

The client can send the following requests:

```
virtual bool Put (const string & url, const string & key, const vector <string> & values) = 0;
```

virtual vector <string> Get (const string & url, const string & key) = 0;

i. init (): API was modified slightly by adding ip and port values, which used to form a link to connect to the manager.
ii. put (key, value): to send data to the server, we first check that the key and data values are specified, then we check that the size of the key and data does not exceed the defined ones. Then, using the Put() function, we send the data to the manager by sending a METHOD_PUT request.
iii. get (key): to get data from the server, we check the key value, it must not be empty and exceed the specified value MAX_KEY_BYTE_PER_REQUEST. Then, using the Get() function, we send METHOD_GET to the server.
iv. finalize (): the function does nothing since the client does not store any data in this implementation.

II. Design Principles

1) Data Partitioning

As mentioned earlier, we are using the Consistent hashing method. In order to add data to the server, we use the Put() function. We calculate the token for the key's value and look for the closest clockwise virtual node using the FindNearestNode() function. Since all the nodes are in the hash ring, we can always find such a node. Then the manager saves the data to the real node that owns this virtual node. The process can be seen in Figure 1 and 2.

In order to get data by a key, we use the Get() function. The algorithm is similar to Put(), we also calculate the token and look for the closest matching node. Then we turn to the real node with a request to receive data, and return it to client.

pros

- The distribution of requests does not depend on the number of nodes.
- When inserting or deleting a node, it is unnecessary to rebuild the hashes of all the nodes, but only the neighboring two nodes.

cons

- The time to search for a node is longer than the scheme when the hash is calculated by modulo the number of nodes.
- Compared to the Round-robin load balancing algorithm, there is a likelihood of less uniform participation of nodes in processing requests.

2) Data Replication

When data is added to a node, it is also added to the replication queue, as already discussed. Using the PutReplica() function, we put information about a replica of the client's data to _storage_replica and then queue the data for replication to neighboring K nodes clockwise. Data synchronization occurs using the Syncing() function, which sends a METHOD_PUT_REPLICA request to neighboring nodes to save a new replica. Synchronization happens after each data modification (function Put ()).

To receive data from a node, each node firstly searches for it in the local storage. If the required data is not found, a METHOD_GET_REPLICA request is sent to the neighboring K nodes clockwise (GetReplica ()). We receive responses from all nodes, but return the data that was last modified (if (value_from_next_node.timestamp <= 0 || value_from_next_node.timestamp <= response.timestamp)). Since replicas are propagated in the background, it may happen that the updated data has not yet appeared on all nodes, so the newest replica is transmitted.

When a node fails, the GTStore manager throws out the failed node and sends its neighbors a METHOD_CHANGE_LEFT/RIGHT_NODE request to change pointers on these nodes. Having received this request, the nodes begin to exchange replicas of all their data with their new neighbors, located clockwise, using the ChangeLeftNode( ).

3) Data Consistency

When the put() function is called, each object is assigned a timestamp (in this implementation, it plays the role of a version). These timestamps are propagated during every replication. When there is no way to get data from the primary replica, the data is taken from the replicas of neighboring nodes, and the most recent record by the timestamp is selected from among the all replicas. Replicas are synchronized whenever data changes using the Syncing() function.

4) Temporary Failure Handler

If the _isNodeCheckingStop flag is set to false, the manager polls the nodes for health checks every 5 seconds by sending a METHOD_HEARTBEAT request. If a node does not respond within 5 seconds, it is added to the tokensToLeave list using the NodeCheckingLoop() function. After that, all nodes included in the list will be disconnected from the manager using the LeaveNode() process. Removing a node and retransmitting replicas between new neighbors were described above.

III. Client Driver Application

We have three types of tests:

1) single_set_get (): where the client sends data to the server (client.put (key, value)) and then checks if it was sent correctly (auto result_value = client.get (key));

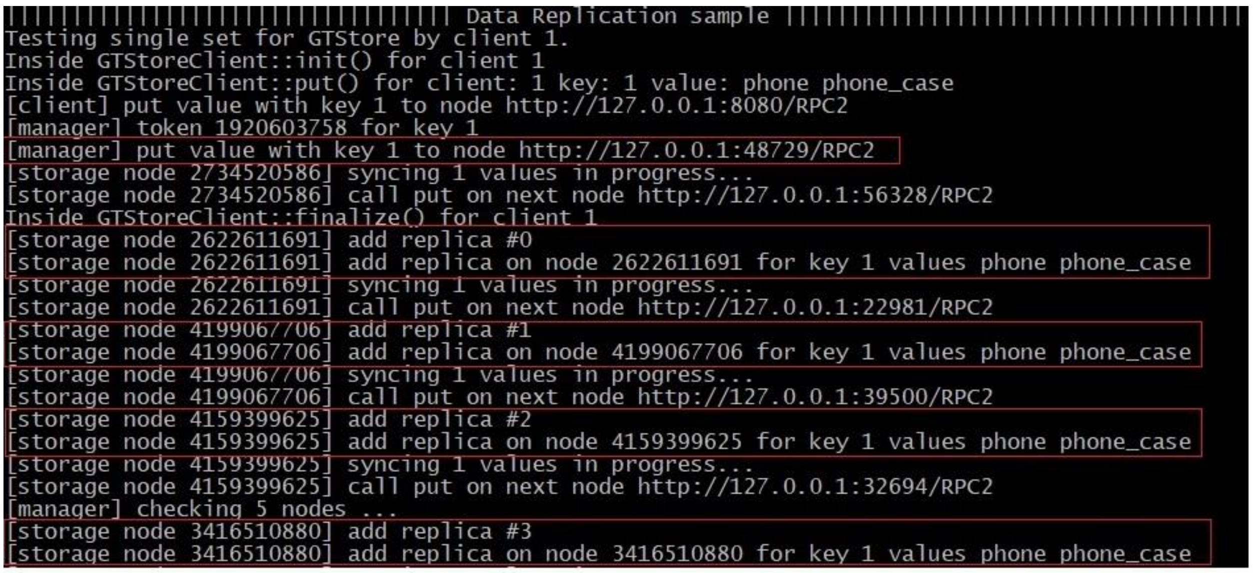
2) single_set (): where the client only sends data to the server (client.put (key, value));

3) single_get (): where the client requests data from the server by key (auto result_value = client.get (key)).

When running the script, we check the design principles for correctness:

1) Data Replication.

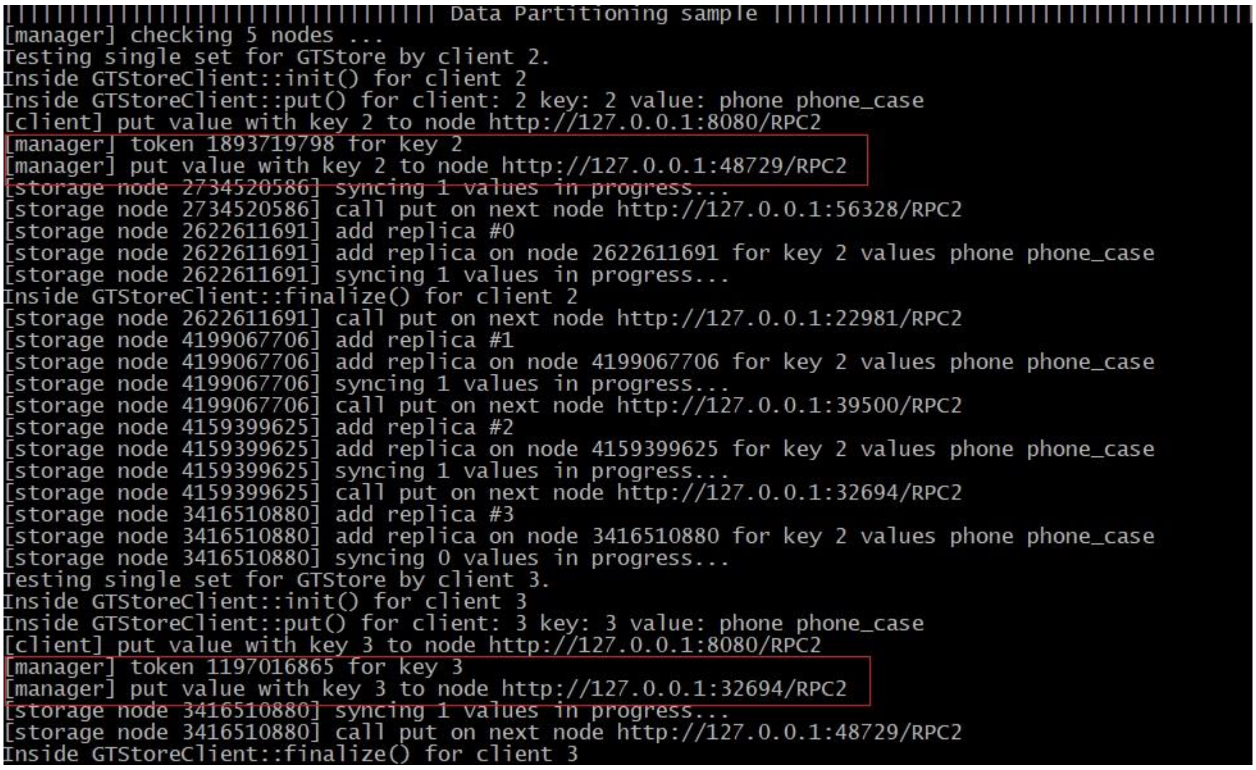
To do this, we send the data of one client (single_set) and check how it will be copied to neighboring nodes.



```
|||||||||||||||||||||||||||||||||| Data Replication sample ||||||||||||||||||||||||||||||||||||
Testing single set for GTStore by client 1.
Inside GTStoreClient::init() for client 1
Inside GTStoreClient::put() for client: 1 key: 1 value: phone phone_case
[client] put value with key 1 to node http://127.0.0.1:8080/RPC2
[manager] token 1920603758 for key 1
[manager] put value with key 1 to node http://127.0.0.1:48729/RPC2
[storage node 2734520586] syncing 1 values in progress...
[storage node 2734520586] call put on next node http://127.0.0.1:56328/RPC2
Inside GTStoreClient::finalize() for client 1
[storage node 2622611691] add replica #0
[storage node 2622611691] add replica on node 2622611691 for key 1 values phone phone_case
[storage node 2622611691] syncing 1 values in progress...
[storage node 2622611691] call put on next node http://127.0.0.1:22981/RPC2
[storage node 4199067706] add replica #1
[storage node 4199067706] add replica on node 4199067706 for key 1 values phone phone_case
[storage node 4199067706] syncing 1 values in progress...
[storage node 4199067706] call put on next node http://127.0.0.1:39500/RPC2
[storage node 4159399625] add replica #2
[storage node 4159399625] add replica on node 4159399625 for key 1 values phone phone_case
[storage node 4159399625] syncing 1 values in progress...
[storage node 4159399625] call put on next node http://127.0.0.1:32694/RPC2
[manager] checking 5 nodes ...
[storage node 3416510880] add replica #3
[storage node 3416510880] add replica on node 3416510880 for key 1 values phone phone_case
```

2) Data Partitioning sample

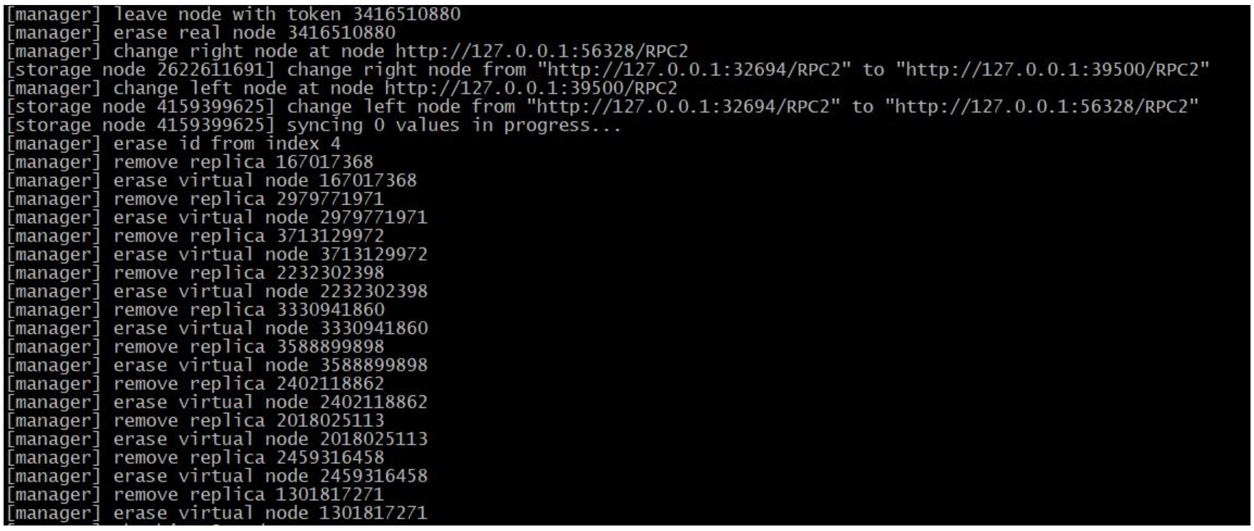
To do this, we add two more clients and track where their original data will be stored.



```
|||||||||||||||||||||||||||||||||| Data Partitioning sample ||||||||||||||||||||||||||||||||||||
[manager] checking 5 nodes ...
Testing single set for GTStore by client 2.
Inside GTStoreClient::init() for client 2
Inside GTStoreClient::put() for client: 2 key: 2 value: phone phone_case
[client] put value with key 2 to node http://127.0.0.1:8080/RPC2
[manager] token 1893719798 for key 2
[manager] put value with key 2 to node http://127.0.0.1:48729/RPC2
[storage node 2734520586] syncing 1 values in progress...
[storage node 2734520586] call put on next node http://127.0.0.1:56328/RPC2
[storage node 2622611691] add replica #0
[storage node 2622611691] add replica on node 2622611691 for key 2 values phone phone_case
[storage node 2622611691] syncing 1 values in progress...
Inside GTStoreClient::finalize() for client 2
[storage node 2622611691] call put on next node http://127.0.0.1:22981/RPC2
[storage node 4199067706] add replica #1
[storage node 4199067706] add replica on node 4199067706 for key 2 values phone phone_case
[storage node 4199067706] syncing 1 values in progress...
[storage node 4199067706] call put on next node http://127.0.0.1:39500/RPC2
[storage node 4159399625] add replica #2
[storage node 4159399625] add replica on node 4159399625 for key 2 values phone phone_case
[storage node 4159399625] syncing 1 values in progress...
[storage node 4159399625] call put on next node http://127.0.0.1:32694/RPC2
[storage node 3416510880] add replica #3
[storage node 3416510880] add replica on node 3416510880 for key 2 values phone phone_case
[storage node 3416510880] syncing 0 values in progress...
Testing single set for GTStore by client 3.
Inside GTStoreClient::init() for client 3
Inside GTStoreClient::put() for client: 3 key: 3 value: phone phone_case
[client] put value with key 3 to node http://127.0.0.1:8080/RPC2
[manager] token 1197016865 for key 3
[manager] put value with key 3 to node http://127.0.0.1:32694/RPC2
[storage node 3416510880] syncing 1 values in progress...
[storage node 3416510880] call put on next node http://127.0.0.1:48729/RPC2
Inside GTStoreClient::finalize() for client 3
```

3) Failure Handler

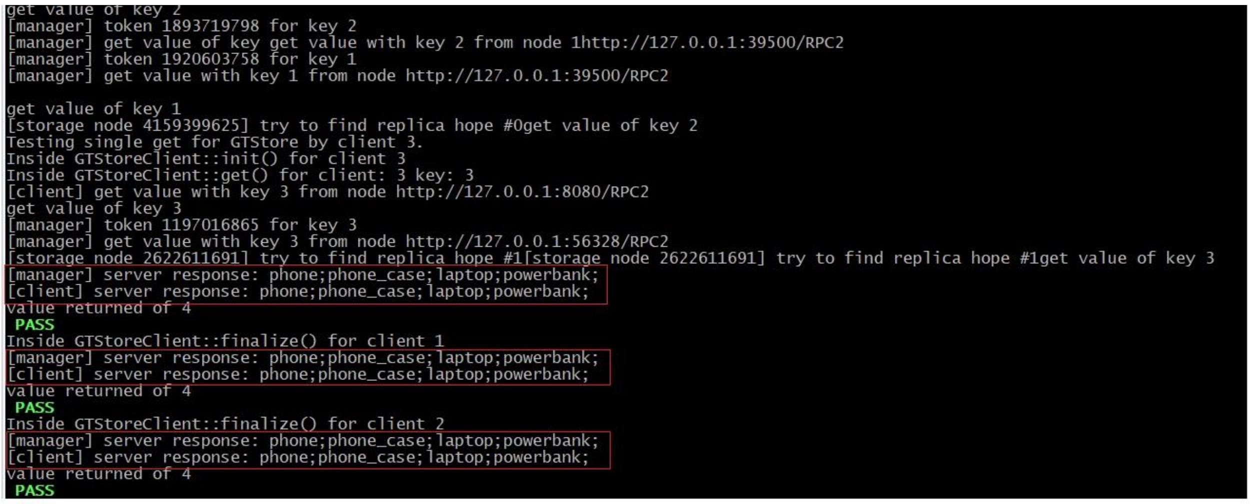
To do this, we disable the node and monitor the manager's behavior. It deletes the real node, virtual nodes related to the real node, and notifies neighboring nodes about the need to change pointers.



```
[manager] leave node with token 3416510880
[manager] erase real node 3416510880
[manager] change right node at node http://127.0.0.1:56328/RPC2
[storage node 2622611691] change right node from "http://127.0.0.1:32694/RPC2" to "http://127.0.0.1:39500/RPC2"
[manager] change left node at node http://127.0.0.1:39500/RPC2
[storage node 4159399625] change left node from "http://127.0.0.1:32694/RPC2" to "http://127.0.0.1:56328/RPC2"
[storage node 4159399625] syncing 0 values in progress...
[manager] erase id from index 4
[manager] remove replica 167017368
[manager] erase virtual node 167017368
[manager] remove replica 2979771971
[manager] erase virtual node 2979771971
[manager] remove replica 3713129972
[manager] erase virtual node 3713129972
[manager] remove replica 2232302398
[manager] erase virtual node 2232302398
[manager] remove replica 3330941860
[manager] erase virtual node 3330941860
[manager] remove replica 3588899898
[manager] erase virtual node 3588899898
[manager] remove replica 2402118862
[manager] erase virtual node 2402118862
[manager] remove replica 2018025113
[manager] erase virtual node 2018025113
[manager] remove replica 2459316458
[manager] erase virtual node 2459316458
[manager] remove replica 1301817271
[manager] erase virtual node 1301817271
```

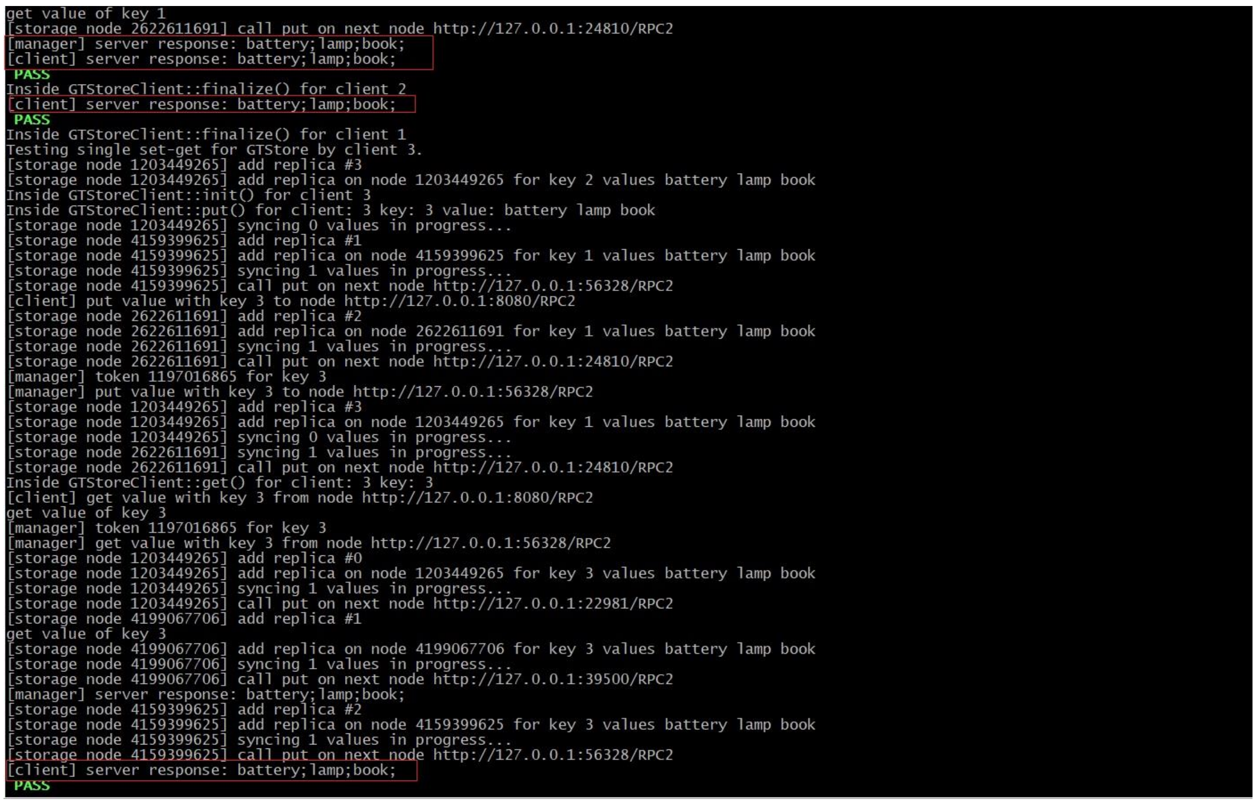
4) Data Consistency

For verification, we request customer data using the single_get function and check that they are valid (phone; phone_case; laptop; powerbank). Then we change the data (single_set_get) and check that the client receives only new data (battery; lamp; book).



```
get value of key 2
[manager] token 1893719798 for key 2
[manager] get value of key get value with key 2 from node 1http://127.0.0.1:39500/RPC2
[manager] token 1920603758 for key 1
[manager] get value with key 1 from node http://127.0.0.1:39500/RPC2

get value of key 1
[storage node 4159399625] try to find replica hope #0get value of key 2
Testing single get for GTStore by client 3.
Inside GTStoreClient::init() for client 3
Inside GTStoreClient::get() for client: 3 key: 3
[client] get value with key 3 from node http://127.0.0.1:8080/RPC2
get value of key 3
[manager] token 1197016865 for key 3
[manager] get value with key 3 from node http://127.0.0.1:56328/RPC2
[storage node 2622611691] try to find replica hope #1[storage node 2622611691] try to find replica hope #1get value of key 3
[manager] server response: phone;phone_case;laptop;powerbank;
[client] server response: phone;phone_case;laptop;powerbank;
value returned of 4
 PASS
Inside GTStoreClient::finalize() for client 1
[manager] server response: phone;phone_case;laptop;powerbank;
[client] server response: phone;phone_case;laptop;powerbank;
value returned of 4
 PASS
Inside GTStoreClient::finalize() for client 2
[manager] server response: phone;phone_case;laptop;powerbank;
[client] server response: phone;phone_case;laptop;powerbank;
value returned of 4
 PASS
```



```
get value of key 1
[storage node 2622611691] call put on next node http://127.0.0.1:24810/RPC2
[manager] server response: battery;lamp;book;
[client] server response: battery;lamp;book;
 PASS
Inside GTStoreClient::finalize() for client 2
[client] server response: battery;lamp;book;
 PASS
Inside GTStoreClient::finalize() for client 1
Testing single set-get for GTStore by client 3.
[storage node 1203449265] add replica #3
[storage node 1203449265] add replica on node 1203449265 for key 2 values battery lamp book
Inside GTStoreClient::init() for client 3
Inside GTStoreClient::put() for client: 3 key: 3 value: battery lamp book
[storage node 1203449265] syncing 0 values in progress...
[storage node 4159399625] add replica #1
[storage node 4159399625] add replica on node 4159399625 for key 1 values battery lamp book
[storage node 4159399625] syncing 1 values in progress...
[storage node 4159399625] call put on next node http://127.0.0.1:56328/RPC2
[client] put value with key 3 to node http://127.0.0.1:8080/RPC2
[storage node 2622611691] add replica #2
[storage node 2622611691] add replica on node 2622611691 for key 1 values battery lamp book
[storage node 2622611691] syncing 1 values in progress...
[storage node 2622611691] call put on next node http://127.0.0.1:24810/RPC2
[manager] token 1197016865 for key 3
[manager] put value with key 3 to node http://127.0.0.1:56328/RPC2
[storage node 1203449265] add replica #3
[storage node 1203449265] add replica on node 1203449265 for key 1 values battery lamp book
[storage node 1203449265] syncing 0 values in progress...
[storage node 2622611691] syncing 1 values in progress...
[storage node 2622611691] call put on next node http://127.0.0.1:24810/RPC2
Inside GTStoreClient::get() for client: 3 key: 3
[client] get value with key 3 from node http://127.0.0.1:8080/RPC2
get value of key 3
[manager] token 1197016865 for key 3
[manager] get value with key 3 from node http://127.0.0.1:56328/RPC2
[storage node 1203449265] add replica #0
[storage node 1203449265] add replica on node 1203449265 for key 3 values battery lamp book
[storage node 1203449265] syncing 1 values in progress...
[storage node 1203449265] call put on next node http://127.0.0.1:22981/RPC2
[storage node 4199067706] add replica #1
get value of key 3
[storage node 4199067706] add replica on node 4199067706 for key 3 values battery lamp book
[storage node 4199067706] syncing 1 values in progress...
[storage node 4199067706] call put on next node http://127.0.0.1:39500/RPC2
[manager] server response: battery;lamp;book;
[storage node 4159399625] add replica #2
[storage node 4159399625] add replica on node 4159399625 for key 3 values battery lamp book
[storage node 4159399625] syncing 1 values in progress...
[storage node 4159399625] call put on next node http://127.0.0.1:56328/RPC2
[client] server response: battery;lamp;book;
 PASS
```

IV. Implementation Issues

There are a number of shortcomings in this implementation:

1) Client's API init() was slightly modified. So to initiate a client we can write client.init(client_id, "127.0.0.1", 8080), where 127.0.0.1 -is ip address of manager.

2) Test API single_set_get(int client_id, vector<string> value) was slightly modified. So you need to put the client's id and value inside it.

3) The side that uses the client's API is responsible for generating the key. Thus the key might be any value. In theory, the intersection of keys is possible. In this implementation, we are not checking key value for unique.

4) When checking the nodes for health, we only turn off the nodes. The process of adding nodes back is not implemented.