Project 4 - GTStore

Collaborators
Nihar Mehta (nmehta80)
Khushhall Chandra Mahajan (kmahajan32)

A. Introduction

In this project we implemented a distributed key-value store (GTStore) system. Nodes in this system communicate with network calls. Our implementation has the following system properties: scalability, availability and resilience to temporary node failures. This system is highly motivated by the shopping cart application.
The manager is always running and takes care of data storage. Thus it creates and removes nodes as and when required. Clients can come at any point of time and can make requests to add/remove items to the shopping cart and can make get requests to print their shopping carts. The system will ensure that the most up-to-date data is returned to the client. The system is very robust and ensures data partitioning, data replication and data consistency dynamically even in case of multiple node failures.

B. System Components

There are three major components in our implementation. We will describe each of them below:

- **GTStore**
    - **Centralized manager:** It handles all the control path decisions of the key-value store. We run the manager first and it creates various node processes for data storage. The manager is responsible of communication of client requests to storage nodes and communicating the storage responses back to the client. Thus it is like a mediator between storage nodes and the clients. It shadows the storage nodes so that the clients do not need to bother about how the data is being stored, replicated and updated. It uses a ring based data structure to store the nodes and the data at the position between the nodes. Any node can be dynamically added or removed at any point of time

without affecting the functionality of the shopping cart application. In this way, the manager handles node membership management. It also hashes the key of the data and requests the corresponding node to store the data and performs eventual consistency by delaying the data replication to other nodes. Thus, it also performs index management.

- **Storage/Data nodes:** These are processes created and terminated by the manager. These are the nodes which actually store the data. They are hosted on the different ports. They store the data as a map which saves the key of the client and maps it to the list of strings which are the items in the shopping cart of that particular client. A put request by a client dumps items in the shopping cart if they do not already exist. The get requests by the client returns the whole shopping cart back to the manager which sends it to the client who requested it. The erase requests by the client removes the specific data from the shopping cart.

- **Driver application:** This application consists of multiple test cases that test the functionality of the GTStore system. It creates M user-space processes (clients) that interact with GTStore. It tests putting data, getting data, erasing data into the shopping cart and multiple processes make requests in parallel to the manager. These interactions are RPC calls and hence are inherently queued which ensures that the manager handles one request at a time. But the clients can run in parallel and can make requests simultaneously. The returned data is the latest version of the data which is updated in the GTStore. New requests of put, erase can also be made and still the latest version will be returned by the manager.
- **Client library:** The driver applications uses the following functions to interact with the key-value store. The clients can put a list of data in the cart. They can erase a specific data if they added it erroneously. They can make get requests to get their updated shopping cart. They can finalize their requests to place the order and these will be flushed to the disk.

C. Data Structures

- **GTStoreStorage**
  - my_server (type xmlrpc_c::serverAbyss)
  - node_name
  - socket_number
  - node_data
  - master_node_map
  - init()
  - void init(string name, int socket_number);
  - void init(string name, string node, string data);
  - put()

- **GTStoreClient**
  - client_id
  - value (vector of string)
  - init()
  - finalize()
  - get(key)
  - put(key, value)
  - erase(key, value)

- **GTStoreManger**
  - max_ring_size,
  - prev_rpc_node_data, next_rpc_node_data (vector of string)
  - min_dist: Each nodes is assigned an id while creation. We want the new node to be separated by a minimum distance from the previous available nodes. This is to ensure that data is evenly distributed across the ring
  - replication_factor
  - ring, ring_reverse
  - process_map

- socket_number
- node_name, prev_rpc_node_name, next_rpc_node_name
- init(), init(max_ring_size);
- add_node(node_name)
- remove_node(node_name)
- print_nodes()
- insert_data(node_name, key, data, data, master_node)
- remove_data(node_name, key)
- find_node(int key)
- put_data(string key, string data)
- get_data(key)
- erase_data(key, data)
- pop_data(node_name, key, data)

D. RPC

We have used XML-RPC library for the RPC. It has a client and server. Multiple client can run concurrently. XML-RPC is a quick-and-easy way to make procedure calls over the Internet. It converts the procedure call into an XML document, sends it to a remote server using HTTP, and gets back the response as XML. This library provides a modular implementation of XML-RPC for C and C++. We show the basic operation on server and client side using xmlrpc_c:

Server side

```
class managerInsertMethod : public xmlrpc_c::method {
public:
        managerInsertMethod(GTStoreManager* manager) {
        // "i:s" defines the type of data being sent/receive
        this->_signature = "i:s";
    }
    void execute(xmlrpc_c::paramList const& paramList,
             xmlrpc_c::value *   const  retvalP) {
             string const node_name(paramList.getString(0));
```

```
            *retvalP = xmlrpc_c::value_int(my_manager->insert_data(node_name));
    }
};

xmlrpc_c::methodPtr const managerInsertMethodP(new managerInsertMethod(&manager));
// Add insert function
myRegistry.addMethod("manager.insert", managerInsertMethodP);

xmlrpc_c::serverAbyss myAbyssServer(
          xmlrpc_c::serverAbyss::constrOpt()
         .registryP(&myRegistry)
         .portNumber(7000));
myAbyssServer.run();
```

We first build the server by adding various methods like insert, remove, put, get, erase etc in the builder. Then we run it on port 7000. It is always running and expects multiple requests from multiple clients. One example of a method is illustrated above. A client request is received via an RPC call and the insert method is invoked. The request parameters are obtained the the manager makes appropriate request to the appropriate storage node to insert the data.

Client side

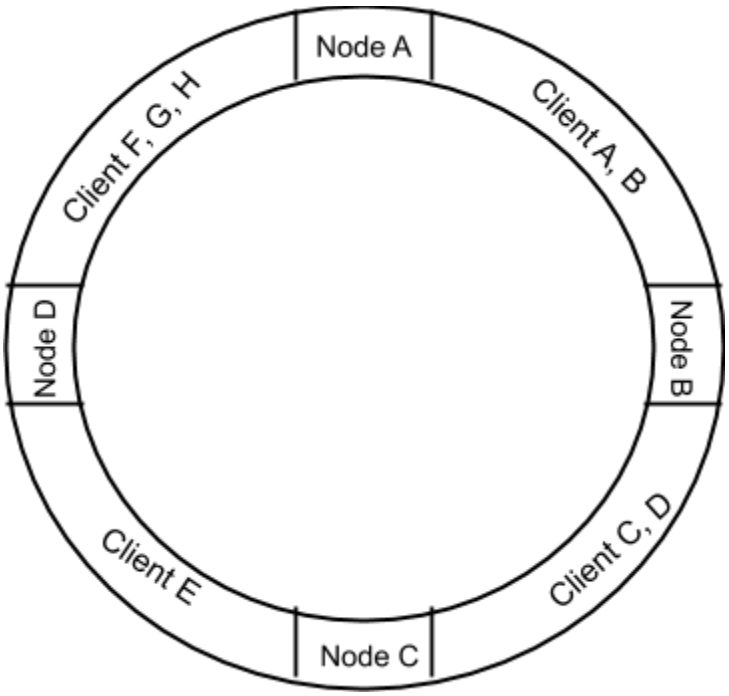
```
string const serverUrl("http://localhost:7000/manager");
string const methodName("manager.put");

xmlrpc_c::clientSimple myClient;
xmlrpc_c::value result;
myClient.call(serverUrl, methodName, "ss", &result, key.c_str(), value[i].c_str());
string const ret_key = xmlrpc_c::value_string(result);
```

The manager is always running on port 7000. Each client knows this and makes a request to the manager. In this example, the client is making a request to add a list of items of the shopping cart and accomplishes so by an RPC call to the manager.

E. Design Overview

- **Data Partitioning**

We want the GTStore to be highly scalable. We cannot store all the data in a single node. We have multiple nodes available. These nodes can be situated apart from each other. Moreover, all the nodes might not be up all the time. There might be node failures. Hence, the implementation should be able to save multiple copies of the data. We also want our system to handle large number of client requests if we increase the hardware resources. (Obviously, if we have limited number of system we cannot handle all the requests because of the resource crunch). We use data partitioning to distribute the load evenly across the multiple nodes. When a node is created, it is assigned a unique key. Suppose there are N nodes and we have maximum ring size as M. We evenly assign each node a number in the range 0 to M. While adding a new node we ensure that the consecutive nodes are apart at least some minimum distance apart. Next, comes assigning each client to a specific node, so that the resources are effectively utilised. We hash the client to get an id. This id is assigned to the nodes. Let's understand how this is done.



Selecting partition

Assume that Node A, B, C, D has an id 0, 90, 180, 270 respectively. Let maximum range size is 360. Now, each of the client will be hashed and get some value in the range 0-360. The hash value will be uniformly distributed. Say, the client A is hashed to 30, client B is hashed to 47, Client E is hashed to 251. Client A and B will be assigned to Node B and client E will be mapped to Node D. Node A, B, C, D has the following ranges (270, 360 or 0], (0,90], (90, 180], (180,270]. For the node A, 360 is treated as 0.

Pros and cons of design

We hash multiple times before assigning a particular id to node to ensure that there is minimum distance between the nodes. The way we are adding new nodes in the storage system ensures that they are uniformly distributed. Also, we are ensuring that there is enough space available for the data which will be inserted in the node. Thus, none of the nodes will get overloaded with the data considering each client also gets an id by uniform hashing. We achieve the goal of data partitioning by hashing the key to the position on the ring. The hash function is random and thus ensures even partitioning of the incoming data. The alternate way might be adding virtual nodes i.e. each nodes can be assigned multiple hash values. This would also help in uniform distribution of data/clients. Since, we are working with large number of nodes our current implementation is good. Our design is robust and it ensures that the uniform distribution of the nodes is not affected by addition or removal of nodes from the storage system. In the following section, we show that this robustness has been extended to data replication too. This means that on addition or deletion of nodes, the data is redistributed based on the replication factor and thus no data is lost even if there is a permanent failure in multiple nodes.

- **Data Replication**

We have many nodes available to us. However, some of the nodes may get down or maybe unresponsive due to some fault. For ex: It might be taking too long to connect to a particular node. In such case, we call that particular node as dead. If

we store all the data of a particular client in a single node, we will not be able to retrieve the data if that node is down. Hence, we use data replication to handle this issue of node failure. We replicate k versions of the client data in the k different nodes. Thus, if one node fails we still have K-1 copies of the client data and the data can be retrieved back. This will ensure that our system of nodes is fault tolerant and under certain consistency schemes increase the system availability.

Replication Protocol

Again, consider the node example from the previous section. Client A and B are assigned to Node B. Consider the case when k is 3. We will store 3 copies of the data. The data of client A and B are also store on next 2 nodes apart from node B. Thus, data of client of A and B is stored in Node B, C, D. Similarly, data of client C, D is store in Node C, D, A.

We have to handle base case carefully i.e. when there are less number of nodes M is less than k. In such scenario, we can store M copies of the data only. There are two things to do here for any insert operation for a new client. Hash the client and if it gets assigned to a new node, then insert the corresponding data to this new node. Second, copy the data from all the previous nodes since k<m.

Dynamic node addition/removal

A major implementation success is the dynamic addition and removal of nodes in the system while ensuring the replication protocol is maintained. Whenever a new node is added to the system, the data partitioning is dynamically changed and the new node stores some of the data of the next node in the ring. Moreover, due to these changes, the replicas of the data are restructured based on the new node structure very efficiently. Similarly, on removal of the node, the data in that node is partitioned to the next node and the replicas are restructured. At any point of time, the replication factor functionality is preserved.

- **Data Consistency**

Data consistency is important part of our implementation. We want to read the latest copy of the data. We are flexible on consistency model in our implementation. We make RPC call to the master node which in turn make replicas

of the data and puts on other nodes. In case the master node is down, we use the next available nodes as the master and make and save the copy of the data. This guarantees that we are always saving the latest copy on the master node and also replicating the data. This happens within bounded time. Once read operations are called we get the data from the master node. If master node fails, we move to the next replicated copy of the data and return from the other node via RPC call. Thus, read operations are guaranteed to receive the latest version of the data it requested.

- **Node Failure (Extra credit)**

Again, consider the previous example. Client A and B are assigned to Node B (master), C, D initially. Suppose the node B failed and we have only two copies of data available (for k=3). The failure of the node does not affect the uniform distribution of the data. It is dynamically handled by the manager. Thus data partitioning is taken care of in case of node failure too. Next, we will create a new backup of the data and the data of client A and B will be assigned to Node C, D, A. This process is a delayed process as the manager sends a request to the node next to the faulty node first (Node C) and if in case there are other requests, the manager will start processing them first. Since there is an updated copy of the data in the Node C, even if the request is to get the data, the manager still returns the updated version of the data and after the call gets back to the process of copying data to the nodes D and A. In this way, we ensure data replication in node failure case. Finally, since always updated data is returned, data consistency is intact too.

F. Implementation details

We will describe our codebase below:

- gtstore.hpp
    - It defines the functions to be declared in manager.cpp and storage.cpp
- manager.cpp
    - This is the centralized manager. It is first run on port 7000. It creates N storage nodes on N different ports. It creates a ring data structure which maps the position of the nodes on the ring to the name of the node and a ring_reverse data structure that map the other way round.

It receives client requests by RPC calls and processes them by making RPC calls to the appropriate storage nodes. The insert_data function ensures that the data is stored in a particular node. It is stored along with a master node entry which defines the original node where the data was inserted into. The put_data function invokes the insert_data function on the master_node and then uses eventual consistency to create its replicas on subsequent nodes equal to the replication_factor. The get method finds the node where the data was pushed to by the key and makes a call to retrieve the data from the node. If the node has failed, it obtains a copy from the nodes which have a replica of the data. The erase_data function removes the data from the corresponding node and also removes all the replicas from the subsequent nodes. The add_nodes and remove_nodes dynamically ensure membership management.

- storage.cpp
    - Once the storage node has been created by the manager, it hosts itself on the socket number provided by the manager. It consists of a local copy of maps node_data and master_node_map. The node data maps keys to a vector of data. The master_node_map maps the key to its master_node. If the master_node is not the node itself, then it means that it is a replica of some preceding node. The storage node receives requests from the manager to put, erase and get data. These methods ensure that the data is managed accordingly. On remove_node call by manager, the storage node terminates the server and the process exits thus deleting all local data. But, the manager takes care that this data is handled and shifted to other master_node and replicated to its subsequent nodes.
- client.cpp
    - get(key): This function get all the data for this client from the node. It ensures that latest version of the data is retrieved. Similar to a shopping cart which might have multiple items in it, this get function will fetch us array of the data it has saved previously. It uses RPC call to fetch the data from the corresponding node through the manager.

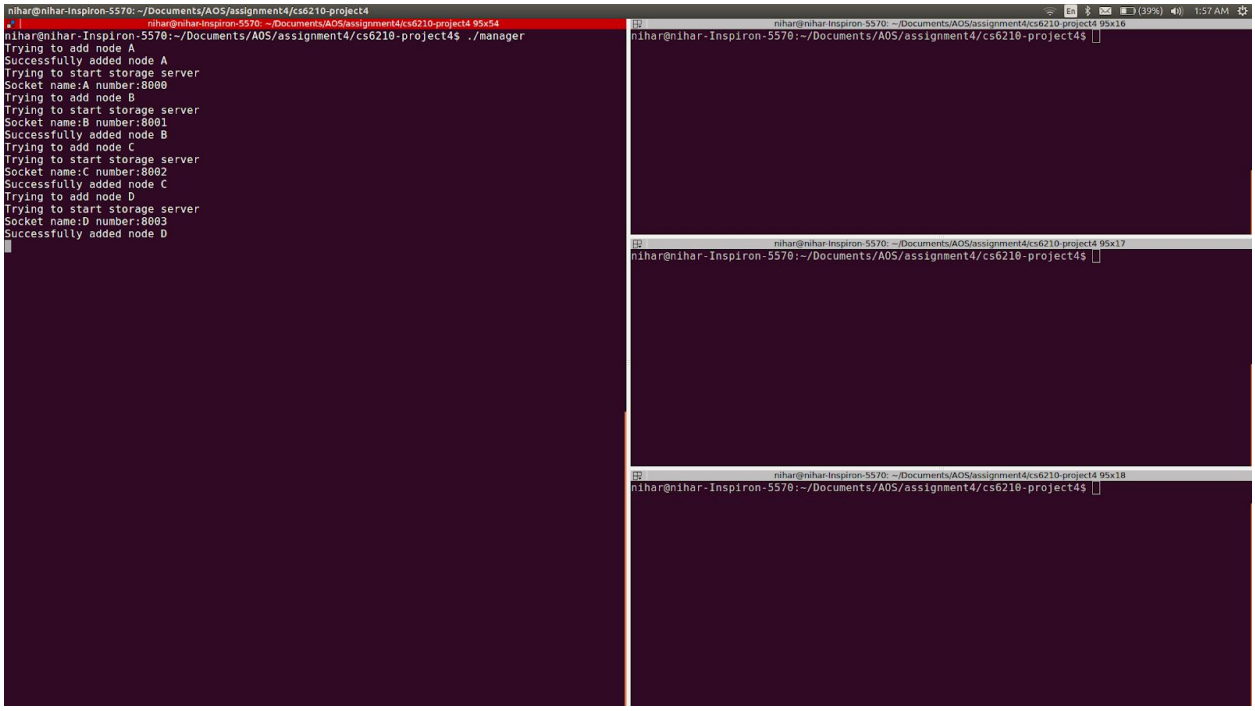
    - put(key, value): It saves the list of elements (items in the shopping cart) by making a RPC call to the node through the manager. The data is saved to the node and copies of data get replicated to other nodes.
- test_app.cpp
    - It has test cases for our implementation. It call the clients and read/write/erase the data.
- Makefile
    - In the Makefile, we compile the manager and test_app
- run.sh
    - It makes the all the GTStore code and run the manager, storage. It then calls the test_app to test our implementation.

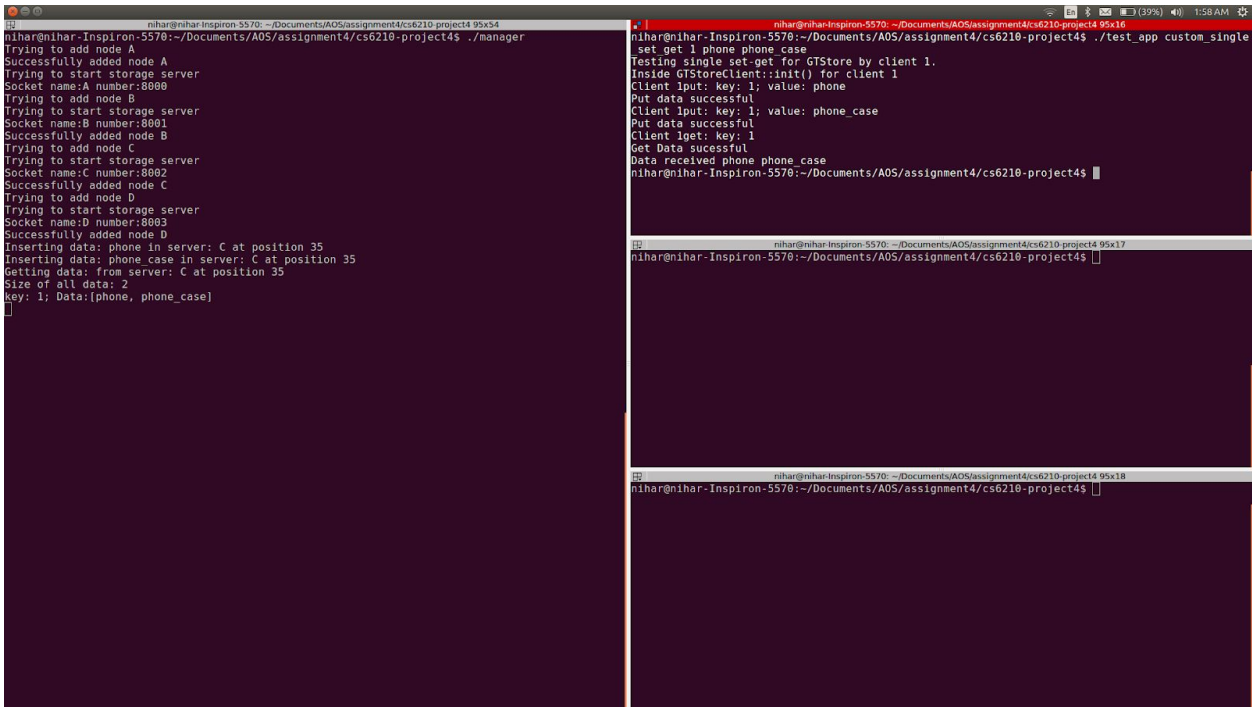
G. Client driver application

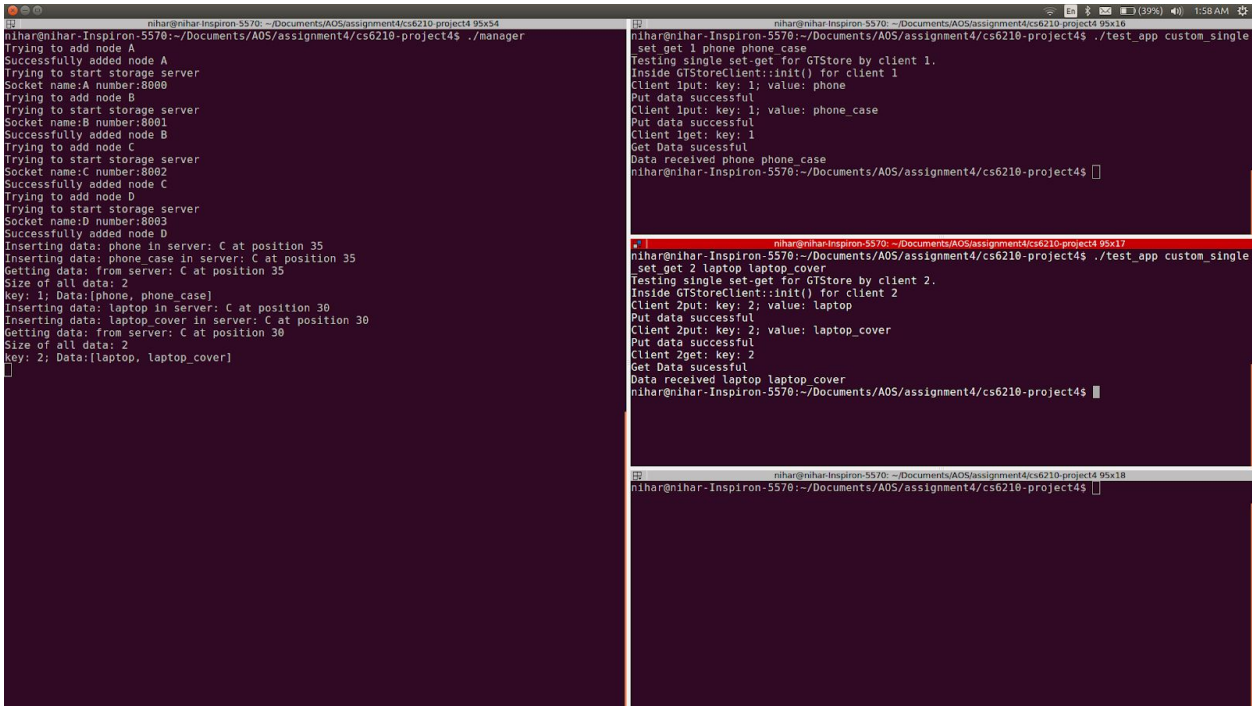
We include the logs below:

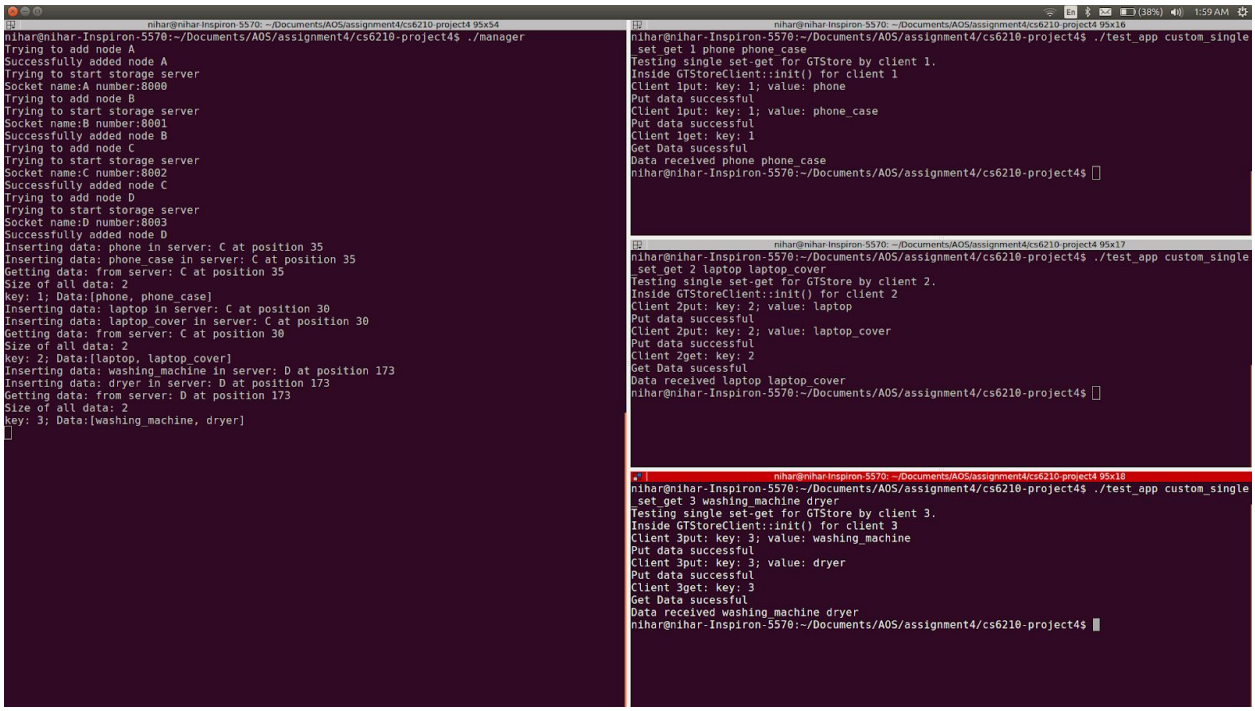
Testing put/get

- In the first image shown, we first run the manager. It adds 4 nodes initially for the storage (using the storage code).
- It also start 4 sockets with different port for each node. Node A, B, C, D are connected to socket 8000, 8001, 8002, 8003 respectively. Now we can make RPC at these ports for data transfer
- In the second image, we run test_app_custom single. In this, we try to put and get data to/from node 1. We first add items phone and phone_case to node_1 using RPC and then get from node using Node1.
- In the third image, we run test_app_custom single. In this, we try to put and get data to/from node 2. We first add items laptop and laptop_cover to node_2 using RPC and then get from node using Node2.
- In the fourth image, we run test_app_custom single. In this, we try to put and get data to/from node 3`. We first add items washing_machine and dryer to node_3 using RPC and then get from node using Node3.
- In all these cases manager is always up and coordinates all the operations with the storage nodes.

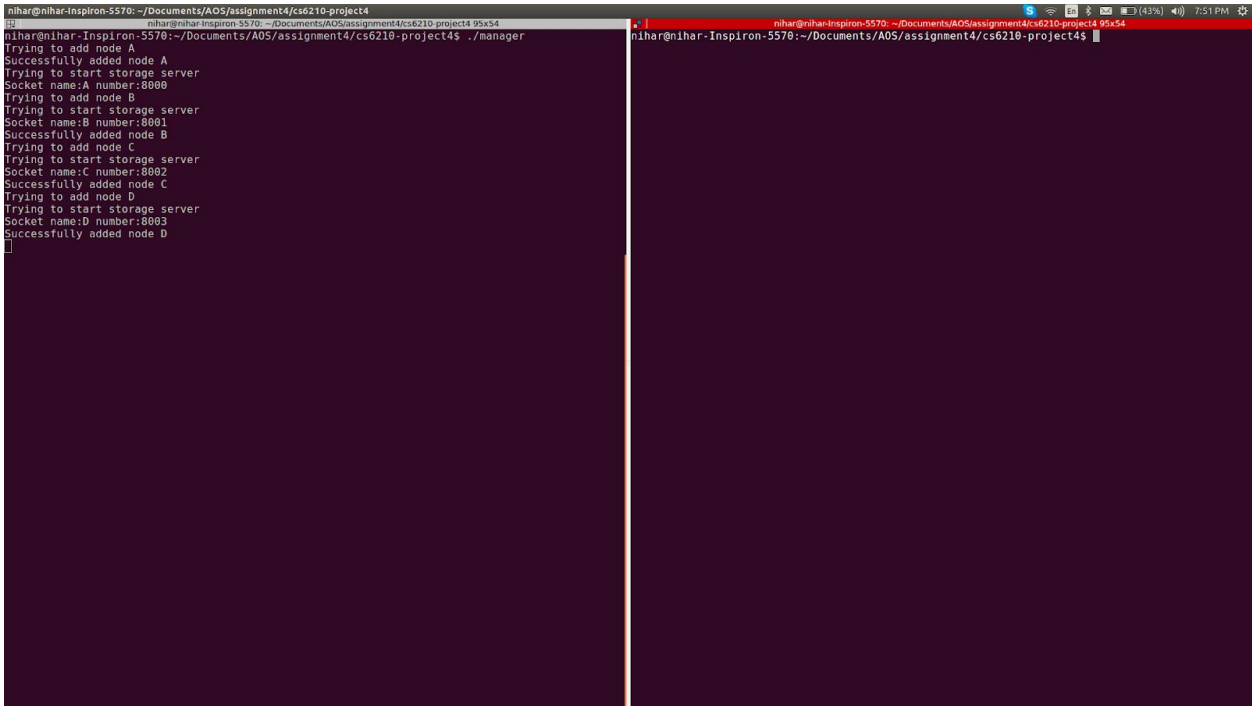

nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x54
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./manager
Trying to add node A
Successfully added node A
Trying to start storage server
Socket name:A number:8000
Trying to add node B
Trying to start storage server
Socket name:B number:8001
Successfully added node B
Trying to add node C
Trying to start storage server
Socket name:C number:8002
Successfully added node C
Trying to add node D
Trying to start storage server
Socket name:D number:8003
Successfully added node D
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x16
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x17
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x18
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$

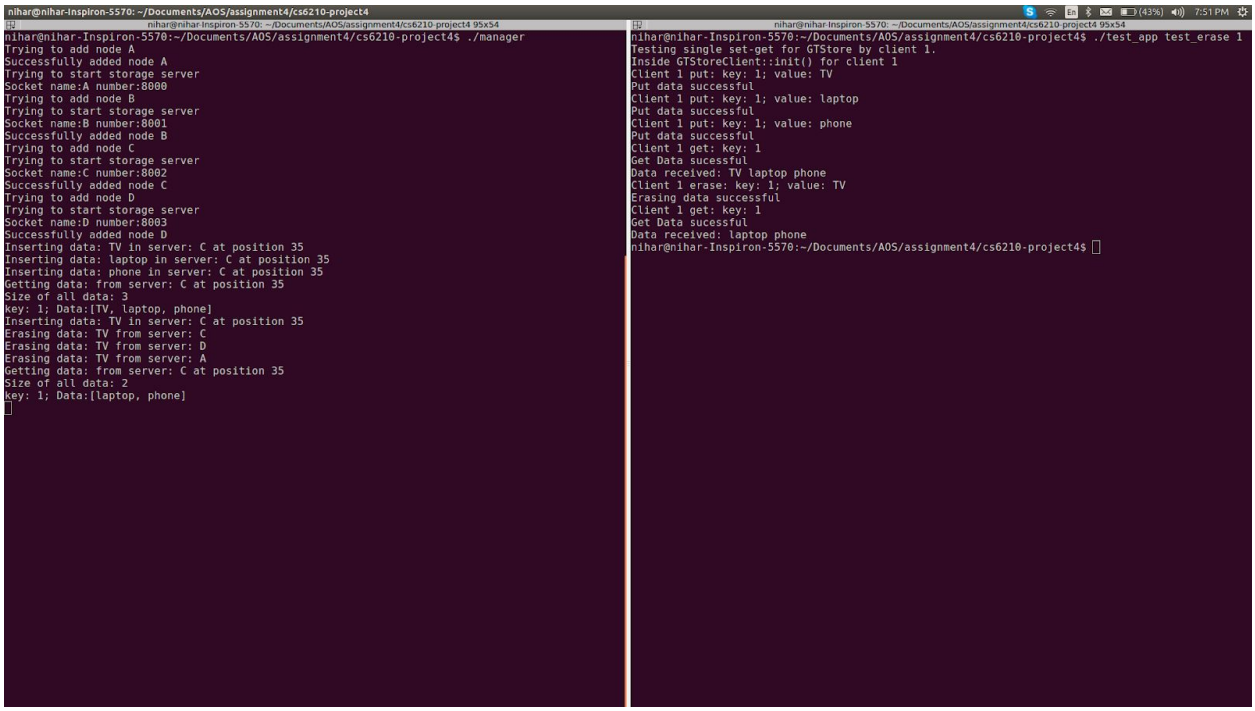

nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x54
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./manager
Trying to add node A
Successfully added node A
Trying to start storage server
Socket name:A number:8000
Trying to add node B
Trying to start storage server
Socket name:B number:8001
Successfully added node B
Trying to add node C
Trying to start storage server
Socket name:C number:8002
Successfully added node C
Trying to add node D
Trying to start storage server
Socket name:D number:8003
Successfully added node D
Inserting data: phone in server: C at position 35
Inserting data: phone_case in server: C at position 35
Getting data: from server: C at position 35
Size of all data: 2
key: 1; Data:[phone, phone_case]
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x16
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 1 phone phone_case
Testing single set-get for GTStore by client 1.
Inside GTStoreClient::init() for client 1
Client 1put: key: 1; value: phone
Put data successful
Client 1put: key: 1; value: phone_case
Put data successful
Client 1get: key: 1
Get Data sucessful
Data received phone phone_case
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x17
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x18
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$


nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x54
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./manager
Trying to add node A
Successfully added node A
Trying to start storage server
Socket name:A number:8000
Trying to add node B
Trying to start storage server
Socket name:B number:8001
Successfully added node B
Trying to add node C
Trying to start storage server
Socket name:C number:8002
Successfully added node C
Trying to add node D
Trying to start storage server
Socket name:D number:8003
Successfully added node D
Inserting data: phone in server: C at position 35
Inserting data: phone_case in server: C at position 35
Getting data: from server: C at position 35
Size of all data: 2
key: 1; Data:[phone, phone_case]
Inserting data: laptop in server: C at position 30
Inserting data: laptop_cover in server: C at position 30
Getting data: from server: C at position 30
Size of all data: 2
key: 2; Data:[laptop, laptop_cover]
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x16
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 1 phone phone_case
Testing single set-get for GTStore by client 1.
Inside GTStoreClient::init() for client 1
Client 1put: key: 1; value: phone
Put data successful
Client 1put: key: 1; value: phone_case
Put data successful
Client 1get: key: 1
Get Data sucessful
Data received phone phone_case
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x17
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 2 laptop laptop_cover
Testing single set-get for GTStore by client 2.
Inside GTStoreClient::init() for client 2
Client 2put: key: 2; value: laptop
Put data successful
Client 2put: key: 2; value: laptop_cover
Put data successful
Client 2get: key: 2
Get Data sucessful
Data received laptop laptop_cover
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x18
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$


nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x54
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./manager
Trying to add node A
Successfully added node A
Trying to start storage server
Socket name:A number:8000
Trying to add node B
Trying to start storage server
Socket name:B number:8001
Successfully added node B
Trying to add node C
Trying to start storage server
Socket name:C number:8002
Successfully added node C
Trying to add node D
Trying to start storage server
Socket name:D number:8003
Successfully added node D
Inserting data: phone in server: C at position 35
Inserting data: phone_case in server: C at position 35
Getting data: from server: C at position 35
Size of all data: 2
key: 1; Data:[phone, phone_case]
Inserting data: laptop in server: C at position 30
Inserting data: laptop_cover in server: C at position 30
Getting data: from server: C at position 30
Size of all data: 2
key: 2; Data:[laptop, laptop_cover]
Inserting data: washing_machine in server: D at position 173
Inserting data: dryer in server: D at position 173
Getting data: from server: D at position 173
Size of all data: 2
key: 3; Data:[washing_machine, dryer]
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x16
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 1 phone phone_case
Testing single set-get for GTStore by client 1.
Inside GTStoreClient::init() for client 1
Client 1put: key: 1; value: phone
Put data successful
Client 1put: key: 1; value: phone_case
Put data successful
Client 1get: key: 1
Get Data sucessful
Data received phone phone_case
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x17
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 2 laptop laptop_cover
Testing single set-get for GTStore by client 2.
Inside GTStoreClient::init() for client 2
Client 2put: key: 2; value: laptop
Put data successful
Client 2put: key: 2; value: laptop_cover
Put data successful
Client 2get: key: 2
Get Data sucessful
Data received laptop laptop_cover
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$
nihar@nihar-Inspiron-5570: ~/Documents/AOS/assignment4/cs6210-project4 95x18
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$ ./test_app custom_single_set_get 3 washing_machine dryer
Testing single set-get for GTStore by client 3.
Inside GTStoreClient::init() for client 3
Client 3put: key: 3; value: washing_machine
Put data successful
Client 3put: key: 3; value: dryer
Put data successful
Client 3get: key: 3
Get Data sucessful
Data received washing_machine dryer
nihar@nihar-Inspiron-5570:~/Documents/AOS/assignment4/cs6210-project4$

Testing erase data

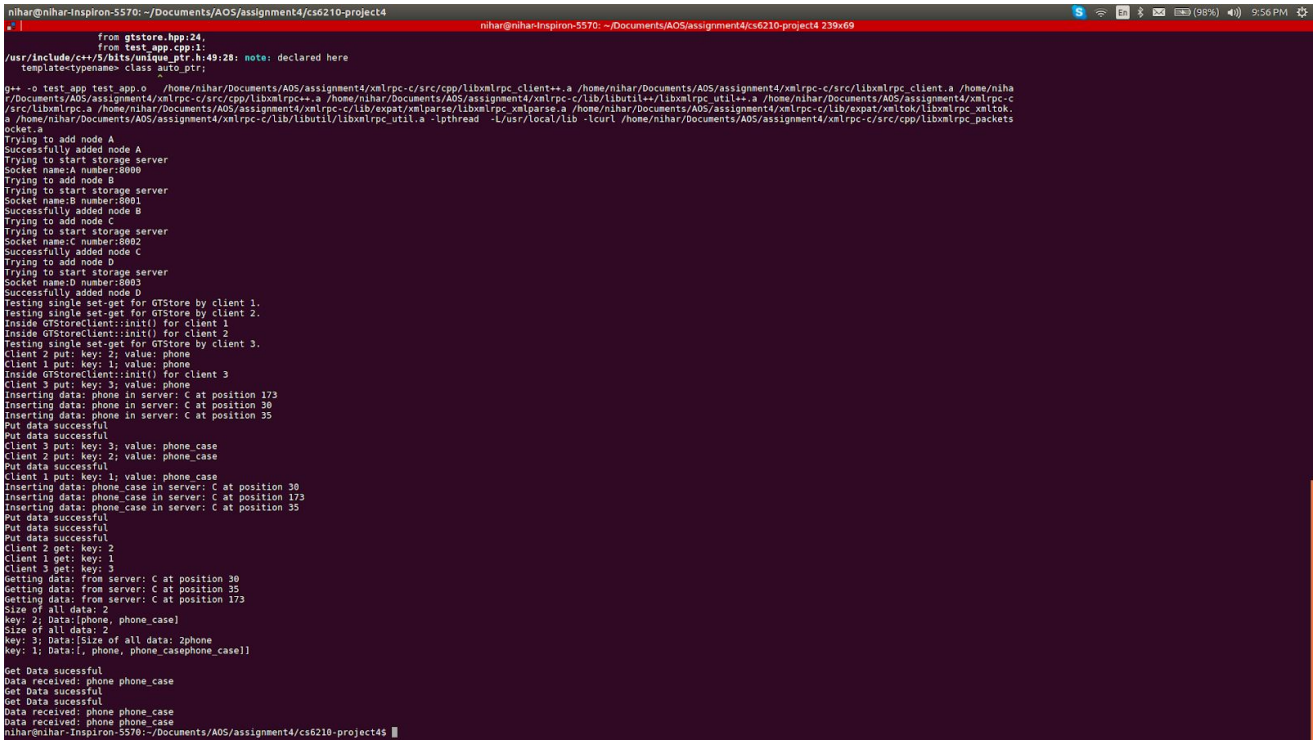




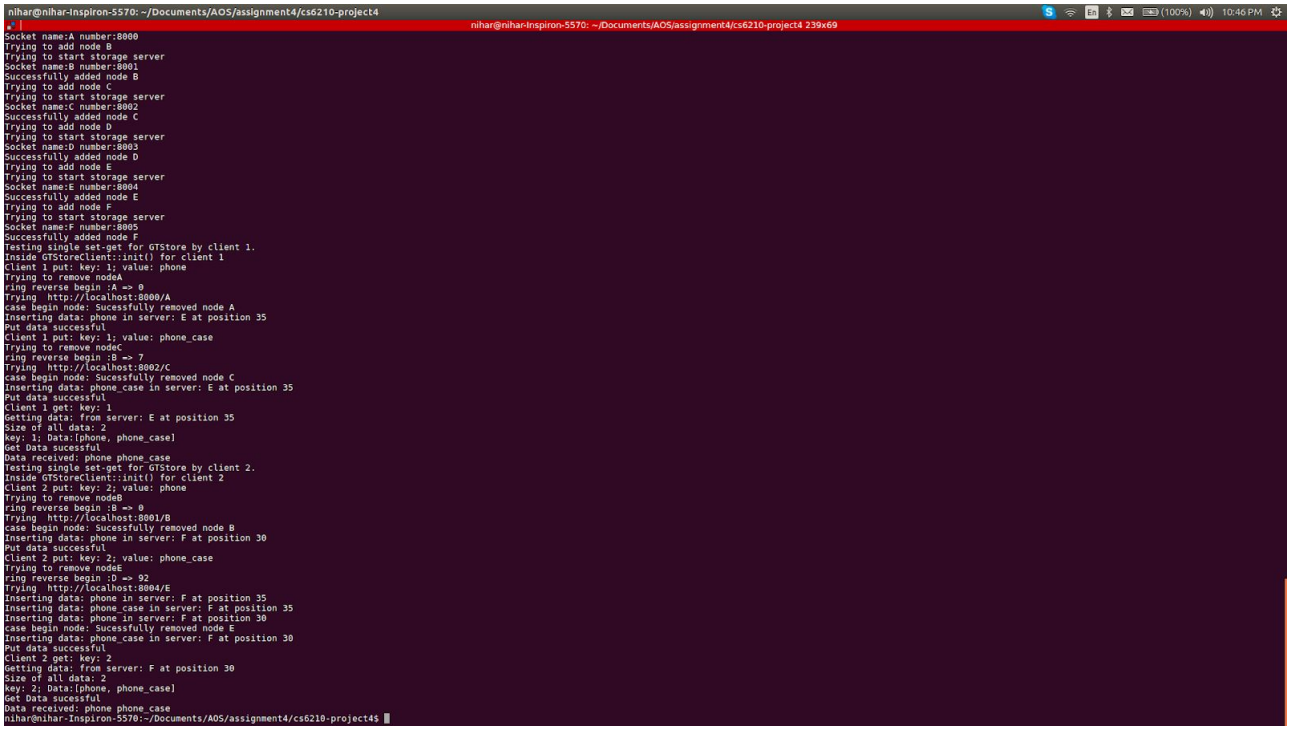
- In the first image shown, we first run the manager. It adds 4 nodes A, B, C, D initially for the storage (using the storage code).
- Node A, B, C, D are connected to socket 8000, 8001, 8002, 8003 respectively. Now we can make RPC at these ports for data transfer
- Now, client 1 perform the put task and add TV, laptop, phone. We show the saved content and then tries to erase the item TV from the list. We then show the item list after the erase operation. It shows only laptop and phone.
- In all these cases manager is always up and coordinates all the operations with the storage nodes.

Test run.sh

This script makes the code and run the manger. It then runs the client 3 times.



Testing run_add_remove_node.sh



In add_remove_node.sh, we demonstrate how nodes even if removed do not affect the functionality of the shopping cart.

H. Implementation issues

The implementation doesn't use virtual nodes for storing the data. The virtual nodes can boost the protection of the system and also contribute to improving the load balancing efficiency of the system. It does not use vector time clocks for data versioning. Instead it always ensures that the updated data is always returned on any get call.

Vector time clocks could help increase the efficiency of response by obtaining the data/replica from random nodes and computing the most recent data out of it. But here, we ensure that master node is transferred as soon as any node is faulty and thus, we directly get the data from the new master node if the original node is faulty.

The nodes do not use a flag which set the status as active/passive but instead are removed as soon as the manager finds the node to be unresponsive.

I. Design tradeoffs

The above decisions have been made so that the system remains totally robust to data partitioning, replication and versioning schemes. The efficiency has been compromised by not using virtual nodes or vector clocks but the functionality is preserved. The virtual nodes are required when there are billions of clients and since this is a small project, we decided not to go with virtual nodes.

Vector clocks can retrieve different versions of data but we ensured we get the latest data from the latest master nodes. Adding vector clocks functionality can be an interesting future work.

Another design consideration was that we are storing all data of a particular client together in specific nodes. Even the replicas consist of all data of the particular client. This is done because it is much faster to invoke rpc call to one storage server and retrieve the entire shopping cart on a get request without any latency. If it were the case that the data is stored on multiple servers, it will result in delays due to the RPC calls. Moreover, we store the data as a vector which is mapped to the client key and since the key can be stored in a specific node only, the whole client data is bundled together in the same node itself.

Running/ Testing our code

- First, install XMLRPC-c, go to the folder xmlrpc-c and run the following command:
    - sudo ./configure
    - sudo make
    - sudo make install
    - sudo ldconfig
- Xmlrpc-c library should be installed on the system.
- The Makefile can be found in the base directory. Open the terminal and use **make** command to make it.
- We run ./manager to begin the manger. In turn storage also gets initialised.
- Open a new terminal, and run our test_app.
    - To test add, run ./test_app custom_single_set_get node_id <item names separated by space>

- We can also test using run.sh. Simply open a terminal and use ./run.sh
- We can also test add/remove node using run_add_remove_node.sh

- Our code is running properly without any issues.

J. Conclusion

In this project, we implemented distributed key-value store (GTStore) system. GTStore system have multiple nodes and it distributes the incoming load. It scales well with the increasing number of client given enough hardware resources. The manager is the main gateway for the clients and the available storage. It decides where to store/get the data from. The implementation mimics the example of shopping cart application. We have taken care that the GTStore is resilience to node failures (even multiple nodes might fail), data partitioning for robust load balancing and the data stored/retrieved is consistent.

K. References

http://xmlrpc-c.sourceforge.net/

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... & Vogels, W. (2007, October). Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (Vol. 41, No. 6, pp. 205-220). ACM.