

CS 4210 Project 4 Report

Jianing Yang & Xiaochuang Han

Introduction

In this project, we built a highly-scalable, highly-available and highly-reliable distributed key-value storage system called GTStore. The implementation exercised a few core concepts of Amazon Dynamo.

Design

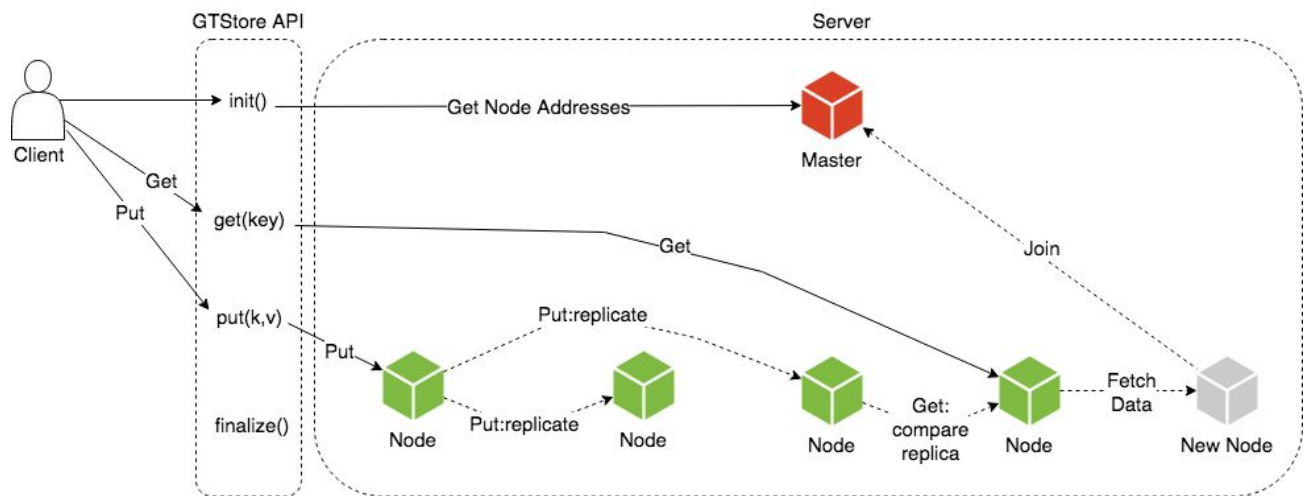


Figure 1. Design Overview

As shown above, our system consists of layers: GTStore API layer and Server layer. A user of the system only interacts with the API layer, and the API layer automatically figures out how to communicate with the server and manipulate the data.

GTStore API Layer

(Code in `gtstore_api.py`)

We provide a simple API interface to users which only consist of four functions:

- `init()`
- `get(key)`
- `put(key, value)`
- `finalize()`

The names of these functions are self-explanatory. A user should call `init()` before they start to interact with the data system. `get()` and `put()` are used to fetch/update key-value pairs. `finalize()` in our implementation does nothing - it is listed here only for the sake of completeness.

These functions simply initiate customized protocols (built on TCP)

Server Layer

The server layer consists of two types of infrastructure:

- Master (*Code in master.py*)
- Node (*Code in node.py*)

Master is a centralized manager whose responsibilities include

1. Manage *Nodes* membership (*Node* join, *Node* heartbeat, etc.)
2. Load balancing of client connections across *Nodes*
3. Maintain consistent hashing (broadcast new node ring to all *Nodes*)

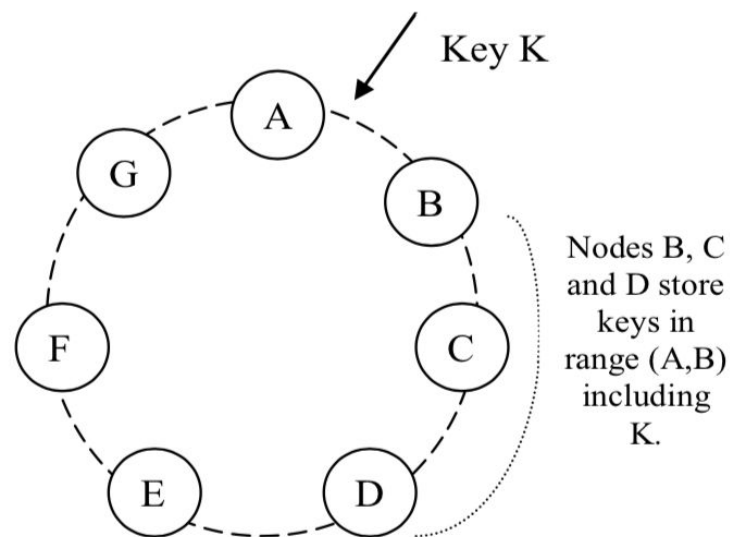


Figure 2. Node Ring

The master manages nodes in a Node Ring data structure shown above. Based on this structure, the master also maintain a consistent hashing function (details discuss later) across nodes. Whenever the node ring changes, the master is responsible for broadcasting the new node ring to all nodes. Master also orchestrates database transfers between nodes should that be necessary.

First time client connection always does a one-time communication with the master to figure out which data node(s) the client should send its request to. After this, the client will only communicate with its assigned gateway node(s).

Node

Node is the unit that actually stores the data. A node in our implementation is a virtual concept of a data storage unit. One physical machine can have many nodes on it as long as its hardware capacity allows that. Since we use Python, starting a new node is as simple as creating a new Node class object. When created, the node will communicate with the master to request to be joined into the Node Ring. Once joined, a node is given its own hash range and it will fetch database from nearby neighbors. After that, a node becomes fully functional and can take request from clients.

A node is capable of being a gateway processor for a client request, meaning that a node can relay put/get request if the key is not in the range of its hash range. When a put/get request comes in, a node is also capable of figuring out replication/eventual consistency. More details are discussed below.

Incremental Scalability

Our system achieved an incremental scalability using the consistent hashing mechanism similar to the one used by Dynamo. The hashing function always takes in a key and returns a float hashcode between 0 and 1, which represents a position on a ring. The nodes together make up a ring, each node becoming responsible for the region in the ring between it and its predecessor node on the ring. The position of each node on the ring is represented by an assigned value from the central manager. This value, unlike Dynamo's random assign, is picked to be within the largest node interval so that we could have a very even distribution of nodes on the ring. For example, the first node coming into the system would get a node value of 0. The second node would get 0.5. The third, fourth and fifth node would get 0.25, 0.75, 0.125, respectively. We believe that this improvement over Dynamo could potentially give each node a more uniform pressure of handling key-value request. The tradeoff could be a longer node joining time.

Why not using a different hashing function each time a node is inserted? (i.e. with 5 nodes, key 25 gets hashcode $25 \bmod 5 = 0$; with 6 nodes, key 25 gets hashcode $25 \bmod 6 = 1$) This is because that using a different hashing function would completely rearrange the key-node mapping so that every node gets a different bunch of keys to handle after each node insertion or deletion. Apparently, this is not acceptable. In contrast, using our consistent hashing function, when a new node comes or leaves, it only affects its next node on the ring.

Our system successfully supports online node insertion and removal. To be more specific, each time a node is inserted, it gets the keys and corresponding values of which it should be responsible from its next node. Similarly, when a node leaves, it pushes its keys and corresponding values to its next node, transferring the responsibility of its ring region to its next node. See the test cases at the bottom for a demo.

Dual Availability

The availability of our system is mainly reflected through two dimensions: client-level availability and key-level availability.

What is client-level availability and why is it important? Upon a client's initialization, it would contact the central manager for a list of random gateway nodes. Later, the client would only send the put and get requests through these gateway nodes. We believe that this is a good load balancing over clients. Note that these gateway nodes are not special nodes. They also take the responsibility of saving key-value pairs. The client-level availability is reflected in a way that if one gateway node is down, the client could still connect to the overall node network using other gateway nodes. And since each node has a complete copy of the node ring, the client could successfully insert the key and the corresponding value knowing the key's consistent hashcode (a float number between 0 to 1) and the node ring.

What is key-level availability and why is it important? With a key's hashcode and the node ring, we can insert the data into the first node on the ring with position greater than the hashcode's corresponding position. However, if this node is down some time later, we would lose the data. To tackle this problem, we go on and save the data with the current timestamp on K (we choose 3 in our project) consecutive nodes after the initial node responsible for the data. Within a bounded time, if a majority of the K nodes successfully put the data, we consider the put request successful. The key-level availability is reflected when a client tries to get data from our system. When a client connects to one of its gateway node and request a get operation, the gateway node would send get requests to the K nodes having the relevant data. If a majority of the nodes reply, we take the data with the latest timestamp among the returned results and give that back to the client. Overall, both put and get operations are highly available.

High Reliability [Extra Credit for Handling Temporary Failures]

The scalability section above talked a lot about online node insertion and removal, but what about the node silently dies? Here we introduces a heartbeat mechanism that

each node reports IS_ALIVE to the central manager every T seconds. If the central manager fails to get any signal from a node in $N \cdot T$ seconds, it would remove the node, recover the data on the node to its next node by looking at the K replicas, and broadcast the new node ring. When the node resolves the failure and comes back to the network, it would connect with the central manager as its first time and get joined in the node ring (the central manager would broadcast this change as usual). The reliability of our system is reflected in a way that the leaving and rejoining of the nodes all happen online without a halt to the overall system. See the test cases for more details.

Test Cases

Test Case: Put/Get

```
# client.py
if __name__ == "__main__":

    conn = DBConnection('localhost', 4210) # Initialization

    conn.put("Jed's cart", "Apple Banana Orange")
    conn.put("Han's cart", "Grape Steak Juice")
    conn.put("Yang's cart", "Noodle Pepper Milk")
    conn.put("Jianing's cart", "Eggs Pork Butter Napkins")

    print("Jed's cart: " + str(conn.get("Jed's cart")))
    print("Han's cart: " + str(conn.get("Han's cart")))
    print("Yang's cart: " + str(conn.get("Yang's cart")))
    print("Jianing's cart: " + str(conn.get("Jianing's cart")))

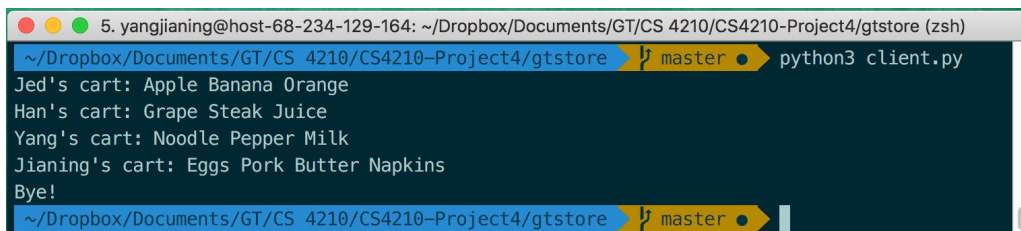
    conn.finalize()
```

```
# client_extra.py
if __name__ == "__main__":

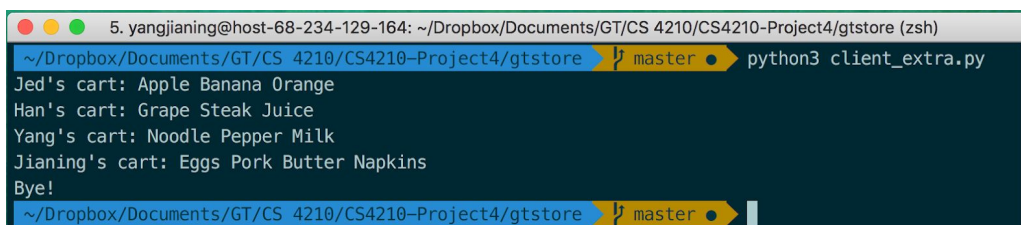
    conn = DBConnection('localhost', 4210) # Initialization

    print("Jed's cart: " + str(conn.get("Jed's cart")))
    print("Han's cart: " + str(conn.get("Han's cart")))
    print("Yang's cart: " + str(conn.get("Yang's cart")))
    print("Jianing's cart: " + str(conn.get("Jianing's cart")))

    conn.finalize()
```



A terminal window showing the execution of client.py. The prompt is 5. yangjianing@host-68-234-129-164: ~/Dropbox/Documents/GT/CS 4210/CS4210-Project4/gtstore (zsh). The command is python3 client.py. The output is: Jed's cart: Apple Banana Orange, Han's cart: Grape Steak Juice, Yang's cart: Noodle Pepper Milk, Jianing's cart: Eggs Pork Butter Napkins, Bye!.



A terminal window showing the execution of client_extra.py. The prompt is 5. yangjianing@host-68-234-129-164: ~/Dropbox/Documents/GT/CS 4210/CS4210-Project4/gtstore (zsh). The command is python3 client_extra.py. The output is: Jed's cart: Apple Banana Orange, Han's cart: Grape Steak Juice, Yang's cart: Noodle Pepper Milk, Jianing's cart: Eggs Pork Butter Napkins, Bye!.

This test case shows that our system can successfully get/put user data. Client_extra.py shows that even in another machine (process), we can access the data that was previously stored by some other user.

Test Case: Online Node Join (Extra Credit)

```
# server.py
if __name__ == "__main__":

    # create master
    master = Master('localhost', 4210)
    mlt1 = MasterListenThread(master)
    mlt1.start()

    # create nodes
    node_list = []
    for i in range(M):
        node_list.append(Node('localhost', 4210 + i + 1))
        t = NodeListenThread(node_list[-1])
        t.start()
        node_list[-1].notify_master_node_status('localhost', 4210, HEADER_NODE_ACTIVE)

    # don't return so that we can see console output
    mlt1.join()
```

```
# server_extra.py
if __name__ == "__main__":

    # create nodes
    node_list = []
    for i in range(M):
        node_list.append(Node('localhost', 6210 + i + 1))
        t = NodeListenThread(node_list[-1])
        t.start()
        node_list[-1].notify_master_node_status('localhost', 4210, HEADER_NODE_ACTIVE)
```

```
5. python3 server.py (Python)
19): 0.0625, ('localhost', 4220): 0.1875}
NodeClientThread: Successfully ended the connection from 127.0.0.1:50738
NodeClientThread: Successfully updated node ring, sending OK to master 127.0.0.1:50737
NodeClientThread: Successfully ended the connection from 127.0.0.1:50735
NodeClientThread: Successfully updated node ring, sending OK to master 127.0.0.1:50731
NodeClientThread: Successfully ended the connection from 127.0.0.1:50739
NodeClientThread: Successfully ended the connection from 127.0.0.1:50737
NodeClientThread: Successfully updated node ring, sending OK to master 127.0.0.1:50734
NodeClientThread: Successfully ended the connection from 127.0.0.1:50731
NodeClientThread: Successfully ended the connection from 127.0.0.1:50734

Broadcasted node ring: {('localhost', 4211): 0, ('localhost', 4212): 0.5, ('localhost', 4213): 0.25, ('localhost', 4214): 0.75, ('localhost', 4215): 0.125, ('localhost', 4216): 0.375, ('localhost', 4217): 0.625, ('localhost', 4218): 0.875, ('localhost', 4219): 0.0625, ('localhost', 4220): 0.1875}

MasterClientThread: Successfully ended the connection from 127.0.0.1:50728
```

```
6. python3 server_extra.py (Python)
~/Dropbox/Documents/GT/CS 4210/CS4210-Project4/gtstore master python3 server_extra.py
NodeGetDatabaseThread: Successfully merged database from localhost:4216 New database: {}
NodeClientThread: Get connected from 127.0.0.1:50758
NodeClientThread: MASTER UPDATE NODE RING received from 127.0.0.1:50758! Updating local node ring
Node: localhost:6211 updated node_ring: {('localhost', 4211): 0, ('localhost', 4212): 0.5, ('localhost', 4213): 0.25, ('localhost', 4214): 0.75, ('localhost', 4215): 0.125, ('localhost', 4216): 0.375, ('localhost', 4217): 0.625, ('localhost', 4218): 0.875, ('localhost', 4219): 0.0625, ('localhost', 4220): 0.1875, ('localhost', 6211): 0.3125}
NodeClientThread: Successfully updated node ring, sending OK to master 127.0.0.1:50758
NodeClientThread: Successfully ended the connection from 127.0.0.1:50758
NodeGetDatabaseThread: Successfully merged database from localhost:4212 New database: {}
NodeClientThread: Get connected from 127.0.0.1:50760
NodeClientThread: Get connected from 127.0.0.1:50771
```

```
5. python3 server.py (Python)

Broadcasted node ring: {('localhost', 4211): 0, ('localhost', 4212): 0.5, ('localhost', 4213): 0.25, ('localhost', 4214): 0.75, ('localhost', 4215): 0.125, ('localhost', 4216): 0.375, ('localhost', 4217): 0.625, ('localhost', 4218): 0.875, ('localhost', 4219): 0.0625, ('localhost', 4220): 0.1875, ('localhost', 6211): 0.3125, ('localhost', 6212): 0.4375, ('localhost', 6213): 0.5625, ('localhost', 6214): 0.6875, ('localhost', 6215): 0.8125, ('localhost', 6216): 0.9375, ('localhost', 6217): 0.03125, ('localhost', 6218): 0.09375, ('localhost', 6219): 0.15625, ('localhost', 6220): 0.21875}

MasterClientThread: Successfully ended the connection from 127.0.0.1:50896
```

In this test case, we first launch server.py, which starts the Master and 10 Nodes. At that point, as shown in the first Terminal window, there are 10 nodes in Node Ring. Then, **without stopping the Master and the previous 10 nodes**, we start up another 10 nodes. As shown in the third Terminal window, these 10 new nodes are successfully added to the Node Ring.