

Distributed Key-Value Store

1. Introduction

1.1 Overview

This project is an implementation of a distributed key-value store whose goals are scalability, availability and system resilience to temporary node failure. Reliability at a massive scale is one of them biggest challenges companies like Amazon, Google and Facebook face today. A node failure could cause significant financial consequences, and diminish user trust in the system.

Additionally, GTStore is built to run over the network. Therefore all of the three components - Clients, Nodes and Centralized Manager - are assigned an IP (localhost for the sake of testing) and a port. The communication system utilized in this project is socket, and the protocol is TCP/IP.

Results show that GTStore overcomes these problems scaling up as needed, and recovering from node failures. Also, the key-values are shared among the nodes themselves making data highly available.

1.2 Disclaimer

This project simulates a customer shopping online, adding items to the cart. Therefore the name client will also refer to the client id (or username), and it's static during the entirety of a session. For example, if a client logs in with key = Mohan or Ranjan, the key will persist until client logs out, and re-login with new key.

2. Components

GTStore simulates the interaction between client and server. To make this possible the project has three main components, and each one of them have their key role in the system.

2.1 Centralized Manager

The Centralized Manager's main job is routing and mapping. Clients and nodes often communicate with CM when they need to know where to get data from, or where to send data to. CM's main functionalities are:

- `init()`: initializes maps and routers and overall system functionalities.
- `Query(char key[])` : maps key to node where value is possibly stored, and returns the node port.
- `Node_Down(int from_node , int failed_node)`: remaps all routes that were previously pointing to the failed node and assign another node to be used as replica.
- `replica(int node)`: returns a node that stores the replica
- `heart_beat()`: periodically checks whether or not nodes are down.

2.2 Node

The nodes are responsible for storing the key-value sent to them by the clients, and making sure data store in them are consistent, highly available and that its response falls into an agreed-upon threshold. **To avoid clogging CM**, nodes are able to talk to one another to request replicas, send replicas, send **put** requests to be saved to replicas and so on. For that, the nodes' main functionalities are:

- `init()`: sets IP and port, and requests node replicas from CM.
- `put(key,value)`: stores the key-value pair sent by the client. It also sends the key-value pair to its replica nodes. **As a design decision, if the node replicas do not respond to this request, the node in question will tell CM, and CM will return a new node to be used as replica.**
- `get(key)`: returns all values associated with the key. In this project it will return a list of items in a cart.
- `finalize()`: removes all pointers and data allocation associated with the client in question. It also sends messages to all nodes that have the replica so they can also erase all data associated with the client in question.
- `get_rep(port)`: returns full replica to the requesting node.
- `Send_replica(port)`: sends full replica to a node (or CM).
- `join()`: sends join request to CM in order to join the system.

2.3 Client

Clients are the ones requesting services from the system. Upon initialization they request a node to talk to from the CM, and from that point on the client will interact with that particular node until it finishes its business, or if the client decides to change key, or if the node fails. For that, the client main functionalities are:

- `init(&env)`: request login from CM and set all metadata for the connection with a node.
- `put(key , value)`: sends key value to node.
- `get(key)`: get list of items from node (remember, this is a cart simulation).
- `finalize(&env)`: sends a finalize request to node, and deletes all pointers and data allocation associated with the event.
- `report_node_failure(port)`: tells CM that a node is down, and request a new node to communicate with.

3. Distributed Hash Map

The data structure used for distributing the key-values pairs among nodes is a hash map that maps the keys to the address space. All char in a key are first converted to their corresponding ASCII value, and added up, then passed to the hash function as argument. Once a node is returned, the CM checks to see if the node is up, before blindly returning the value. If the node is down, the CM will reroute the request to a node that has the replica of the key-value stored in it.

The mapping is dynamic which allows the system to accommodate unexpected node failure. For that reason, the CM not only have a routing map, it also has a map of all replicas, in case nodes fail.

6.3 Cons of the Hash Map design

Even though the number of nodes in the system, at the start, is dynamic, once the system is running the number of nodes cannot easily be increased. It will require CM to remap all routes and replicas, and have nodes send their primary data to the new key owner. This functionality, however, was not implemented.

4. Recovering from failure

There are three ways for the system to know that a node is down: 1) the client loses communication with the node , 2) the node does not get a response from its replica node, 3) the node does not respond to a CM request.

When a node failure is detected the system goes through 3 steps to recover from it:

1. Node tells CM about a node failure.
2. CM updates route map removing any path that leads to the failed node.
3. CM returns a new replica node to the node that reported the failure.

And once the failed node is up again:

1. It sends a join request to CM.
2. CM updates the route map and add a route to the node.
3. CM sends back a replica node that will become the node's main storage list
4. Node request list from the port on item 3.

When a client detects a node failure:

1. It sends the port corresponding to the node to the CM.
2. CM updates the router map and the replica map.
3. Sends back a valid node port to client
4. Client re-inits and establishes connection with new node.

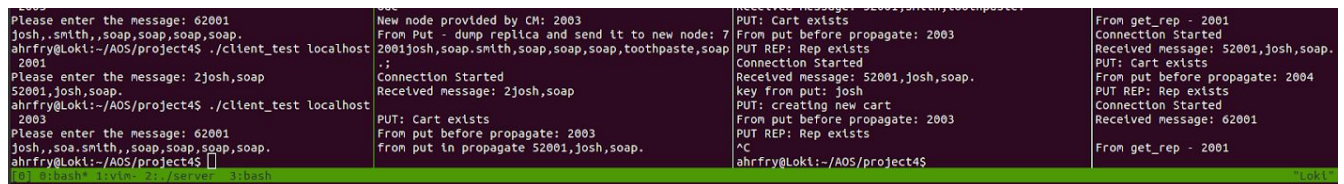


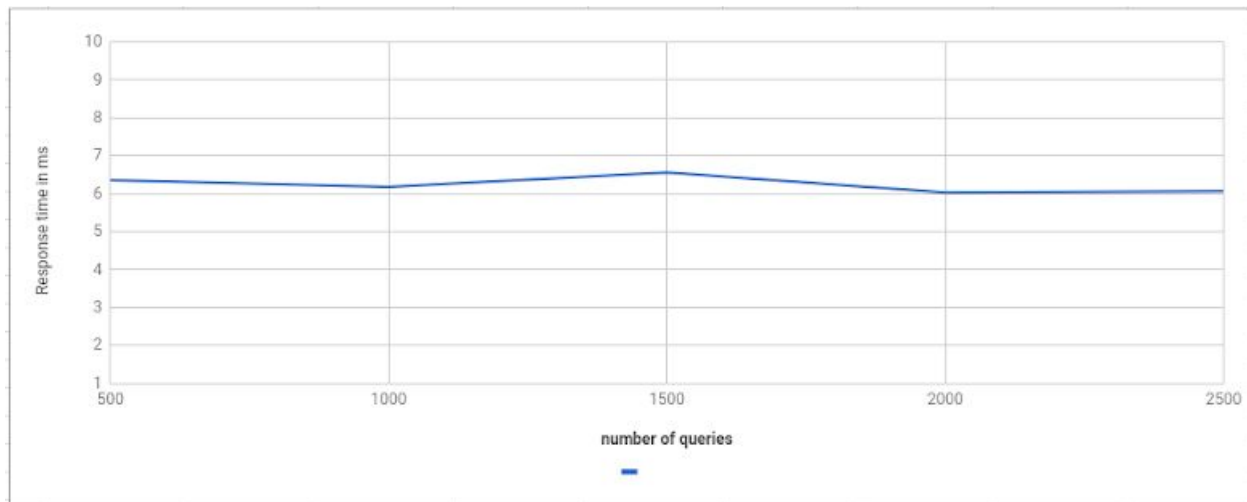
Image 1. Client, node1, node2 and node3 in action. Node2, on third block, has node1's replica, but it dies mid process. On block 4, node3 comes to action and downloads node1's data and becomes node1's main replica.

5. Design tradeoffs and results

Graph 1 below shows that the system scales really well even when the number of queries increase 5 folds. It is worth mentioning, however, that a good amount of tuning had to be made in order to arrive at this level of scalability. The CM, for example, was a major source of latency increase, provided you made all requests go through it. The tests worth mentioning are:

1. Make all put-get requests go through CM: When the number of queries doubled, the response time increased by 1.8. (not the approach taken)
2. Make put-get request bypass CM, but use finalize(&env) every time: this had the same effect as the previous test. Because client had to re-login every time it made a request. (not the approach taken)
3. Make client talk to CM only upon login: This is the most scalable option. CM adds a little latency only on system init, then, as can be noted on Graph 1, the

system scales up even when the number of queries increase. (this is the approach taken)



Graph 1. The graph illustrates how the system scales well even when the number of queries increase considerably. Notice that 1500 clients do `finalize(&env)`, which requires them to restart session through CM, which adds latency to the response time.

One of the best design choice in this implementation is the fact that nodes are able to take some decision on their own in case CM is busy. The nodes have the hashmap function in their library, which allows them to guess where to send put requests to in case of node failure. This is not always accurate because only CM has the updated version of the route map, but for most cases the new route would simply be the returned node from the hash function + 1. For example, if node2 is down, and CM is busy, node1 will warn CM about node2's failure, but will try to guess where CM would have sent the replica to, which would be $\text{node2} + 1 \rightarrow \text{node3}$.

6. Replica versioning

Put requests are followed by a `list.push(time_stamp)` to preserve versioning. When a replica is being downloaded from multiple nodes the node handling the download checks the end of the list for timestamps and keep the most recent list, discarding the older one. Provided there was enough time, the requesting node should, then, send the last version back to the node that does not have the updated version. But this was not implemented.

7. HeartBeat

Heartbeat has been implemented, but substituted for a see-tell method. In this method, a failure is reported to CM as soon as it is detected by either an adjacent node or a client. If the node tries to communicate with a failed adjacent node it will instantly tell CM, and CM will remap the address space, and warn only nodes and clients related to such failure. This prevents CM from unnecessarily bothering nodes that are busy serving clients. On the other hand, if a client sees a failure to communicate with a node, it tells CM, CM will remap the address and return a new node port to the client. The implications of this decision has its pros and cons:

- Pro: better suited for system that does not fail as often. In this case, there is no need to query nodes for heartbeat, since, once a failure is detected, CM can take care of it immediately, without bothering other nodes.
- Cons: If the system has a lot of failures the message exchange becomes exponentially high since nodes and clients will be reporting the same issue to CM.