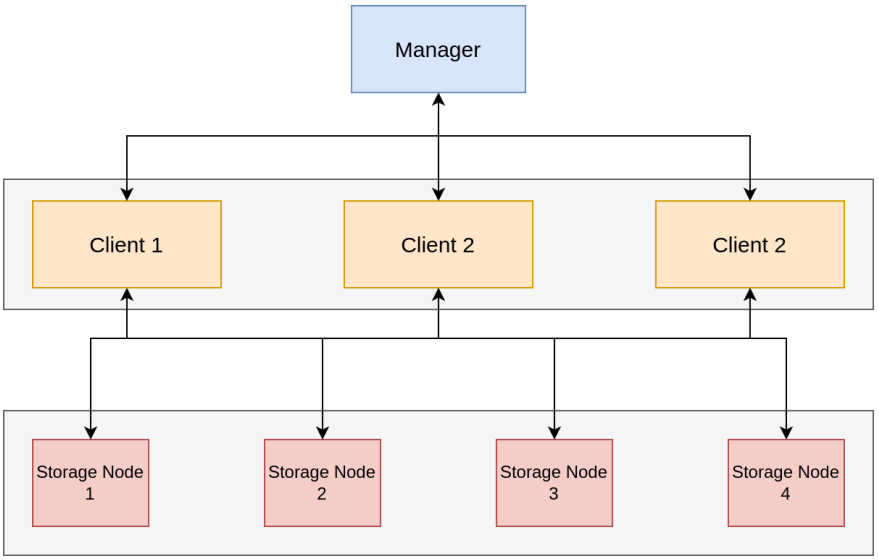
AOS Project 4 - GTStore

Rayyan Shahid
903945349

Overview

GTStore is a distributed key-value store that has three major components

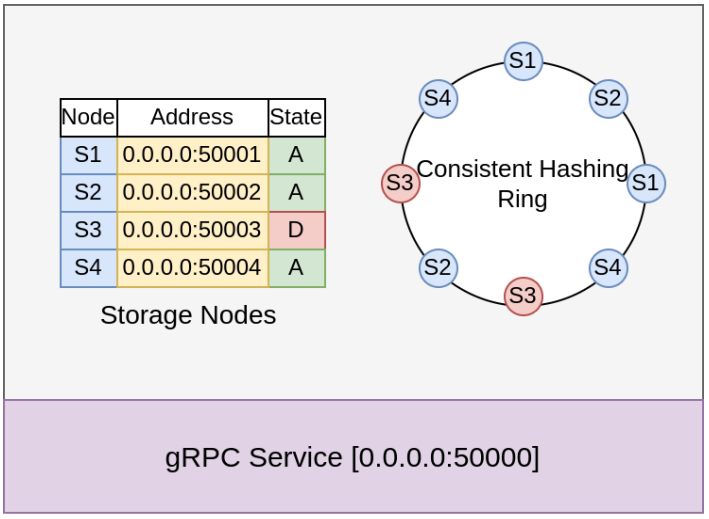


1. Centralized Manager
2. Storage Nodes
3. Clients

GTStores supports data partitioning, provides data replication functionality and provides data consistency guarantees.

System Components

1. Manager

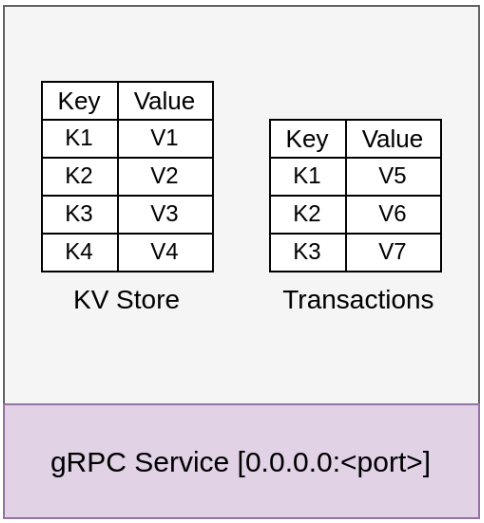


Manager

GTStore manager has three major components

1. Storage Nodes Metadata - This is a table that stores the addresses of the storage nodes as well as the gRPC connection status for the storage nodes
2. Consistent Hashing Ring - This is a ring structure that stores the hashes of the virtual storage nodes that is used for the consistent hashing implementation for routing the get and put requests from the clients
3. gRPC service - Running gRPC service that is used by the clients as well as the storage nodes during initialization

2. Storage Nodes

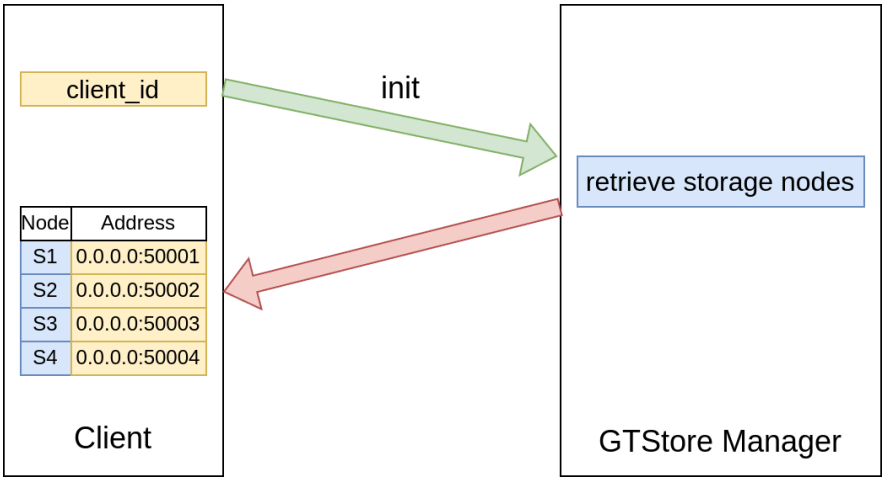


Storage Node

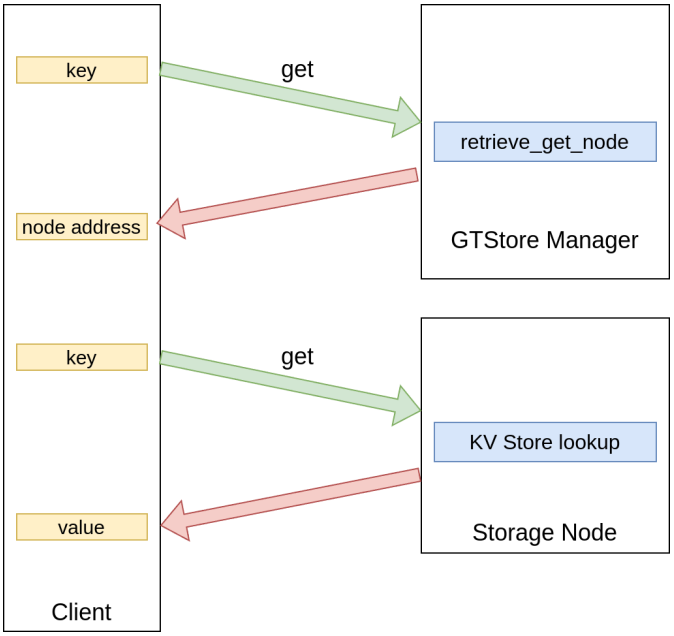
Storage Nodes have three major components

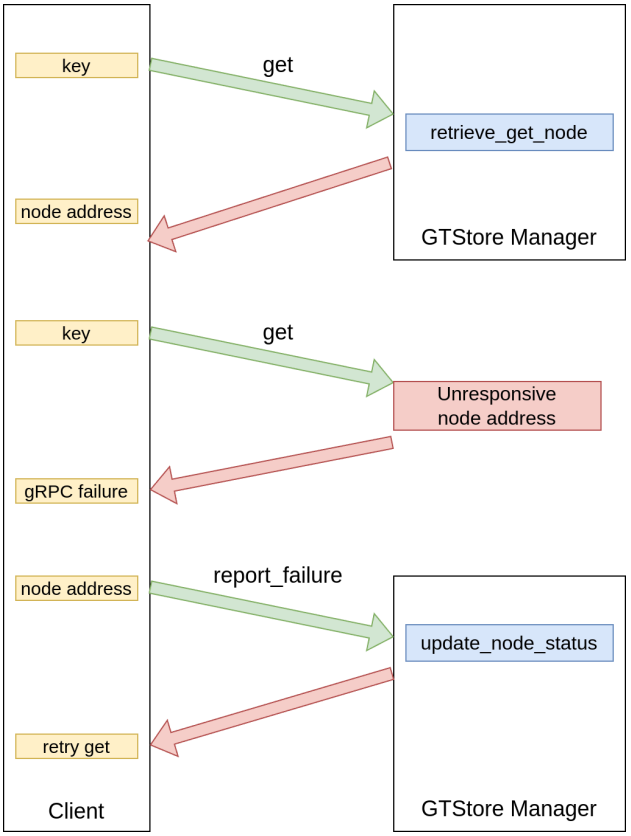
1. KV Store - Stores the key-value pairs for the responsible key partition
2. Transactions - Stores the ongoing un-committed and un-aborted **put** transactions. Synchronization primitives (mutexes and condition variable) allow for concurrent transactions.
3. gRPC service - Running gRPC service that is used by the clients for get/put calls.

3. **Client API calls**
    a. **Init:** Retrieves the information about all the storage nodes and their addresses from the GTStore manager



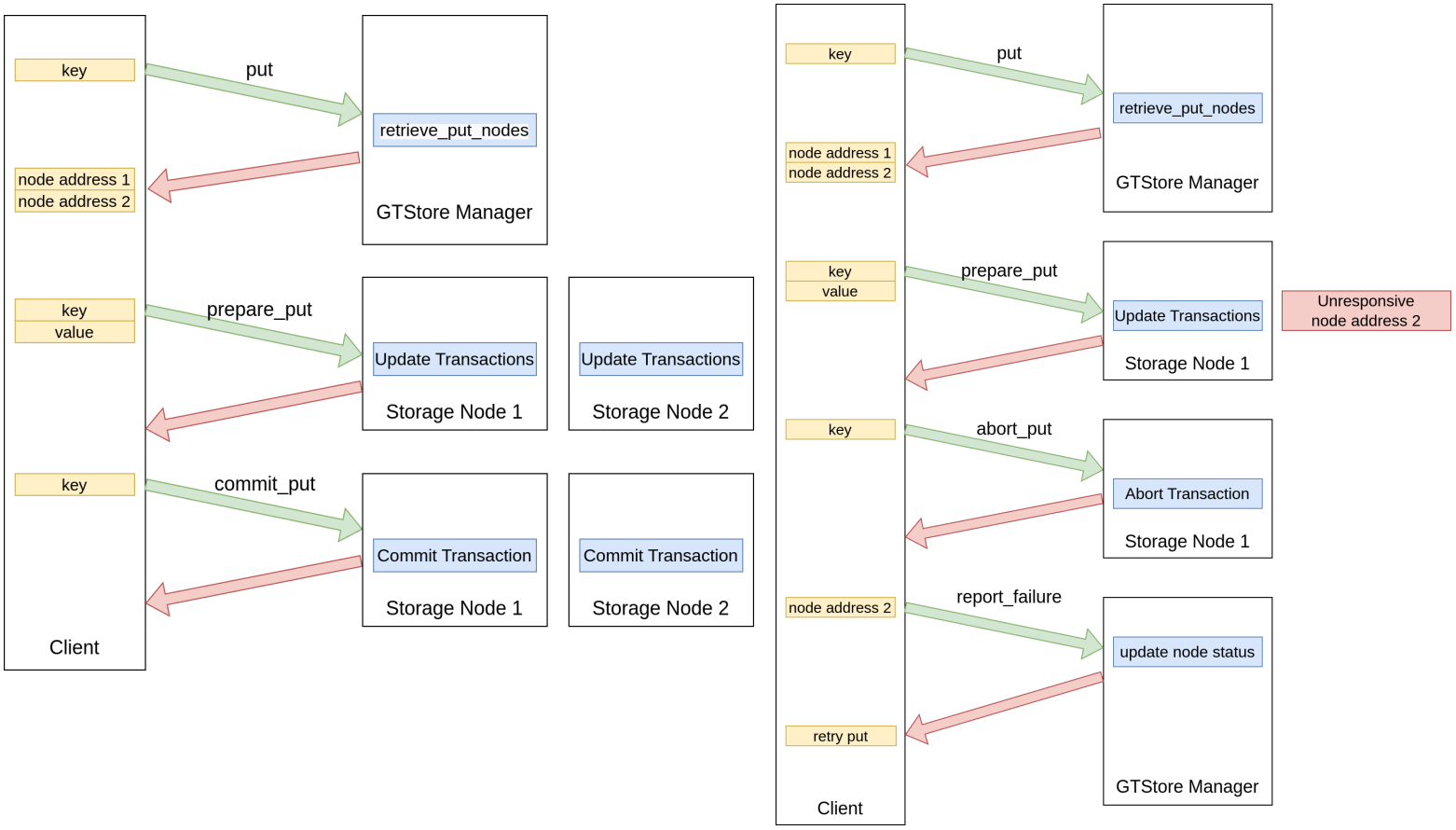
    b. **get**





i. Retrieves the storage node address from the GTStore manager for the queried key
ii. Retrieves the value from the storage node for the queried key
iii. In case of unresponsive storage node, the client reports the failure to the manager and retries the get request.

c. put

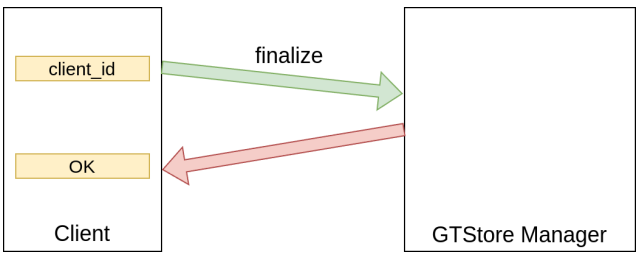


i. Retrieves the storage node addresses from the GTStore manager corresponding to the replicas that would store the queried key.

ii. Uses 2-phase commit for updating the key-value on the storage nodes.
   - Sends prepare_put request to the storage nodes
   - Sends commit_put request to the storage nodes

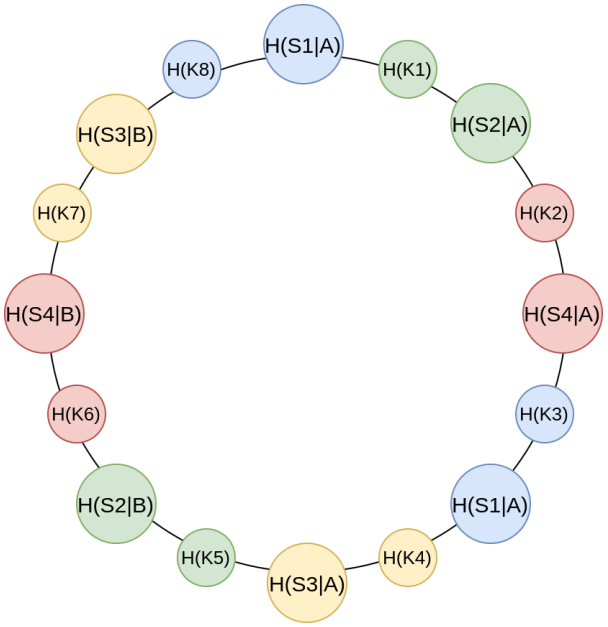
iii. In case of unresponsive storage nodes, the client reports the failure to the manager, sends abort_put to the responsive storage nodes and retries the put request.

d. Finalize: Waits for the GTStore manager's response. This is effectively a null op.



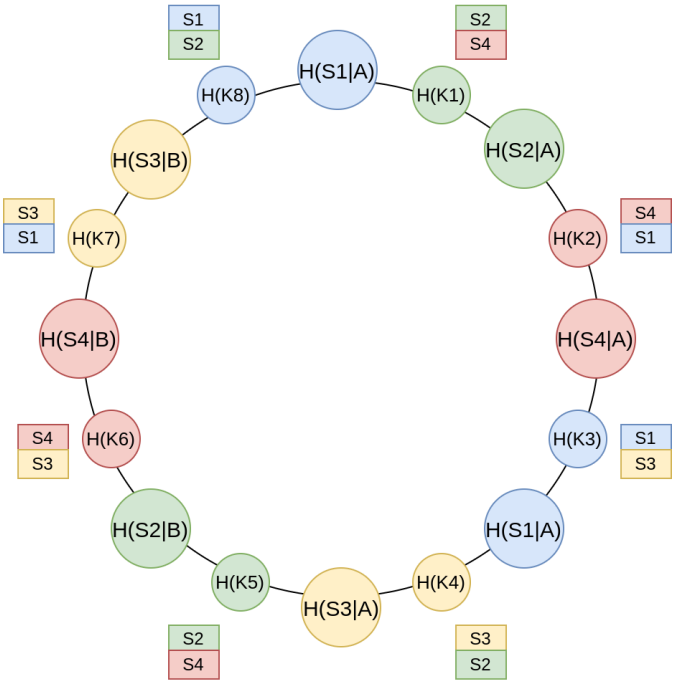
Design Principles

1. Data Partitioning



a. GTStore manager uses consistent hashing with virtual nodes as the partitioning scheme
b. We use the *std::hash<std::string>* function for hashing as it provides a good distribution without being too expensive, and is also very easy to include.
c. Each virtual node [storage_node || virtual_replica_id] is hashed and mapped on to the hash ring.
d. The storage node corresponding to the immediately following virtual node of the hashed key identifies the target partition of the key.

2. Data Replication



S1		K2	K3			K7	K8
S2	K1			K4	K5		K8
S3			K3	K4		K6	K7
S4	K1	K2			K5	K6	

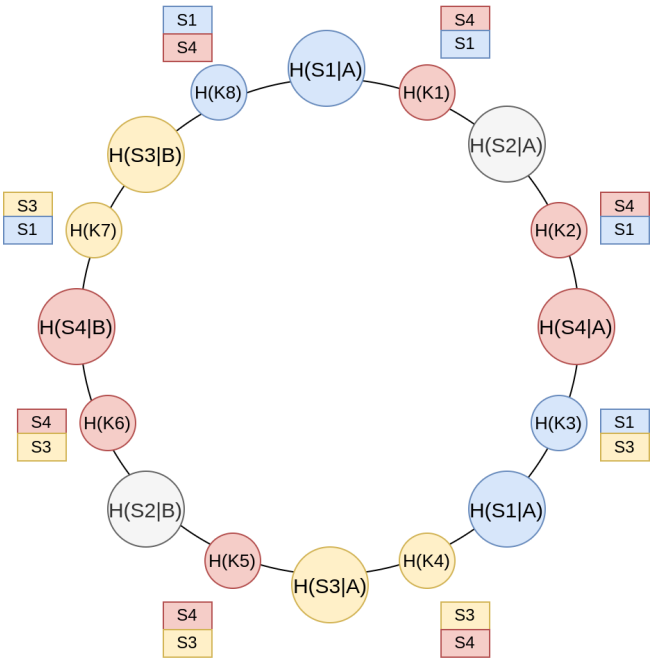
Replication Factor (R) = 2

a. GTStore uses an extended consistent hashing to support data replication
b. The R distinct storage nodes corresponding to the immediately following virtual nodes of the hashed key identifies the replica nodes of the key.

3. Data Consistency

a. As described in the client API calls section, we use the **2-phase commit protocol** to provide **strong consistency** through immediate replication and distributed atomic transactions for the put requests by the client.
b. The transaction is divided into the phases - prepare_put and commit_put. In case of failures, we abort the transaction using abort_put.

4. Node Failures



S1	K1	K2	K3			K7	K8
S3			K3	K4	K5	K6	K7	
S4	K1	K2		K4	K5	K6		K8

Re-partitioning on failure of S2

a. The GTStore is resilient to node failures because of data replication.
b. In case of node failures, the keys are re-partitioned to ensure uniform distribution of keys across the storage nodes.
c. Only a subset of keys need to be re-partitioned because of the use of consistent hashing.

Test Scenarios

1. Basic single server GET/PUT

```
Running Single Server Test...
Server listening on 0.0.0.0:50000
Storage node initialized on 0.0.0.0:50001
Test 1: Basic Single Server GET/PUT
<PUT> key1, value1 , to 0.0.0.0:50001,
<GET> key1, value1 , from 0.0.0.0:50001
<PUT> key1, value2 , to 0.0.0.0:50001,
<PUT> key2, value3 , to 0.0.0.0:50001,
<PUT> key3, value4 , to 0.0.0.0:50001,
<GET> key1, value2 , from 0.0.0.0:50001
<GET> key2, value3 , from 0.0.0.0:50001
<GET> key3, value4 , from 0.0.0.0:50001
Cleaning up processes
```

The above test demonstrates that GET/PUTs with a single storage node works correctly. The client is able to obtain the last modified value for the keys.

2. **Basic multi-server GET/PUT**

```
Running Multi-Server Test...
Server listening on 0.0.0.0:50000
Storage node initialized on 0.0.0.0:50001
Storage node initialized on 0.0.0.0:50003
Storage node initialized on 0.0.0.0:50005
Storage node initialized on 0.0.0.0:50004
Storage node initialized on 0.0.0.0:50002
Test 2: Multi-Server GET/PUT
<PUT> key1, value1 , to 0.0.0.0:50002, 0.0.0.0:50003, 0.0.0.0:50005,
<GET> key1, value1 , from 0.0.0.0:50005
<PUT> key1, value2 , to 0.0.0.0:50002, 0.0.0.0:50003, 0.0.0.0:50005,
<PUT> key3, value3 , to 0.0.0.0:50001, 0.0.0.0:50003, 0.0.0.0:50005,
<PUT> key5, value4 , to 0.0.0.0:50001, 0.0.0.0:50003, 0.0.0.0:50004,
<GET> key1, value2 , from 0.0.0.0:50005
<GET> key3, value3 , from 0.0.0.0:50003
<GET> key5, value4 , from 0.0.0.0:50001
Cleaning up processes
```

The above test demonstrates data replication [R = 3] of keys on storage nodes. It also demonstrates partitioning of keys across the storage nodes.

3. **Availability through a single Node Failure**

```
Running Single Node Failure Test...
Server listening on 0.0.0.0:50000
Storage node initialized on 0.0.0.0:50003
Storage node initialized on 0.0.0.0:50002
Storage node initialized on 0.0.0.0:50001
Test 3: Single Node Failure Test
<PUT> key1, value1 , to 0.0.0.0:50002, 0.0.0.0:50003,
<PUT> key2, value2 , to 0.0.0.0:50002, 0.0.0.0:50003,
<PUT> key3, value3 , to 0.0.0.0:50001, 0.0.0.0:50003,
<PUT> key4, value4 , to 0.0.0.0:50001, 0.0.0.0:50002,
<PUT> key5, value5 , to 0.0.0.0:50001, 0.0.0.0:50003,
<PUT> key6, value6 , to 0.0.0.0:50001, 0.0.0.0:50003,
<PUT> key7, value7 , to 0.0.0.0:50001, 0.0.0.0:50002,
<PUT> key8, value8 , to 0.0.0.0:50002, 0.0.0.0:50003,
<PUT> key9, value9 , to 0.0.0.0:50002, 0.0.0.0:50003,
<PUT> key10, value10 , to 0.0.0.0:50001, 0.0.0.0:50003,

Overwrite data...
<PUT> key1, value11 , to 0.0.0.0:50002, 0.0.0.0:50003,

Data before node failure:
<GET> key1, value11 , from 0.0.0.0:50003
<GET> key2, value2 , from 0.0.0.0:50002
<GET> key3, value3 , from 0.0.0.0:50003
<GET> key4, value4 , from 0.0.0.0:50001
<GET> key5, value5 , from 0.0.0.0:50001
<GET> key6, value6 , from 0.0.0.0:50003
<GET> key7, value7 , from 0.0.0.0:50002
<GET> key8, value8 , from 0.0.0.0:50003
<GET> key9, value9 , from 0.0.0.0:50003
<GET> key10, value10 , from 0.0.0.0:50003
```

```
Killing storage node 3...

Data after node failure:
<GET> key1, value11 , from 0.0.0.0:50002
<GET> key2, value2 , from 0.0.0.0:50002
<GET> key3, value3 , from 0.0.0.0:50001
<GET> key4, value4 , from 0.0.0.0:50001
<GET> key5, value5 , from 0.0.0.0:50001
<GET> key6, value6 , from 0.0.0.0:50001
<GET> key7, value7 , from 0.0.0.0:50002
<GET> key8, value8 , from 0.0.0.0:50002
<GET> key9, value9 , from 0.0.0.0:50002
<GET> key10, value10 , from 0.0.0.0:50001

New operations after node failure:
<PUT> key1, value12 , to 0.0.0.0:50001, 0.0.0.0:50002,
<GET> key1, value12 , from 0.0.0.0:50002
Cleaning up processes
```

This test demonstrates that the system is tolerant to node failures because of data replication. It also demonstrates the repartitioning of the keys after node failure.

After storage node 3 fails, the GET requests for keys 1, 3, 6, 8, 9 and 10 are rerouted to the storage nodes 1 and 2.

4. **Availability through multiple Node Failures**

For brevity, this test is not included in the report. But, this can be executed from *tests/multi_node_failure_test.sh* as described in the README.

Design Tradeoffs Discussion

1. Consistency vs. Availability
   - GTStore prioritizes strong consistency through the use of the 2-phase commit (2PC) protocol for put operations. While this ensures atomic updates and immediate replication, it can impact availability in scenarios where some storage nodes are unresponsive.

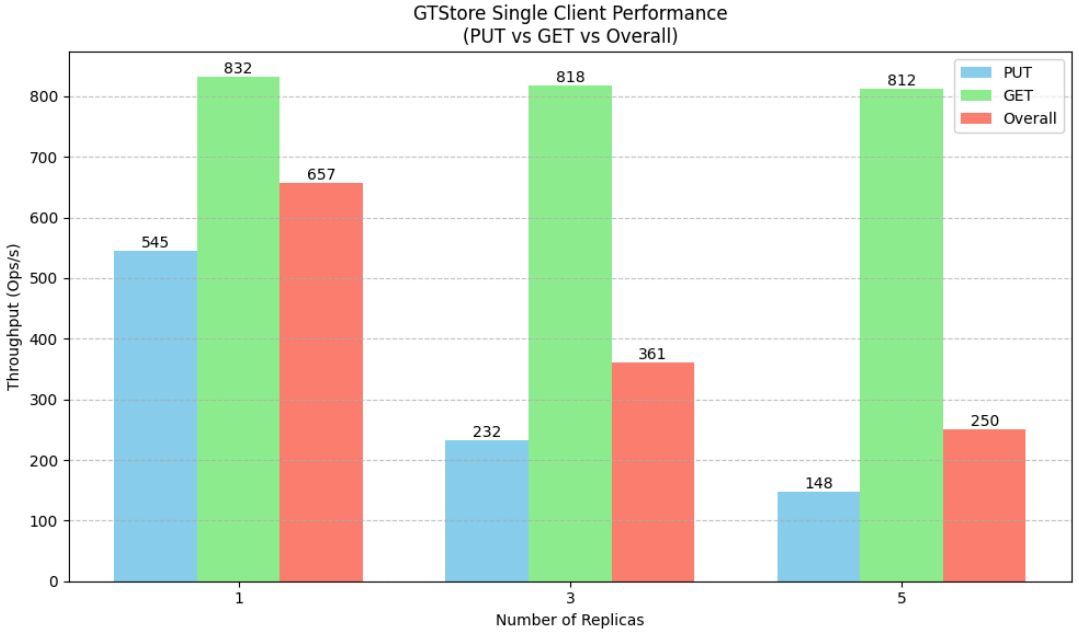
2. Partitioning Complexity vs. Performance
   - GTStore uses consistent hashing with virtual nodes to partition data across storage nodes. This minimizes data movement during node joins or failures but adds complexity to key lookups and hash ring management.

3. Fault Tolerance vs. Resource Utilization
   - Data replication across R distinct storage nodes ensures fault tolerance but increases storage overhead and network bandwidth usage for replication.

4. Simplicity vs. Scalability
   - A centralized GTStore manager simplifies the coordination of metadata, consistent hashing, and failure detection but may become a bottleneck as the number of storage nodes and clients increases.

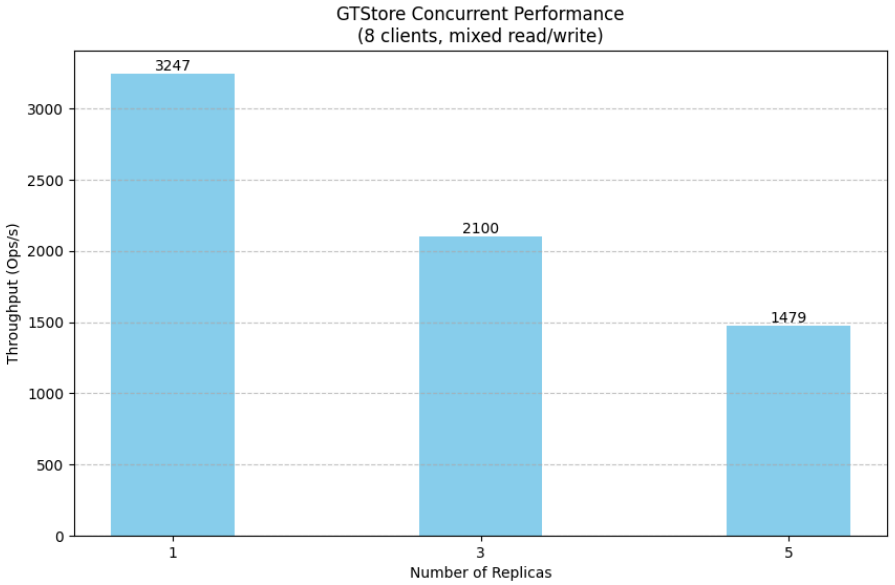
Benchmarks and Results

1. Single Client Throughput



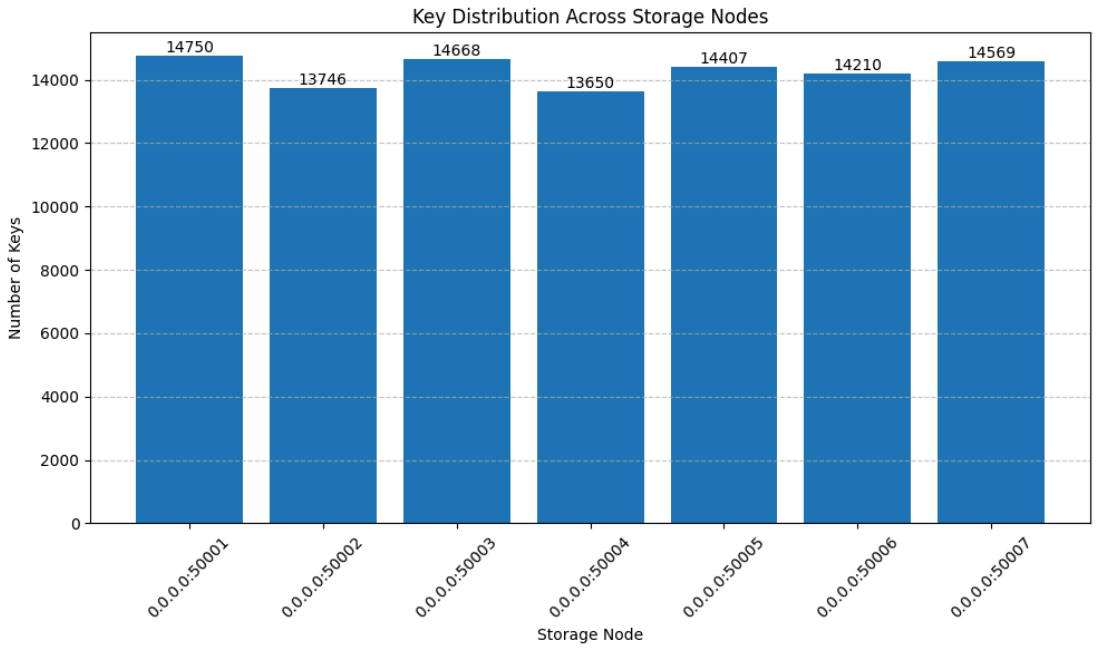
a. A single client makes 200k mixed get/put requests and the throughput is reported with varying number of replicas
b. Throughput of get requests remains the same with an increasing number of replicas. This is because the cost of retrieving the storage node address and retrieving the value from the storage node remains the same irrespective of number of replicas
c. Throughput of put requests goes down with an increasing number of replicas. This is because we are using the 2-phase commit protocol for immediate replication. As a result, the cost of put operation is proportional to the number of replicas.
d. The total throughput also goes down with an increasing number of replicas because of the reduced throughput of put.

2. Concurrent Clients Throughput



a. 8 client threads concurrently make a combined 200k mixed get/put requests and the total throughput is reported with varying number of replicas
b. Throughput goes down with increasing number of replicas because of lower put throughput.
c. However, the total throughput is much higher than the single client scenario which shows that the total system capacity is higher and can be exploited with concurrent client calls to the system.

3. Load Balancing



a. A client thread makes 100k put requests across 7 storage nodes with a replication factor of 1.
b. We observe that the keys are partitioned fairly uniformly across the storage nodes which shows that the choice of the hashing function as well as the virtual replicas are suitable for our system.