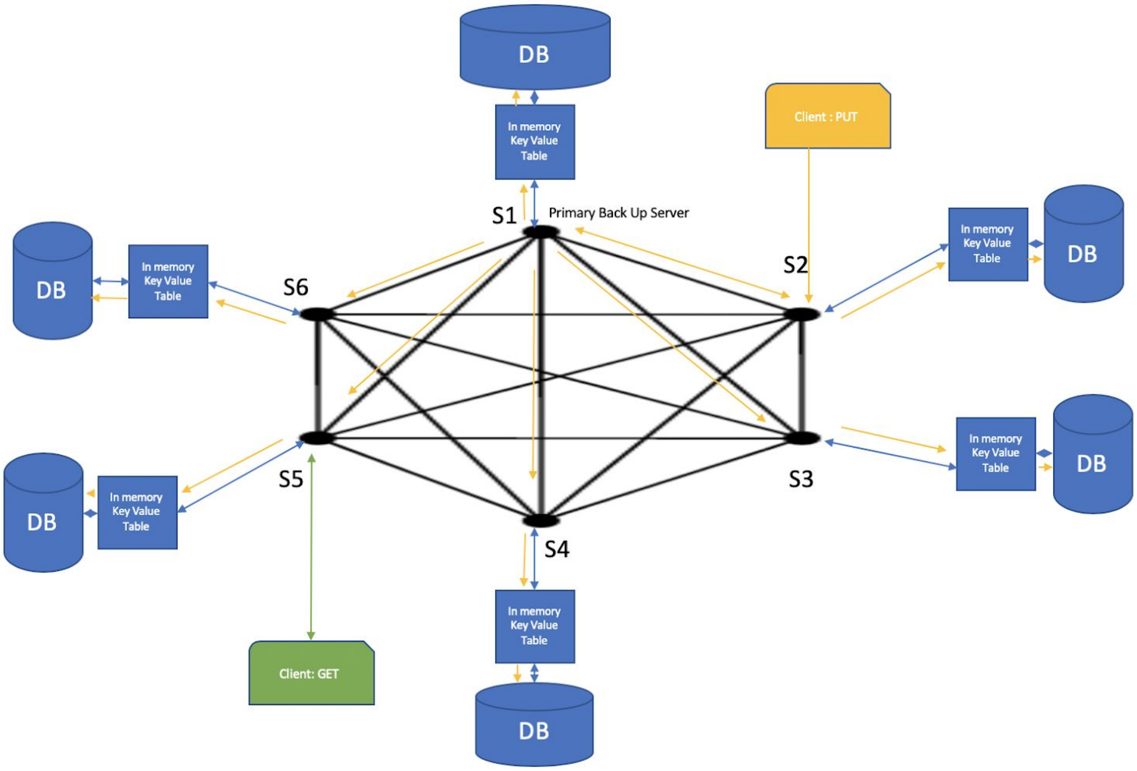
CS739 Project1

Team members: Sri Harshal Parimi, Sangeetha Sampathkumar
Partner team members: Shebin Roy Yesudhas, Sandhya Kannan
Server Github link: https://github.com/harshal95/grpcDemo
Client Github link: https://github.com/harshal95/grpcClient

Architecture Diagram:



System Architecture:

We have implemented a fully-replicated distributed key-value store with a primary backup server for handling updates. gRPC was our primary technology for communication among processes and the language-agnostic protocol buffers were used to send and receive messages between the clients and the servers which were programmed in C and Java respectively.

The server is started as a set of processes from a static configuration. Each server process is assigned a rank(identifier) to distinguish itself from other server processes. On startup, a server communicates with every other server in the system to get the primary backup information. After receiving responses, each server chooses the primary backup with the largest timestamp value and persists this information locally. The timestamp value indicates the time at which a particular server was elected as the primary backup. When the servers start up for the first time, each server picks a default value of rank(0) and timestamp(1) from their database and pass on this primary backup information. This ensures that all of them agree on choosing the server with rank 0 as the primary backup during bootup.

Each server has an equal probability of being picked up by the client for communicating every get/put operation. This ensures that the client requests are distributed among the set of servers. A server responds to a get operation by retrieving the key-value information from its in-memory hash-table that is always in sync with its local database where the key-value store is persisted. During a put operation, the server that receives the put request, forwards it to the primary backup. The primary backup node is responsible for making changes to its local data-store and broadcasting that information to all other servers. After this, the server that was initially called, assumes control and returns to the client with appropriate information. The key-value store changes are persisted individually in the database of the respective servers that receive the broadcast.

During failure of a normal server, the primary backup is the first one to realize this while broadcasting updates. It adds the rank of the failed server to a dead-set it maintains. Every server in the system has an in-memory deadset that keeps track of the failed servers in the system. This ensures that the primary backup doesn't communicate with the failed server until it comes back and notifies everyone in the system. This dead-set is propagated by the primary backup while broadcasting updates to other servers and thus, every server knows about failed nodes eventually. However, when a primary backup node fails, the server that routes an update to the primary is the first one to realize the failure. Since we start up ***n*** servers initially and assign a rank to each process, the routing server does a simple updation of **(primary_back_up_rank + 1) % n** and forwards the same request to the new primary along with an updated deadset containing the failed primary node. The receiving server updates the primary node information along with the sender's timestamp locally and broadcasts this information to other servers along with the update message. From hereon, all put operations are routed to the newly elected primary backup node.

When a failed node comes back up, it starts its workflow by contacting other servers in the pool with its local primary backup information. The other servers respond with the primary backup information they have locally. If more than one node is revived into the system, we don't want them to be influenced by the stale backup information they exchange with each other. Hence, the reviving nodes consider the response that contains the **primary backup with the largest timestamp**. If the primary backup had changed in the time between its failure and revival, which will definitely be the case when a server which was earlier assigned as primary goes down, the primary backup info will be updated correctly in all reviving servers to reflect the current state of the system. Now, the reviving nodes contact the primary to get the key-value store records which were added/updated since it last went down. This ensures that the reviving nodes have data that is consistent with other servers. When all the servers fail, we start the revival process with the server that was last elected as the primary backup because it contains the most current state of the system.

Consistency guarantees:
In the absence of failures, our system enforces strict consistency guarantees. When a put operation occurs at a server, it is propagated to all the replicas before returning the result to the client. As a result, any subsequent operation on the same key would return results consistent with the previous operation to the client, regardless of the server it is in contact with.

If a failure happens at a normal node, there is no change to the consistency guarantee as we just lost one replica for handling reads but writes can go only through the primary backup. Even in the scenario where a primary backup fails, the server routing the put request to it will timeout and try connecting with the next available server in the pool. If there are multiple failures(in rank-order), this process cascades until the routing server finds an available server to send the update and return the result to the client. This mechanism upholds strict consistency at all times.

Client-Server Protocol:

Initially the client creates stub connections with the subset of servers provided to it.
During get/put operation, the client randomly picks a server from the initialized subset and places a gRPC call to it. If the request fails, we mark that server as contacted and retry the operation for remaining servers. The client returns -1 only if all servers are down in its subset. This approach makes the system available even during individual failures but with a tradeoff for latency involved in repeating the request.

If a server routing a put request goes down *after* the update has propagated to all other servers, the client will timeout with a failure. Since it retries the operation, there is a risk of the operation executing twice(as other nodes have persisted that change). The main idea behind the retry mechanism is that we wanted to provide service even if there was at-least one instance of the servers running. In this particular case, there was no way to distinguish if an operation failed as a whole or the client just received a -1, regardless of the change persisting in other servers. A possible alternative would have been to handle updates locally and return to the client immediately. Propagation of updates could have been decoupled from the put operation but a client would be falsely lead to believe that the operation completed successfully, if the contacted server went down while broadcasting updates. This would have caused loss of data as the system would fail to remember that particular operation. We decided to make peace with the earlier tradeoff in our design.

Testing procedure:
We wrote a script for generating the input file and this input file is passed to our test program. The script accepts a list of keys and values pre-defined in a file and would output a test-file in the following format:

```
init,9090,10                    #init operation on 10 servers starting from the port 9090
get,Nike
put,Nike,Shoe
```

….

The test program parses each line from this file and calls the shared library to invoke calls from the client. The result of every operation is stored in a separate output file. This output file was particularly handy for verification while running correctness tests on the system.

1) Correctness tests:

First, we checked for correctness in the scenario where the system has no faults. We took a smaller version of the earlier generated input file and ran our tests on the system with single as well as multiple clients. In the case of a single client, after running the test, we verified the test_output file with our expected output. For the case of multiple clients, we used the same input_file for all the clients and made the clients write their outputs to individual output files. Obviously, we didn't expect the output files to be the same for all the clients because, the response received by the individual client would have depended on the order in which the requests were serviced among the clients. However, we checked the individual database files for each server, to ensure that they held the same number of records. The scenario where multiple clients request the server with non-overlapping keys was trivial and we just had to verify the output files and the database files(to ensure that updates from all clients were propagated to all servers). Additionally, we simulated tests where we performed put on one subset of servers and get from the other subset.

2) Performance tests:

To gather metrics on performance, we ran our system against *varying* distributions of data(uniform key distribution with varying operations, only gets, only puts, hot-key distribution with varying operations), server capacity, number of clients and number of requests. We gathered the throughput and latency for every test-suite configuration possible from the above considered criteria. We wrote a python script to generate varying distributions of data and rpc operations that could be fed to the test program.

Test Results:

1) Correctness results:

i) General replication and consistency:

We conducted a simple test for 10 servers where we send a set of put operations to one subset of servers(from a client) and keep performing get operations from the other subset of servers(via another client). If each get operation of a particular key happens later than the corresponding put operation, we should be able to get success response values from the client that performs get. Our system exhibits this correct behavior.

ii) Availability and consistency under fault tolerance:

To check for availability and data consistency in the wake of failures, we follow the killing of each server with running of a sample input file that contains get operations ***on the writes that***

happened prior to the failure, to check for correctness. We also perform additional writes after every failure to check if the updates are propagated and persisted until atleast one server is alive in the system. We follow the same procedure for simulating group failure of servers. We performed this test procedure on a pool of 10 servers and realized consistent results every time.

iii) Data consistency during revival of servers:

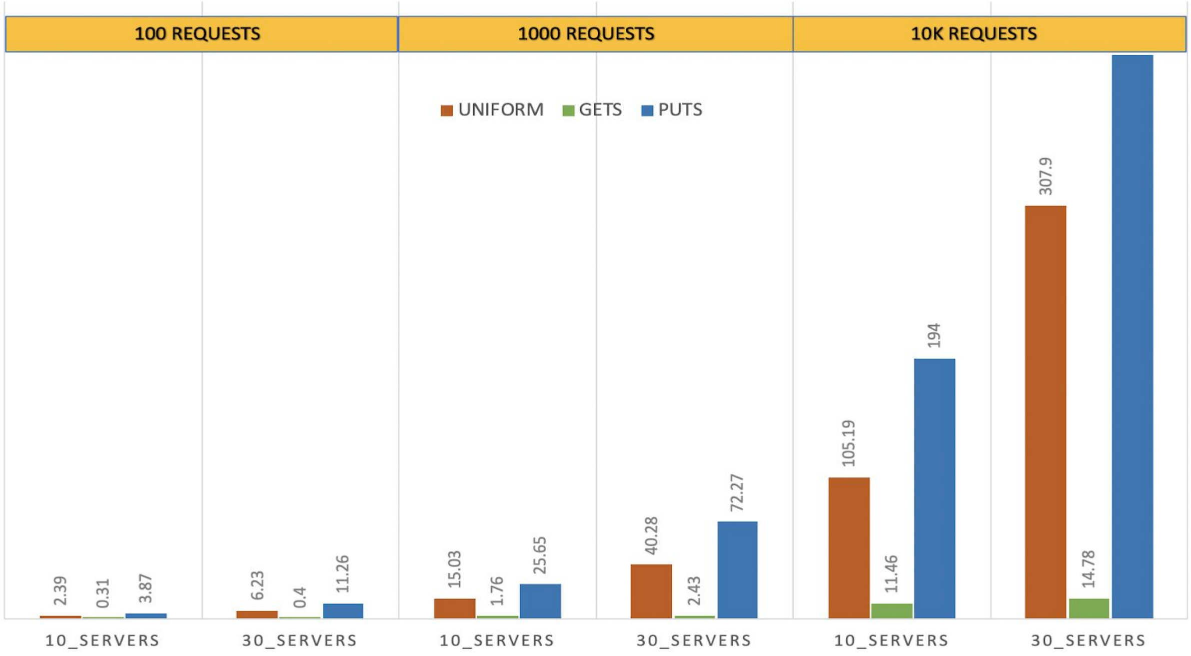
This test was simulated by killing a group of servers in the system and reviving them one by one into the system. The reviving server got the latest primary backup information from other alive servers and did a data sync up from the primary backup. We tested the correctness in this scenario by performing get operations on the data that was created in the time between the said node’s failure and revival. Additionally we ensured that the operations were now directed to the specific revived server from the client(init’s parameter). We do this revival procedure iteratively to see if all servers sync up the latest information properly and are able to service the clients.

We repeat this test procedure for reviving a group of servers simultaneously. Even if there are a majority of incoming revived servers, they must sync up data only from the already alive servers. We checked the db dump of the revived servers to see if they synced up from the latest primary backup. The repeated reads performed on the earlier unseen data by the revived servers will ensure that fresh data has been correctly replicated to the revived servers.

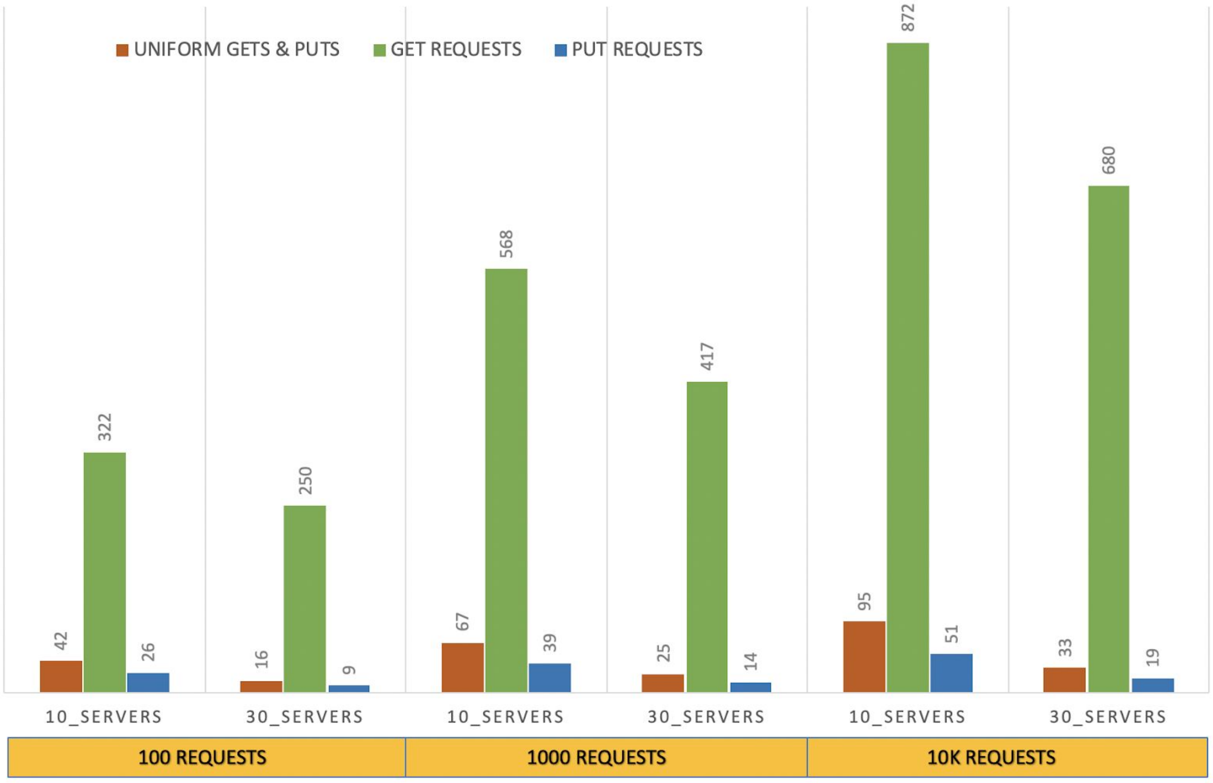
2) Performance results:

Get operations happen the fastest in our system as there is only one communication that happens between the client and the server. The server returns the value from in-memory hashtable. Put operations are bound by the synchronous forwarding of requests to the primary backup followed by a broadcast to other servers in the system. We performed tests on uniform distribution of keys with random get and put operation. Its performance fell in between a system driven purely by get and put requests. We performed tests on hot-key distributions of data but they were not significant to our system as the client picks a server randomly for every request. There is no hash-key based server routing that causes the system to perform bad for hot-key data distributions.

Ideally, when we scale the servers from a configuration of 10 to 30, the latency should definitely decrease, at least for get operations(because our latency of put is bounded by full-replication to all servers). But the performance follows a downward trend suggesting that the single machine where we were running all the processes in parallel got overloaded and was incurring the overhead of process context switches.



COMPARISON OF LATENCY MEASURED IN SECONDS BETWEEN 10 AND 30 SERVERS



COMPARISON OF THROUGHPUT(#REQUESTS PER SECOND)BETWEEN 10 AND 30 SERVERS

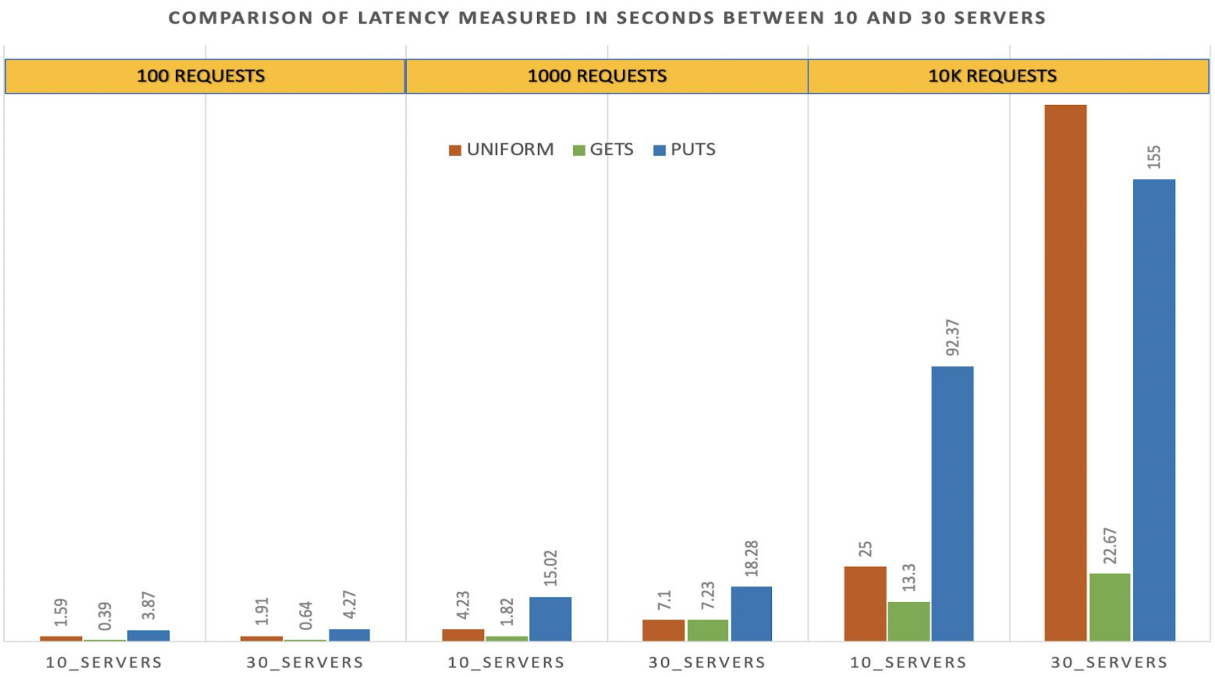
However, we wanted to show that the system would scale well for multiple clients, without running the risk of overloading our test machine. As a result, we conducted an additional test scenario, where we compared the performance of our system servicing a client requesting 1000 requests vs two clients raising 500 requests each. The 10 server-processes parallelized the handling of requests from multiple clients, thus proving the benefits of using a distributed system for a large user-base.

Comparison of Multi-client vs Single Client for a total load of 1000 requests

	Client 1(500 req)	Client 2(500 req)	Single client(1000 req)
Get	1.192s	0.917s	1.76s
Put	17s	17s	25.65s
Uniform Get/Put	11.42s	10.55s	15.03s

Partner results:

The relative trend of gets being faster compared to puts and the system getting slower with multiple processes on a single machine holds true for the partner team as well. However, their put operations were relatively fast. This could be attributed to us using a blocking API for replicating information to all servers in the system. They followed an asynchronous approach to broadcast information among their servers leading to faster replication while performing puts. Their system worked similar to ours against all the correctness tests we simulated. The only discrepancy we noted was that their get operations were slightly slower than our system for all the simulated tests. We have no valid reason for this behavior as we inferred that their get operation also involves a single communication to the server that reads from an in-memory hashmap. The following graph shows the results we obtained by running our tests on their system:



Scope for improvement:

- Our system performs great for get operations. However, the synchronous calls in put operation has proved to be a bottleneck for our system's performance. We could improve our system's performance by making the update and replication strategy at the primary backup-server asynchronous.
- The consistency guarantees augur well for all the scenarios that we anticipated and evaluated in our testing phase. However, we realized that during total failure, the tenets of consistency holds true only if we start revival with the server that was the primary backup at the time of total failure. Pair-wise communications incorporating vector clocks/Bayou's algorithm could be used to exchange unseen data between servers during revival.