SS-KVSTORE: Simple and Small Key-Value Store

Arjun Balasubramanian   Mohammed Danish Shaikh

University of Wisconsin - Madison

Abstract

This study presents our experiences in designing and implementing a *simple* and *small* key-value store named SS-KVSTORE. We first outline the mechanism for building a key-value store assuming that there are no failures and then outline how failure handling can be incorporated into our mechanisms. We find that we can implement a highly reliable and performant key-value store by using just a small amount of auxiliary metadata associated with each key-value pair. Evaluation on a prototype implementation shows that SS-KVSTORE is able to achieve a read throughput of up to ~1400 keys/sec and a write throughput of up to ~45 keys/sec.

1 Introduction

Key-value stores are a popular class of services used for a variety of purposes such as object caching, storing customer preferences, handling session management, etc. A lot of research has gone into how to make key-value stores more performant [5, 8, 10, 12], resilient to failures [1, 6], and to provide varying consistency guarantees [3, 4].

This paper presents our experience in building a *simple* and *small* key-value store which we call SS-KVSTORE. SS-KVSTORE stores key-value pairs persistently and is built to tolerate failures and provide at least eventual consistency in the presence of failures. Under the covers, SS-KVSTORE consists of a bunch of *shared-nothing* servers that provide replicated service. Additionally, SS-KVSTORE is also capable of handling multiple clients that can concurrently *get* or *put* values to the key-value store.

We first present a basic design (Section 2.3) that works under the assumption that none of servers fail. Then, we examine the behavior of the system under the presence of failures and introduce mechanisms to handle these failures (Section 2.4).

We evaluate our design (Section 4) by building a prototype implementation and running it against two common workloads - uniform workload and a hot and cold workload. Our results show that SS-KVSTORE is able to achieve a read throughput of up to ~1400 keys/sec and a write throughput of up to ~45 keys/sec. We also perform a set of micro-benchmarks that help point out certain improvements that we can make in our prototype to further improve performance. Our prototype implementation is available at https://github.com/Arjunbala/KVStore.

2 Design

This section describes the internal design of SS-KVSTORE. In Section 2.1, we introduce the assumptions that help simplify the design of the system. We highlight the goals for SS-KVSTORE in Section 2.2 which then leads onto the basic design outlined in Section 2.3. The basic design section covers details of the interface exposed to clients of SS-KVSTORE as well as internal specifications of the protocol used to interact between components assuming that there are no failures. In Section 2.4, we extend the basic protocol to handle various types of failures.

2.1 Assumptions

SS-KVSTORE provides a simple key-value storage service where the keys and values are both strings. Our system imposes a bunch of restrictions for simplicity. It does not support the storage of binary objects (BLOBs) like Dynamo [4]. Keys can be printable valid ASCII characters that are at most 128 bytes in length and cannot include the characters "[" or "]". Values have the same restrictions as keys except that they can have a size of upto 2048 bytes.

At a bare minimum, we assume that SS-KVSTORE needs to be runnable as a set of processes on a single machine. However, since we use the socket abstraction for communicating between the server processes (Section 3), we expect our implementation to work seamlessly when processes belonging to the service reside outside the boundaries of a single machine. We implement SS-KVSTORE using a shared-nothing model where each server maintains it's own individual copies of the data. Furthermore, SS-KVSTORE consists of a small number of servers, hence the system replicates all data across

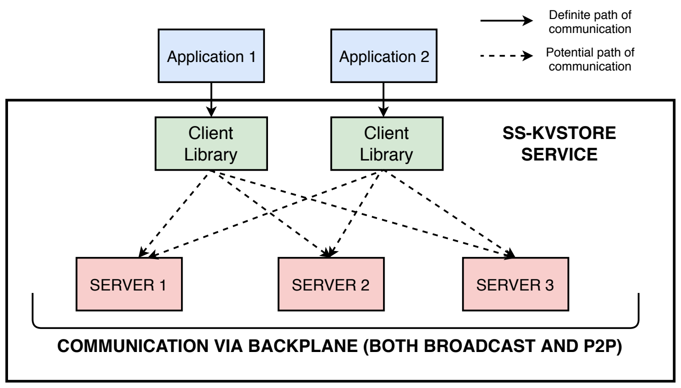

Definite path of communication
Potential path of communication
Application 1
Application 2
Client Library
Client Library
SS-KVSTORE SERVICE
SERVER 1
SERVER 2
SERVER 3
COMMUNICATION VIA BACKPLANE (BOTH BROADCAST AND P2P)

Figure 1: High-level architecture illustrating the various components in SS-KVSTORE

all servers to ensure high availability and fault tolerance.

In Section 2.4 where we deal with making our protocol resilient to failures, we assume that processes can fail independent to each other. We do not handle cases where the OS crashes or the machine fails as a whole.

2.2 Goals

We design SS-KVSTORE with the following goals in mind:

- **High availability and fault tolerance:** We design SS-KVSTORE with the goal to be highly available even in the presence of failures. This translates into 3 sub-goals - (i) Clients should be able to use the service as long as at least one server is up. (ii) Clients should be able to read values from or write values to any server that is up at any given point of time. The service should be responsible for sufficiently replicating writes to ensure the availability of recent writes at other servers. (iii) In anticipation of failures, the service should take care of replicating write operations as quickly and reliably as possible so as to have them available at other servers.

- **Nearly strongly consistent in the absence of failures:** Clients should be able to observe strong consistency in values as much as possible in the absence of failures. The view of consistency should be stronger for a single client that makes a sequence of reads/writes than two concurrent clients issuing a sequence of reads/writes.

2.3 Basic design

Figure 1 outlines the high level structure of the SS-KVSTORE service. Applications interact with the service through APIs exposed by the client library (Section 2.3.1). The client library internally communicates with one/more servers to serve client read/write requests. In our setting, a *server* is an entity that individually manages data in isolation. It may be simply a process or a virtualized entity [7, 9, 11]. For the purposes of our prototype, we assume that server entities are isolated as different processes. Any two servers may or may not reside on the same physical machine. The servers utilize a backplane to communicate with each other for purposes of replication and consensus (Section 2.3.2). In the sections to follow, we show that using broadcasts and peer-to-peer communication in different situations helps SS-KVSTORE provide a balance of reliability and performance.

2.3.1 Client library APIs

In this section, we outline the APIs used by applications to utilize the service and outline the internals of the client library.

int kv739_init(char ** server_list): An application uses this API to initialize it with a subset of servers that it can communicate with, as specified by *server_list*. The API returns 0 if at least one of the servers specified in *server_list* is available at the moment of initialization. If none of the servers specified in *server_list* are available, then the API returns -1. Internally, upon receiving the server list, the client library tries to connect to each of them and check if at least one server is available. The client library then designates one of the servers as a CONNECT SERVER. The purpose of the CONNECT SERVER is to handle *PUT* requests as we will discuss below.

int kv739_shutdown(void): An application uses this API to terminate and cleanup it's connections with the servers specified in *server_list*. Any application that wishes to communicate with a different group of servers must call *kv739_shutdown()* before calling *kv739_init(server_list)* for the new set of servers. The API returns 0 if the cleanup was successful, else it returns -1.

int kv739_get(char *key, char *value): This API is used to retrieve the value for a given key. If the key is present, then the API stores the value in the provided *value* string and returns 0. If the key is not present, the API returns 1 and if there is a failure, the API returns -1. Internally, the client library contacts the list of servers specified in *server_list* and returns the value stored by majority of the servers. This choice is further examined in Section 2.3.2.

int kv739_put(char *key, char *value, char *old_value): This API is used to put a value for the given key. If the key has a value already set, then the API sets this value through the *old_value* string and returns 0. If there is no old value, then the API returns 1 and *old_value* would simply be a null-terminated string. The API returns -1 in case of failures. Internally, the API issues a *PUT request* to the CONNECT SERVER that was designated during the *kv739_init* call. Details about how this write is replicated across servers is discussed in Section 2.3.2.

2.3.2 Basic protocol details

Algorithm 1 describes the basic protocol used for reading and writing values in SS-KVSTORE. We first describe the

Pseudocode 1 SS-KVSTORE Basic Protocol

```
 1: SERVER LIST                                  ▷ List of servers from kv_init API
 2: CONNECT SERVER                     ▷ Primary server designated by client library
 3:
 4: ▷ Client library APIs start here
 5: ▷ Client library GET API
 6: procedure KV739_GET(String Key, String Value)
 7:     VALUES = []
 8:     for all server ∈ SERVER LIST do
 9:         VALUES.append(server.get_value(key))
10:     end for
11:     return majority value in VALUES
12: end procedure
13:
14: ▷ Client library PUT API
15: procedure KV739_PUT(String Key, String Value, String Old_Value)
16:     Old_Value = CONNECT SERVER.put_value(key,value)
17: end procedure
18:
19: ▷ Server internals starts here
20: SERVER DATASTORE            ▷ Datastore for storing key-value pairs for a server
21: SERVER LIST                             ▷ List of servers belonging to service
22: SERVER ID                                         ▷ ID of this particular server
23: procedure GET_VALUE(String Key)
24:     Return value stored for Key in SERVER DATASTORE
25: end procedure
26:
27: procedure PUT_VALUE(String Key, String Value)
28:     PRIMARY_SERVER_FOR_KEY = hash(Key) % SERVER LIST.length()
29:     if PRIMARY_SERVER_FOR_KEY == SERVER ID then
30:         Old_Value = Update Value for Key in SERVER DATASTORE and mark
    entry as dirty
31:         seq = increment sequence number associated with Key
32:         for all server ∈ SERVER LIST do
33:             server.broadcast_write(Key,Value,seq)
34:         end for
35:     else
36:         Old_Value = Update Value for Key in SERVER DATASTORE and mark
    entry as dirty
37:         PRIMARY_SERVER_FOR_KEY.put_value(Key,Value)
38:         return Old_Value
39:     end if
40: end procedure
41:
42: procedure BROADCAST_WRITE(String Key, String Value, Integer seq)
43:     current_seq = Sequence Number for Key in SERVER DATASTORE
44:     if seq >= current_seq then
45:         Update Value for Key with sequence number of seq in SERVER DATAS-
    TORE
46:     else
47:         No need to apply broadcasted write.
48:     end if
49: end procedure
```

functionality offered by the server and then describe how the client library interacts with the server.

Setup. Each server offers two APIs - a *GET* API and a *PUT* API used to read and write values respectively to/from that particular server. Each server consists of a SERVER DATASTORE that persistently holds keys and values. For purposes of versioning the values of keys, we associate a sequence number with each key. The current value of the sequence number is an indication of how many times the particular key has been updated.

Each server is responsible for acting as a *primary server* for a partition of key values. The *primary server* for a key is responsible for ordering all writes associated with that particular key. Each key also has a *dirty* bit associated with it. When updates are applied by a non-primary server, the *dirty* bit is used to indicate that the key's value has been updated locally, but the primary server for that key has not ordered it yet. We discuss the reason for a primary server to mark an entry as dirty in Section 2.4.

Handling reads. For reads, the client library issues reads to all servers specified by the application through the *kv739_init* API. The client library returns the majority value among all values returned to it (Line 6 in Algorithm 1). When a server receives a *GET* request, a server simply reads the value for the key from it's SERVER DATASTORE and returns the value (Line 23 in Algorithm 1). Hence, with respect to reads, the guarantee offered is that the value returned is the value most propagated among the servers.

Handling writes. For writes, the client library issues a *PUT* request to one of the servers (called a CONNECT SERVER) specified by the application through the *kv739_init* API. The client library then simply returns the old value of the key obtained from that server to the application along with the appropriate return code (Line 15 in Algorithm 1). When a server receives a *PUT* request, it computes a hash value for the key associated with the request and identifies the *primary server* for that key, which will be responsible for ordering writes for that key. If the server is itself the *primary server*, then it simply applies the update to its SERVER DATASTORE, updates the sequence number associated with the key and propagates the update to all other servers via a *write broadcast*. The broadcast consists of the key, value, and the updated sequence number. If the server that received the request is not the *primary server*, then it persists the update in its datastore. Here, it does not update the sequence number and simply marks the update as dirty. It then forwards the *PUT* request to the *primary server* (Line 27 in Algorithm 1).

Handling broadcast writes. A server receives a broadcast write when any server has committed an update for which the server was a *primary server*. On receiving a broadcasted write, a server checks the current sequence number of the key associated with the broadcasted update in it's SERVER DATASTORE. If the current sequence number is lower than the sequence number of the broadcasted write, then the write is applied to the datastore. Else, the broadcast is ignored. This protocol helps with cases where the broadcast for the same key might arrive out of order (Line 42 in Algorithm 1).

2.4 Protocol changes for server failure

In this section, we describe how our protocol can be augmented to handle failures of servers. The key assumption in our design is that each server knows the presence of other servers in the system. Thus, when a server becomes unavailable, this information needs to be propagated and updated at all servers in the system. Let us consider a case where a server S goes down. First, we discuss how the system detects that S is down and when(if) it comes back up. Next, we discuss how to manage keys for which S served as a primary.

2.4.1 Detecting server failure

The failure of a server S can be detected by another server S' in the cluster if S' receives a PUT request for a key having S as it's primary server. When this happens, server S' sends a *SERVER_DOWN* broadcast to all servers in the system indicating that server S is down. When server S comes back up, S can send a *SERVER_UP* broadcast to inform all other servers that it is up and running again.

For performance reasons, each server caches the current state of all the other servers in the system. To do so, each server stores a bitmap representing the status of other servers in the cluster. The bitmap is arranged such that it contains the status (UP/DOWN) for all servers ordered by their unique server IDs. The bitmap is updated as follows:

- If a PUT request arrives at a server that is not primary, the server needs to forward the request to the *primary server* as mentioned in Section 2.3. However, if the *primary server* is down, the forwarding will fail. The server that received the request then sets the *primary server*'s bit in it's bitmap to indicate the *DOWN* status, and broadcasts the same to other servers in the cluster so that they can also update their bitmaps.

- When a server comes up, it sends a broadcast message to all the servers in the cluster. On receipt of such a message, each server updates it's bitmap to indicate the *UP* status of the server that sent the message.

2.4.2 Handling failure of primary server

We may have a scenario where a client sends a PUT request to a server that is not primary. In such a case, the server that received the request writes a dirty record locally (Section 2.3) and forwards the PUT request to the *primary server*. *However, what happens if the primary server is down?*

When the *primary server* for a key is down, we need to designate another server as the *primary server* for that key. An important consideration here is that all servers must agree on the same new *primary server*. We use the *bitmap* mechanism described above to help identify a new *primary server* for the key.

Let us say that the status of *primary server* is stored at position i in the bitmap, and there are N servers in the cluster. Then, the server that received the PUT request simply forwards the request to the next alive server as specified in the bitmap, i.e. $(i+k)\%N$ and servers at positions $i, i+1, \ldots k-1$ are all $DOWN$. Since all servers have the same view of the bitmap, each server will therefore identify the server specified in position $(i+k)\%N$ as the new *primary server* for that key.

2.4.3 Handling non-propagated updates

In an eventually consistent system, there can be scenarios where a server might crash before any recent update it got can be replicated to other servers in the cluster. In this section, we consider a bunch of those scenarios and outline how we handle them.

Scenario 1. A *primary server* for a key can crash after it has persisted the update for a key but before the update has been replicated to other servers. We handle such a scenario using the *dirty* field associated with each key. During a local update, we mark the key as *dirty* and wait until we receive back the write broadcast to unmark the key as *dirty*. When a server comes back up, it re-issues the write broadcast for all keys whose entries were marked as *dirty*. This design choice opens up the possibility that a server might receive a stale write or a duplicate write. Hence, we must have a mechanism in servers to tolerate this *at-least-once semantics*. The sequence number is tagged along with the write broadcast so that servers will apply the update only if the sequence number is higher than that in their SERVER DATASTORE.

Scenario 2. A non-primary server can crash between the time it persisted an update and before it could forward the request to the *primary server*. Once again, we can handle such a scenario using the *dirty* field associated with each key. When the server comes back up, it can re-forward all updates marked as *dirty* to the corresponding primary servers. Once again, the notion of sequence numbers helps primary servers avoid making stale updates. However, in such a scenario the *primary server* notifies the server which forwarded the request that the *dirty* write is out of date and provides the server with the updated value.

Scenario 3. Any server can crash between the time it received a write broadcast from a *primary server* to when it persisted the update in the SERVER DATASTORE. We handle such scenarios by having clients poll all servers specified by them in the *kv_init* API during reads so that the client can see the value held by a majority of servers.

Scenario 4. We may have a case where a server crashes while handling a client *PUT* request. In such a case, we expect clients to retry the *PUT* operation. The semantics for such a retry is described in Section 2.4.5.

2.4.4 Handling stale values

When a server S is down, other servers are acting as the *primary server* for keys belonging to the key space for which S was the *primary server*. Hence the server needs to read most recent values of keys belonging to it's key space after it comes up. For doing so, whenever a server comes up, all objects read from it's SERVER DATASTORE are marked as *stale*. Subsequently, if the server receives a request for an object that is marked as stale, the server first reconciles the state of the object by consulting with other servers in the system and then serves the request.

2.4.5 Client failover

Client needs to handle server failure during PUT requests since GET requests are served using a quorum (Section 2.3.2). In case of a PUT request, if the client fails to establish connection with the CONNECT SERVER(Section 2.3.2) or gets a failure return code, it *fails over* to the next server in the list of servers specified in the server list during *kv739_init* API call (Section 2.3.1).

3 Implementation

This section discusses the implementation details of SS-KVSTORE.

3.1 Communication mechanism

There are two approaches to marshal data to be sent across the network or across processes - One is to use mechanisms that couple the schema information with the data values (like JSON) and the other is to use serialization approaches [2, 13] that decouple schema specification from data values. In SS-KVSTORE, requests and responses between the server and client are encoded in JSON format, since the size of schema specification adds only negligible overheads in comparison to the size of data.

3.2 Client library

The client library is implemented in Cython, it provides the client APIs specified in Pseudocode 1. This library can be compiled into a shared library that the applications can use to interface with SS-KVSTORE servers.

3.3 Server

The server is implemented as a multithreaded program in Java, and persists it's objects in SQLite database as described further in Section 3.3.1. We then go on to describe the network ports exposed by each server for external communication with clients and internal communication among servers in Section 3.3.2. Post this, we describe the internal threading structure of the server in Section 3.3.3. Next, we give a brief overview of how the server manages communication between these threads in Section 3.3.4. Finally, we describe an end-to-end control flow for client *GET* and *PUT* requests in Section 3.4.

3.3.1 Data Store

We use SQLite, a SQL based database to persistently store data in each server. The database consists of a table containing the following fields (refer Table 1):

1. **Key:** This is a column of string type that stores the key.
2. **Value:** This is a column of string type that stores the value corresponding to the key.
3. **Sequence Number:** This field essentially stores the version number of the corresponding key which can later be used to detect and resolve conflicts. The sequence number for a given key is assigned by the *primary server*, and is applied at non-primary servers on receipt of a broadcast message from the *primary server*.
4. **Dirty:** This is a boolean field indicating whether a valid sequence number has been assigned to a given key after a corresponding client write. For instance, if the client sends a PUT request to a server that is not the *primary server* for the key, the server writes the entry into it's database as dirty and forwards the PUT request to the server which is the *primary server* for the key. The *primary server* then assigns a sequence number to the write, writes it into it's database and broadcasts the update along with the assigned sequence number. On receipt of the broadcast, all other servers update the sequence number and invalidate the dirty bit, if set.
5. **Stale:** This is a boolean field set to indicate that the record might not be recent. If this field is set, the server reconciles the state of the record before serving the client request.

Key	Value	Sequence Number	Dirty	Stale

Table 1: Database table schema

Client requests are then translated into the following SQL commands:
Let's say that the client wants to **GET** the value of a key *K*:

SELECT value from T where key=K

Let's say that the client wants to **PUT** value *V* into key *K*, let's also assume that the current sequence number of the record is *seq*:

BEGIN
OLD_VALUE = SELECT value from T where key=K
INSERT INTO T (key, value, sequence number, dirty, stale)
VALUES (K, V, seq+1, 0, 0)
COMMIT

Note that the SQL transaction semantics are used to make client PUT requests atomic and isolated from concurrent transactions.

3.3.2 Network port configurations

The servers in SS-KVSTORE consist of the following configuration options:

- **External port (EP):** This is the port where the server receives and responds to client requests.
- **Internal port (IP):** This is the port which servers in SS-KVSTORE use for peer-to-peer communication with each other.
- **Multicast IP and port:** This is a cluster-wide configuration. Each server in SS-KVSTORE subscribes to messages on multicast receiver port *MRP* on a multicast IP address *MIP*.
- **Multicast sender IP and port:** Each server in the cluster creates a *DatagramSocket* at multicast sender port *MSP* to send packets to multicast IP address *MIP*.

3.3.3 Server threads

On startup, each server creates the following threads:

1. **Client Request Handler (CRH):** This thread creates a pool of server sockets listening on *EP*. This is the thread that responds to client GET and PUT requests.
2. **Multicast Sender (MS):** This thread creates a *DatagramSocket* on multicast IP *MIP* and port *MSP*.
3. **Multicast Receiver (MR):** This thread listens to multicast messages sent by servers on multicast IP *MIP* and port *MRP*.
4. **Internal Thread (IT):** This thread creates a server socket on port *IP* and listens to peer-to-peer communication requests from other servers in the cluster.

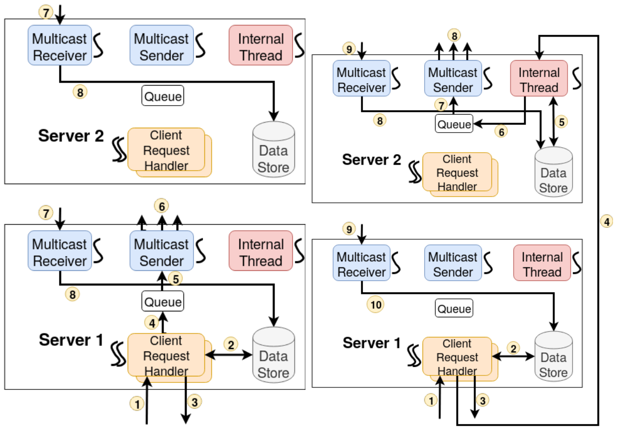
3.3.4 Queue

Each server has a thread-safe queue. Whenever a server needs to broadcast a message to all other servers in the cluster, it adds the message to this queue. Whenever a new message is added into this queue for broadcast, the multicast sender thread consumes and broadcasts this message. The following messages can be added to this queue for broadcast:

- **SERVER_UP:** When a server comes up, it adds a *SERVER_UP* message to it's queue.
- **SERVER_DOWN:** When a server finds out that another server is down, it adds a *SERVER_DOWN* message to it's queue.
- **PUT:** When a server writes a value corresponding to a key for which it is the *primary server*, it assigns a sequence number and adds the message along with the sequence number to it's queue.

3.4 Control flow for client requests

On receipt of a GET request, the server just responds with the value stored locally, if it exists. Otherwise, the server responds with a *null*.



(a) PUT to primary server (b) PUT to non-primary server

Figure 2: Control flow for client PUT requests to a primary and non-primary server

Figure 2(a) shows step-by-step how client PUT requests are handled if received by the primary server P:

1. *CRH* of P receives the PUT request.
2. *CRH* of P writes/updates the value locally, and increments the *sequence number*.
3. *CRH* of P returns the old value to the client.
4. *CRH* of P adds the request to the Queue along with it's assigned sequence number.
5. *MS* of P reads the message from the queue.
6. *MS* of P broadcasts the message read from the queue.
7. *MR* of all servers in the cluster receives the message.
8. *MR* of all servers in the cluster applies the received message locally.

Figure 2(b) shows step-by-step how client put requests are handled if received by a server that is not a primary server S:

1. *CRH* of S receives the PUT request.
2. *CRH* of S writes/updates the value locally, and marks the record as dirty.
3. *CRH* of S returns the old value to the client.
4. *CRH* of S forwards the request to the primary server P on it's *IT*.
5. *IT* of P writes/updates the value locally and assigns/increments the *sequence number*.
6. *IT* of P adds the message along with the sequence number to the Queue.
7. *MS* of P reads the message from the queue.
8. *MS* of P broadcasts the message read from the queue.
9. *MR* of all servers in the cluster receives the message.
10. *MR* of all servers in the cluster applies the received message locally.

4 Evaluation

We evaluate our system with respect to two aspects:

- **Correctness of the system:** We perform a series of correctness tests (Section 4.1) to evaluate the semantics of SS-KVSTORE both in scenarios with and without failures.

- **Performance of the system:** We primarily evaluate the performance of SS-KVSTORE when there are no failures (Section 4.2). We use a variety of workloads which have different popularity distributions for keys.

4.1 Correctness tests

Below we describe a series of tests and describe the correctness aspects of the system validated by each of the tests. Note that each of the below tests are executed assuming a fresh start for the key-value store, i.e, the key-value store is initially empty.

4.1.1 Test 1 - Single client without failures

Details. This test involves a single client issuing a sequence of write and read requests. We start up three server processes as a part of the SS-KVSTORE service and assume that none of the server processes fail. We then configure the client to be able to communicate with any of the above 3 servers by passing the server names of all three servers. As a first step, the client writes 1000 randomly generated keys and values to SS-KVSTORE. During this time, it checks that the write was successful and asserts that the return value for *kv739_put* is 0 since there is no *old_value*. After this, we set the client to sleep for a configurable amount of time. This sleep time will allow the *PUT* requests to replicate among servers. We then read back the values for the same keys and check if the written values can be read back successfully. Since we require eventual consistency, we measure the minimum amount of sleep time needed between the write phase and read phase in order for all values to be read back successfully. We repeat the same exercise for multiple iterations.
Results. We notice that SS-KVSTORE is able to provide strong consistency for the *atomic put and read* operation, which is evident from the fact that we did not require any sleep between successive iterations of *PUT* requests. This is because in the absence of failures, write requests from a client always go to the *same* CONNECT SERVER. Additionally, we also observe that no sleep time was required after an iteration of writes in order to get strong consistency for the reads that follow. We did not expect this as our read protocol returns the majority value from all servers. Hence, we expected a small sleep time to be required in order for results to propagate. We believe that we did not notice this because the time for replicating among servers mostly overlapped with the time to complete the iteration of writes. We focus more on this aspect through *Test 2*.

For our partner team, we also noticed that no sleep time was required after an iteration of writes. However, we were not able to stress test their system due to crashes. When the test involved running 5 iterations of our workload, all reads were successful. However, we noticed that 84% of the writes returned failure in one of the iterations. When the test involved running 10 iterations, we observed that the last iteration of reads had 84% failures and all successive writes failed.

4.1.2 Test 2 - Concurrent clients without failures

Details. This test involves 2 concurrent clients that interact with different partitions of servers. We start up three server processes as a part of the SS-KVSTORE service and assume that none of the server processes fail. On starting up, we issue *fork()* to create a parent process and child process corresponding to client 1 and client 2 respectively. The two processes coordinate with each other using shared semaphores. We configure client 1 to communicate only with server 1 and client 2 to communicate only with server 2 using the *kv_init* API. Client 1 writes a value to SS-KVSTORE, wakes up Client 2 to read the value for the same key, and then waits to be signalled by Client 2. Client 2 can sleep for a configurable amount of time before reading to allow for write operations to propagate across servers. After reading the value, Client 2 signals Client 1 to write the value for the next key and then waits to be once again signalled by Client 1. The outline of the test is shown below:

```
sem_t *sem1 = sem_open("kvstore1", O_CREAT, S_IRUSR
     | S_IWUSR, 0);
sem_t *sem2 = sem_open("kvstore2", O_CREAT, S_IRUSR
     | S_IWUSR, 0);
int pid = fork();
if(pid == 0) {
    // Init to interact with server 2
   for(int i=0;i<KVSTORE_SIZE;i++) {
      int value;
      sem_wait(sem1);
       usleep(SLEEP_TIME_US);
       ret = kv739_get(keys[i], old_val);
       // validate result
      sem_post(sem2);
    }
} else {
   // Init to interact with server 1
    for(int i=0;i<KVSTORE_SIZE;i++) {
        ret = kv739_put(keys[i], values[i], old_val);
       sem_post(sem1);
        // validate result
       sem_wait(sem2);
    }
}
```

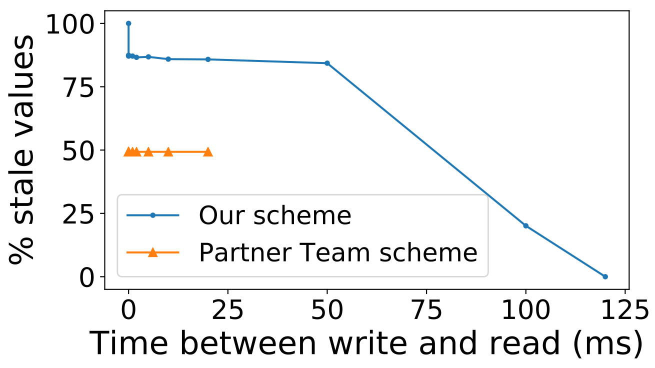


Figure 3: Comparison of time taken for updates to propagate across servers.

This test checks that write operations are getting propagated across servers and also measures the time required for this propagation through the configurable *SLEEP_TIME_US* parameter.

Results. Figure 3 show the variation in the percentage of stale reads that we get as we vary *SLEEP_TIME_US*. We notice that SS-KVSTORE is able to successfully propagate values to all servers in ~125 ms.

For our partner team, we noticed that there is a significantly lesser percentage of stale values when *SLEEP_TIME_US* is set to zero. We tried to estimate the minimum value of *SLEEP_TIME_US* for which no stale reads are observed. We noticed that upto *25ms*, the percentage of stale values remained ~50%. Increasing the *SLEEP_TIME_US* beyond this caused their system to crash.

4.1.3 Single client with server failures

Details. This test involves a single client issuing a sequence of write and read requests but in the presence of failures. The setting is essentially the same as that in *Test 1* but with two scenarios - (i) We kill one of the servers after writing 1000 values and before reading back the values. (ii) We kill two servers after writing and before reading.

Results. For SS-KVSTORE, we observe that it is largely able to tolerate both single and double server failures. The client library is able to transparently handle getting majority value from the servers that are up during a read operation. We do however notice that sometimes a server may be killed by the time it is able to propagate its updates to other servers. In such a scenario, we notice stale value reads. We used two mechanisms to verify the behavior of our system in such scenarios - (i) We introduce a small delay after completing all writes, *SLEEP_TIME_US* before killing the server processes and starting reads. When the value of *SLEEP_TIME_US* is set to ~125 ms (following from 3), we notice that there are no stale values since all writes are propagated to all servers before the server(s) get killed. (ii) After observing stale values, we retry the read operations after restarting the killed server(s). We observe that once we do this, we are able to read back values without any staleness. We attribute this to to our design choice of designating written entries as dirty and re-transmitting dirty key entries upon a restart.

For our partner team, we had to make the following changes to their scripts in order to simulate failures:

1. The server code had a logic where the server gets shutdown when it has not received a connection for 5 seconds. We think that this logic is incorrect and that a server should be running continuously unless manually killed. We modified their code to remove this logic, so that we can have servers run continuously, and hence enable us to simulate server failures.
2. The script that starts up servers was also responsible for executing the test. For simulating failures, we had to decouple the starting up of servers from execution of tests.

Despite these efforts, we were unable to get our failure simulation tests working, as *kv_init* always returned a *-1*. Hence, we cannot present any evaluation results for failure handling. We have shared the details of all tests with the partner team to enable them to debug further.

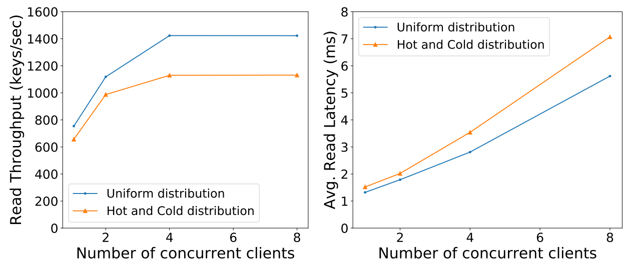
4.2 Performance tests

In this section, we evaluate the performance of SS-KVSTORE. For sake of uniformity, we assume that all keys are of size 128 bytes and all values are of size 512 bytes. We study the performance of the system using two workloads in the absence of failures.

- **Uniform distribution.** In this setup, we consider SS-KVSTORE having 1000 keys. First, we perform 10,000 read operations and select a key at random to read. Next, we perform 10,000 write operations and select a key at random to write a new value for.
- **Hot and Cold distribution.** In this workload setup as well, we consider 1000 keys. However, unlike uniform distribution, 10% of the keys are designated as *hot* and are accessed 90% of the time. The remaining 90% of the keys are *cold keys* and are accessed only 10% of the time.

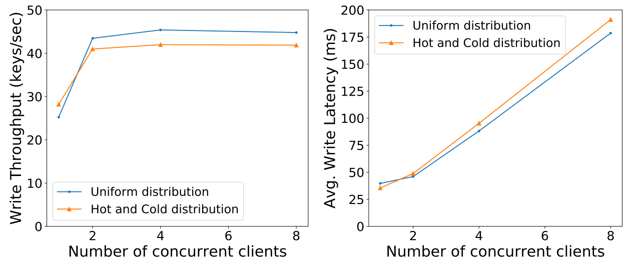
Figure 4 and Figure 5 are macro-benchmarks that show the performance of the system as the number of concurrent client applications utilizing the system changes. In the setup, we consider SS-KVSTORE having 3 servers.

From Figure 4(a), we notice that read throughput first increases and then saturates as the number of concurrent applications increases. This trend is expected since the system has a fixed capacity of requests that it can handle. We also observe that SS-KVSTORE is able to give better throughput with the *uniform workload*, which could be an artifact of



(a) Throughput (b) Latency

Figure 4: Comparison of read throughputs and latencies for Uniform and Hot and Cold workloads as the number of concurrent clients change

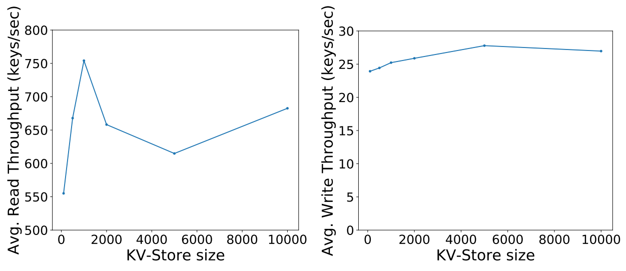


(a) Throughput (b) Latency

Figure 5: Comparison of write throughputs and latencies for Uniform and Hot and Cold workloads as the number of concurrent clients change

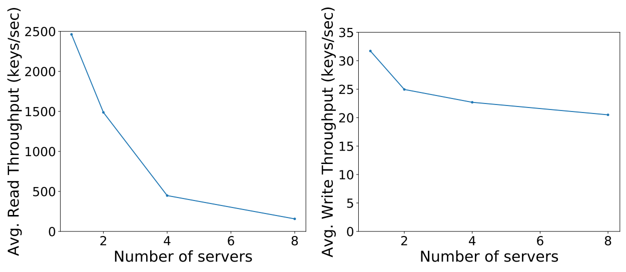
contention for reading on the same row in the SQL database. One potential problem in our implementation is that even though we allow for multiple threads to receive input requests (multiple *CRHs*), we utilize only a single thread for database operations. To solve this problem, we could potentially have each *CRH* have it's own database thread for better scalability. In such a scenario, the SQL database would essentially handle concurrency control and isolation between concurrent transactions from mutliple *CRHs*. Similarly, from Figure 4 (b), we notice increasing latencies for both workloads as the number of concurrent client applications increase. Once again, this can be attributed to same reason and issue as described above. Similar trends are observed with respect to the write throughput and latencies as observed in Figure 5.

Next, we consider a micro-benchmark where we evaluate the performance of SS-KVSTORE when the number of key-value pairs stored by the service changes. In this test, we consider that only a single client uses the service and that there are 3 servers in total. We do not show latency trends, since when there is a single client, latency numbers can be easily derived as the inverse of throughput. We expect both read and write throughputs to more or less remain the same irrespective of the size of SS-KVSTORE. From our measurements (Figure 6), we observe that this largely remains the case except for



(a) Read Throughput (b) Write Throughput

Figure 6: Comparison of read and write throughputs as the size of SS-KVSTORE *changes*



(a) Read Throughput (b) Write Throughput

Figure 7: Comparison of read and write throughputs as the number of servers in SS-KVSTORE *changes*

some variations which we believe are due to disturbances on our test environment. Please note that we did run multiple iterations of the test and the values presented are the average values across all runs.

Finally, we consider another micro-benchmark where we evaluate the performance of SS-KVSTORE while we vary the number of servers. In this test, we use a single client and assume that there are 1000 key-value pairs. We expected read throughput to remain nearly the same since our protocol issued reads to all servers in parallel. However, from our measurements (Figure 7(a)), we noticed a huge degradation in read performance as the number of servers increased. Upon further investigation, we found that even though we were using threads, the execution of the threads got falsely serialized due to Python's *Global Interpreter Lock*. A possible fix to this problem would be to use Python's *asyncio* feature, but we did not have the time to try out this fix. We expected write throughput to slightly degrade as the number of servers increased due to the need to perform more replication. Our measurements as shown in Figure 7(b) reflect that this trend indeed holds.

5 Conclusion

We have successfully designed, implemented, and evaluated a key-value store. The design of SS-KVSTORE shows that one can design a highly reliable and performant key-value store by storing only minimal auxiliary information such as *sequence numbers, dirty flags, and staleness information*. Our evaluation shows that we are able to give the stipulated consistency guarantees and that our system is sufficiently performant.

6 Acknowledgements

We would like to thank Prof. Michael Swift for the interesting mini-project and our partner team members Nick Daly and Sek Cheong for providing us with their implementation to present results on our correctness tests. We would also like to thank our batchmates for raising important observations on Piazza.

References

[1] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 1–14.

[2] Apache Avro. http://avro.apache.org/docs/1.9.1/, 2018.

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 205–218.

[4] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[5] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 401–414.

[6] Escriva, R., Wong, B., and Sirer, E. G. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 25–36.

[7] AWS Firecracker. https://github.com/firecracker-microvm/firecracker, 2018.

[8] Kalia, A., Kaminsky, M., and Andersen, D. G. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 185–201.

[9] Kubernetes. https://kubernetes.io/, 2019.

[10] Li, B., Ruan, Z., Xiao, W., Lu, Y., Xiong, Y., Putnam, A., Chen, E., and Zhang, L. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 137–152.

[11] Merkel, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[12] Mitchell, C., Geng, Y., and Li, J. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 103–114.

[13] Varda, K. Protocol buffers: Google's data interchange format. Tech. rep., Google, 6 2008.