PmWIki /

Project1

The goal for this project is to implement a small distributed system, to get experience in cooperatively developing a protocol specification, and to get experience benchmarking a system.

You should do this project in groups of 2 or 3.

Notes and additions

- To enable writing the client in different languages, it is fine o.k. to implement a client interface and share the code for those interfaces rather than a binary library.

Due date

- The project is due on Thursday, September 25th at midnight

Groups

- This project must be done in groups of 2-3 people.
- Your group must partner with one other group (or two if we have add odd number of groups).

Service

The service to provide is a simple key-value store, where keys and values are strings. You service should be as consistent as possible, so that requesting a value should return the most recently value set as often as possible. Furthermore, your service should recover from failures, so you will need to store data persistently.

There is no standard protocol. You must work with your partner group to create a protocol.

Client library

You must implement a client library to access key/value data. The interface is detailed in the specification below.

You will provide a shared library named "lib739kv.so" implementing the interface to your server. The client code must run on standard CS department workstations.

Here are the functions you must implement:

- int kv739_init(char *server) - provide the name of the server in the format "host:port" and initialize the client code. Returns 0 on success and -1 on failure.
- int kv739_get(char * key, char * value) - retrieve the value corresponding to the key. If the key is

present, it should return 0 and store the value in the provided string. The string must be at least 1 byte larger than the maximum allowed value. If the key is not present, it should return 1. If there is a failure, it should return -1.
- int kv739_put(char * key, char * value, char * old_value) - Perform a get operation on the current value into old_value and then store the specified value. Should return 0 on success if there is an old value, 1 on success if there was no old value, and -1 on failure. The old_value parameter must be at least one byte larger than the maximum value size.

There are restrictions on keys and values:
- Keys are valid printable ASCII strings without special characters and are 128 or less bytes in length. They cannot include the "[" or "]" characters.
- Values are valid printable ASCII strings with neither special characters nor UUencoded characters and 2048 or less bytes in length. They cannot include the "[" or "]" characters.

Tests

You should write a test program that links against the shared library described above. You should write two kinds of tests:
1. Correctness tests: ensure the code does what it is supposed to (including failure/recovery)
2. Performance tests: measure the performance, as latency and throughput, for some workloads

You must test your code in three configurations
- Your tests, your library, your server.
- Your test, your partner's library, your server.
- Your partner's tests, your partner's library, your server.

Implementation

You may use any implementation language for the server and client, as long as it implements the required API and protocol. For example, you can use regular sockets, RPC Google's protocol buffers for communication between your servers, or an HTTP implementation.

What to turn in

You will turn in a short report describing your effort that includes:
1. Your names, and the names of the members of your partner group
2. A short discussion of how you implemented your client and server
3. A specification of the protocol between client and server
4. A description of the tests you wrote and an explanation why they are appropriate/suitable for testing your service. Please include the test methodology
5. The results of the three sets of tests

Grading

Performance is not the primary concern of this project. We are instead interested in the ability to return consistent results under failure conditions.

For the writeup, we will look for these things:

1. Does the writeup adequately describe the project?
2. Does the design make sense?
3. Does the described tests adequately test the guarantees of the design?
4. Does the code interoperate correctly?

Page last modified on September 12, 2014, at 02:31 PM

Group Names: Sreeja Thummala
Kai Zhao

Partner Group Names: Prashant Chadda
Chetan Patil
Bhaskar Pratap

University of Wisconsin-Madison
Department of Computer Sciences
CS 736 Distributed Systems
2012 September 25

Title

A Key-Value Store System using Caches, Append Only Logs, and B-Tree

Abstract

Data storage is one of the most popular cloud services. Key value stores are a popular option for storing billions of key-value pairs. Building a key-value store for high performance, reliability, efficiency, and scalability starts with first building a single reliable and efficient server to service multiple clients. The scalability of the key-value store can then be accomplished by running the server of multiple machines and adding efficient communication and scheduling protocols.

This paper presents the design, implementation, and performance of a single key-value storage system. The system considers network bandwidth, faulty server, correctness, and performance. The system is able to handle inits, gets of present and absent keys, puts of present and absent old_value, and server faults. Furthermore, the system handles requests from 20 microseconds for gets to 28 milliseconds for gets.

1. Introduction

Data storage such as Dropbox, Box, and Google Drive is one of the most popular cloud services. Key value stores are a popular option for storing billions of key-value pairs. A minor inefficiency, such as using extra network bandwidth, in a single server will cause the inefficiencies will constructively for large delays when the system is scaled with more server nodes. Therefore, the key to building a large, fast, reliable, and efficient system is to scale from a single fast, reliable, and efficient server. Although scalability can be considered later, this paper will present a server that trades scalability for speed and reliability.

This paper is structured as followed. Section 2 presents the client implementation, section 3 presents protocol implementation, section 4 presents the server implementation, section 5 presents and discusses the correctness tests, section 6 presents and discusses the performance tests, section 7 presents the test methodology, section 8 presents the results and discussion, and section 9 is the conclusion and lessons learned.

2. Client Implementation

The client implements kv739_init, kv739_get, and kv739_put. kv739_init opens the socket connection between the client and the server, and should be called before any other communication. kv739_get reads the value corresponding to key. kv739_get will return 1 and a null value if key is not present in the database. kv739_put reads the old_value corresponding to key. kv739_put will return 1 and null value if old_value is not present in the database.

3. Protocol Implementation

Our group decided to use TCP sockets for communication using variable messages. The protocol will send a few bits to signify how many bytes to read from the socket buffer to obtain the key and value. Using variable length messages reduces network congestion because the client and server can continue after reading the specified number of bytes, rather than reading the maximum length of keys and values. At worst, even if the key and value is at maximum length, using a header will only cost a 1.1% overhead.

The client to server request consists of a request header and request body (Figure 1). The request header contains 2 bits for the command (init, get, or put), 1 byte for key_len (key length, which ranges from 0 to 128), 2 bytes for arg1_len (value, which ranges from 0 to 2048), and 2 bytes for arg2_len (old_value, which ranges from 0 to 2048). The request header will inform the server of how many more bytes to read to obtain the key and value.

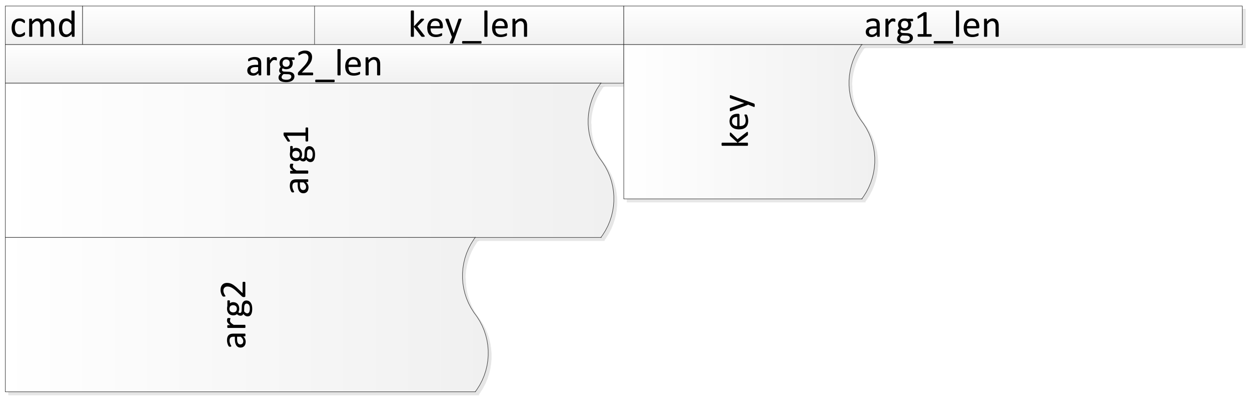

Figure 1: Request message. The client will build this message to send to server.

The server then reads/writes to/from the database to form a server to client response (Figure 2). The server to client response consists of a response header and response body. The response header contains 1 bit for the code (0 means success, 1 means failure), 1 byte for key_len (key length, which ranges from 0 to 128), and 2 bytes for value_len (value length, which ranges from 0 to 2048). The response header will inform the client of how many more bytes to read to obtain the key (used just for confirmation) and value (value for get, old_value for put).

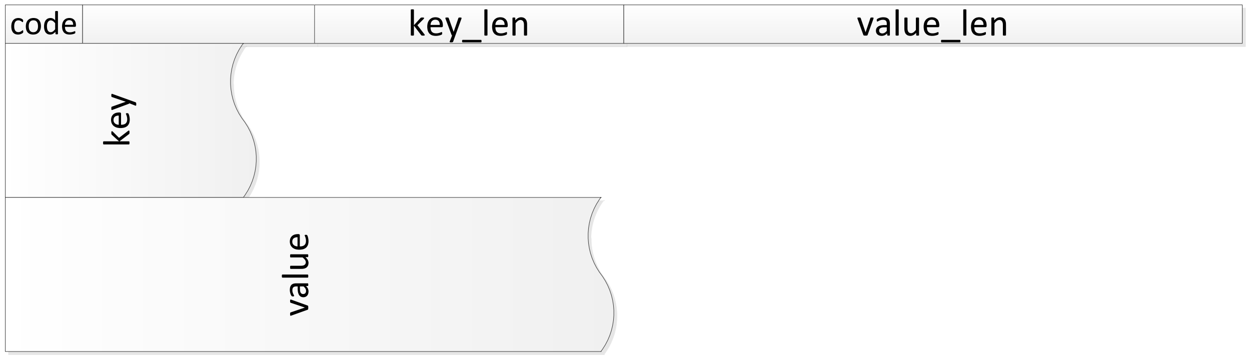

Figure 2: Response message. The server will build this message to send to client.

4. Server Implementation

Our key-value store server design priorities are consistency, failure recovery, and fast reads and writes. These three priorities have conflicting optimizations. For instance, the data can be consistent if the server always reads from and writes to disk, but this will cause slow read and write speeds. On the other hand, we can have fast reads and writes by implementing cache on the client side, but doing so will eliminate consistency because updates from other clients will not be detected. Therefore, we need to balance between consistencies, failure recovery, and fast reads and writes.

The server needs to eventually write every key-value pair to disk. Since disk accesses are very expensive, we implement a B-tree to index the data stored into the disk. B-trees are balanced trees that are optimized for situations when part or all of the tree must be maintained in secondary storage such as a magnetic disk. Since disk accesses are slow, b-tree tries to minimize the number of disk accesses. For instance, a b-tree with a height of 2 and a branching factor of 1001 can store over one billion keys, but requires at most two disk accesses to search for any node [1].

Although b-tree is implemented to index data in the disk, the reads and writes are still not fast enough. Therefore, we implemented a Least Recently Used (LRU) cache in the memory of the server to improve reads of recent keys. The server will first attempt the cache for all reads because cache reads has magnitudes of performance over disk reads. We did not depend on the cache for the writes because cache does not persist to disk. Therefore, the server will still perform writes on disk for the sake of consistency and failure recovery if the server were to crash.

Writing to append-only log is less expensive than writing to b-tree which requires random reads and writes [2]. So, our server writes to the append only log first, and then a separate thread writes to b-tree.

When a get() operation is requested, the server will first searches cache for the key. If key is not present in the cache, it reads the logs to search for the key. If the key is still not present, then it searches b-tree. Although this increases the complexity or reads, we believe that restricting the number of logs to 5, and then adding cache will minimize overhead and significantly improve writes.

When the number of records in logs increase to a MAX_RECORD_COUNT, a new log is created based on the timestamp, and the old log file is queued in a shared queue. The elements of the shared queue are removed and processed by the indexer thread. The indexer thread removes one element from the shared queue to write to b-tree. When all the log records are written to the b-tree, the thread blocks access and deletes the log. If the number of logs increased to 5, then writes every log to b-tree.

For a get() operation, the server tries to search the cache for the key. If present, it returns the value. If not, it searches the logs. If key is present in the logs, it is returned. Otherwise it searches the underlying b-tree. If still key is not found, it sends an error response.

The consistency is maintained in our design as logs and the records in logs are searched in Last In First Out basis. The recovery failure is attained as everything is stored in the secondary storage in the form of .log files (append only log files) and .dat file (data blocks of b-tree). When the database is restarted, the system initializes the b-tree using the .log files and the .dat file. Hence, any data written by the server is present in the b-tree for use after it is bootstrapped.

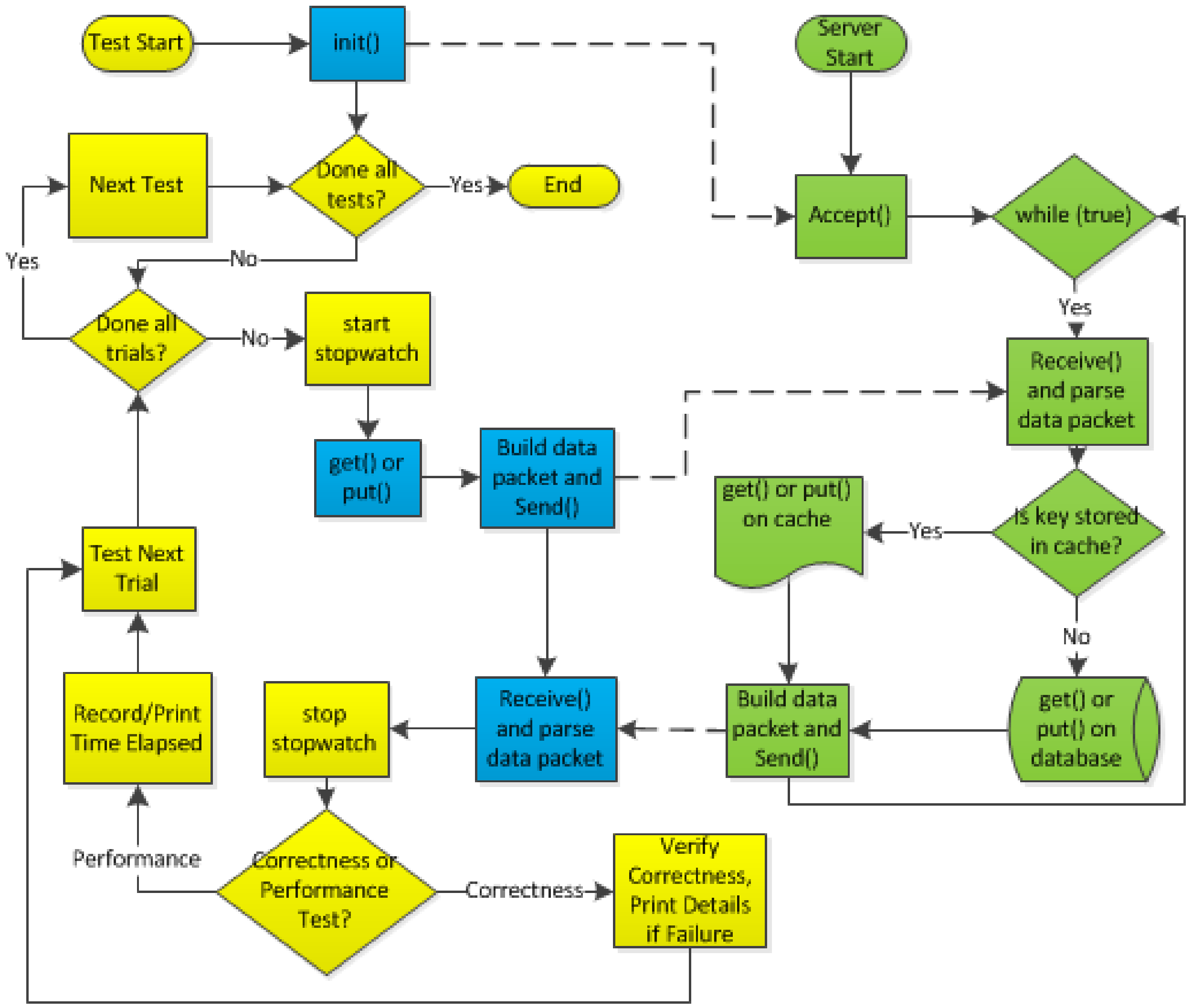
Test Start
init()
Server Start
Next Test
Done all tests?
Yes
End
Accept()
while (true)
Yes
No
Done all trials?
No
start stopwatch
Yes
Receive() and parse data packet
get() or put()
Build data packet and Send()
get() or put() on cache
Yes
Is key stored in cache?
Test Next Trial
No
Record/Print Time Elapsed
stop stopwatch
Receive() and parse data packet
Build data packet and Send()
get() or put() on database
Performance
Correctness or Performance Test?
Correctness
Verify Correctness, Print Details if Failure

Figure 3: Overview of the client and server interaction. The client consists of the test cases (yellow) and the library (blue). The test cases calls the library to send a request (dotted lines) to the server (green). The server will check the cache and/or database and send a response back to the test cases.

5. Correctness Tests

1) Test init. This tests whether the connection to a socket was successful. If not, then either the protocol was buggy, client is using the wrong host, or the client is using the wrong port.
2) Test writing a value to a key. This test will check whether kv739_put works without any segmentation faults.

3) Test reading the written value and test if the read_value matches written_value. This test checks whether both the client and server can package and receive data packets that matches the network protocol specifications.
4) Test overwriting a key and test if the old_value matches written_value. This test checks whether values can be updated without causes primary key conflicts on database inserts.
5) Test reading the overwritten value and test if the read_value matches the overwritten_value. This test checks whether keys can be updated in the database correctly. Furthermore, since the old_value is replaced in the database, reading the current value should not have any residue of the old_value, which sometimes occur if the new value is shorter than the old value.
6) Test reading a written value twice and test if the two read_values are consistent. Unless there are multiple clients, this test checks that the read is non-corrupting. The client should retrieve the same value on every read.
7) Test for successful writes and reads for keys with 1 to 128 valid characters, and unsuccessful for keys with invalid characters or with length greater than 128. This test checks whether the keys meets the specifications.
8) Test for successful writes and reads for values with 1 to 2048 valid characters, and unsuccessful for values with invalid characters or with length greater than 2048. This test checks whether the values meets the specifications.
9) Test reading from a server that just crashed after multiple write. This test checks whether the server can recovery after failure. And if so, what is the last write that the server is able to recover.

6. Performance Tests

1) Test speed of writing the same key and same value multiple times. This test checks whether the server performs faster or slower as write count increases. This also checks if there is a write buffer on the client side that will only push to the client after certain number of keys are updated.
2) Test speed of reading the same key multiple times. This test check whether the client cache and hash any recent information to avoid slow server communication for requests on the same key, in case the user likes to constantly refresh the webpage.
3) Test speed of writes after writes of different keys and different values. This test will benchmark how the server handles many write requests, in case the user likes to save changes a lot when there are no changes to be made.
4) Test speed of reads after reads of different keys. This test uses the same keys from the previous test to benchmark how the server handles many read requests that hits.
5) Test speed of reads after reads of random keys. This test uses the completely random strings of keys to benchmark how the server handles many read requests that misses.
6) Test read and write speed of various key and value lengths. This test the write algorithm of the server into cache (e.g. whether the server uses write first-fit or write best-fit).

7. Test methodology

The performance measurements were conducted on computers in University of Wisconsin-Madison Computer Sciences Department's instructional labs, galapagos-14 (client) and galapagos-13 (server). Both of these machines have i5-4570 quad-core processors @ 3.20GHz with 6 MB cache and 16 GB of main memory. The client and servers were remote on different computers to better model distributed systems that needs to communicate with a network. Furthermore, there tests were done in the instructional labs to attempt to make the performance data more comparable if other groups used the instructional labs as well.

The test suite contains correctness tests and performance tests. The testing was conducted by starting the server, connecting the client to the correct host and port, and letting the test suite execute each test case sequentially and automatically. The tests ran in three different configurations: 1) our tests, our library, and our server; 2) our tests, partner's library, and our server; and 3) partner's tests, partner's library, and our server.

8. Results and Discussion

The results uses puts and writes interchangeably and gets and reads interchangeably.

Correctness Test	**Result**	**Comments/Meaning**
init	passed	correct host and port
put	passed	correct library usage

get after put	passed	correct protocol usage
put after put	passed	no primary key conflicts
get after put after put	passed	correct updates
get after get after put	passed	consistent reads
key validity	passed	successful valid keys, unsuccessful invalid keys
value validity	passed	successful valid values, unsuccessful invalid values
server crash	passed	failure recovery
partner group's put test	passed	handles thousands of consecutive puts
partner group's get test	passed	handles thousands of consecutive gets

Table 1: Correctness tests results of all three configurations. All correctness tests passed. Every passed correctness test means that our server is capable of handling certain test cases.

Figures 4 through 9 shows box whiskers plots to highlight 5 percentile, 25 percentile, 50 percentile, 75 percentile, and 95 percentile. Figure 4 is the only figure that reads the same data to check the performance of cache. Figures 5 through 6 is the latency and throughput of sequential reads and writes of different data, where cache will not increase the performance. Figure 9 is the same as figure data as Figure 8, but uses our partner's library. Last, Figure 10 is the throughput of configuration 3, our partner group's test.

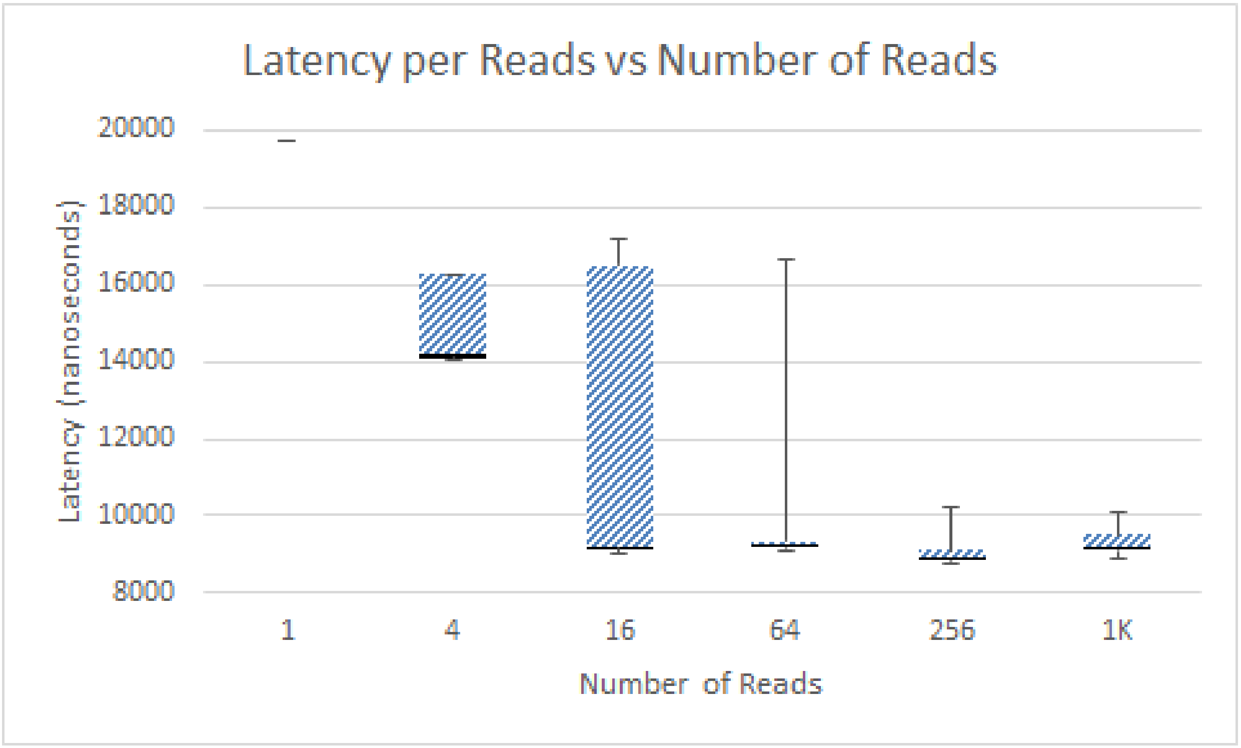

Figure 4: Latency per read vs number of reads of same data on configuration 1. This figure shows that it is increasing more efficient to read the same data up to 256 reads. 1 read is slow because the data is not stored in cache. 4 reads will have 1 read from disk and 3 reads from cache, making it faster. More reads will have more cache hits, leading to better read performance.

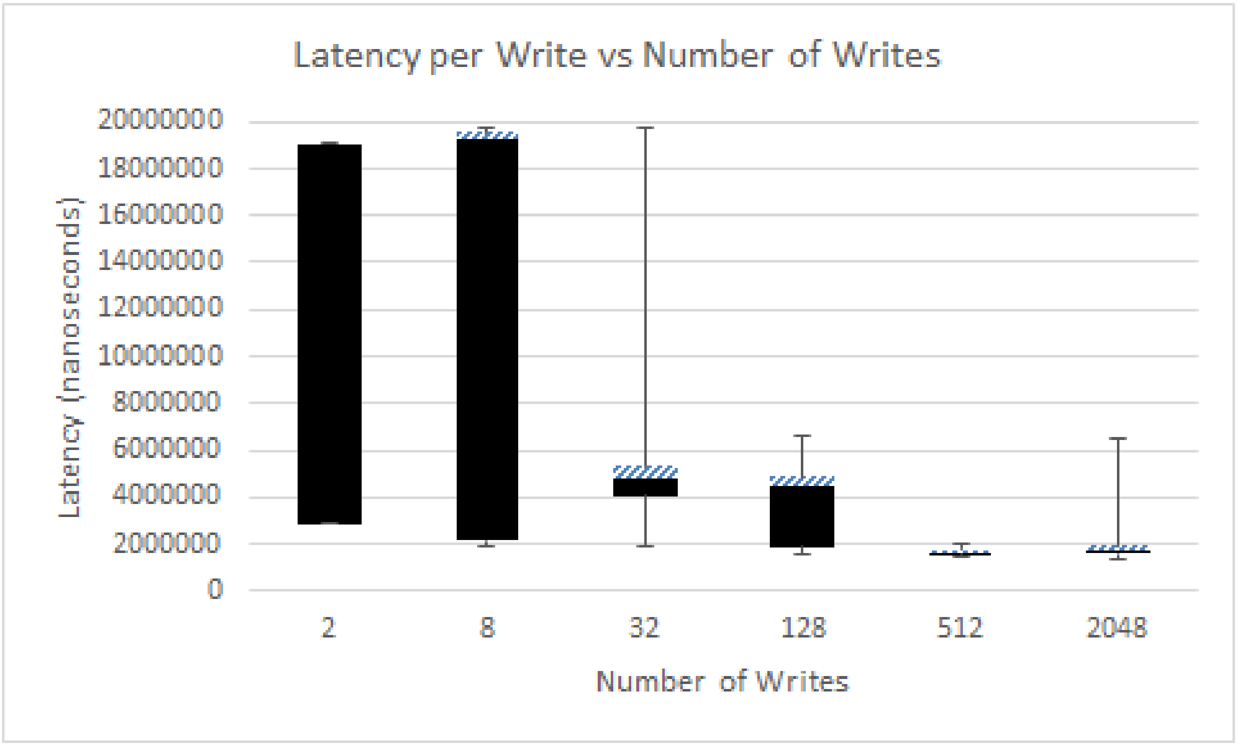

Figure 5: Latency per write vs number of writes of different data on configuration 1. This figure shows that it is increasing more efficient to write more data up to 512 writes. The average write speed is about 4 milliseconds.

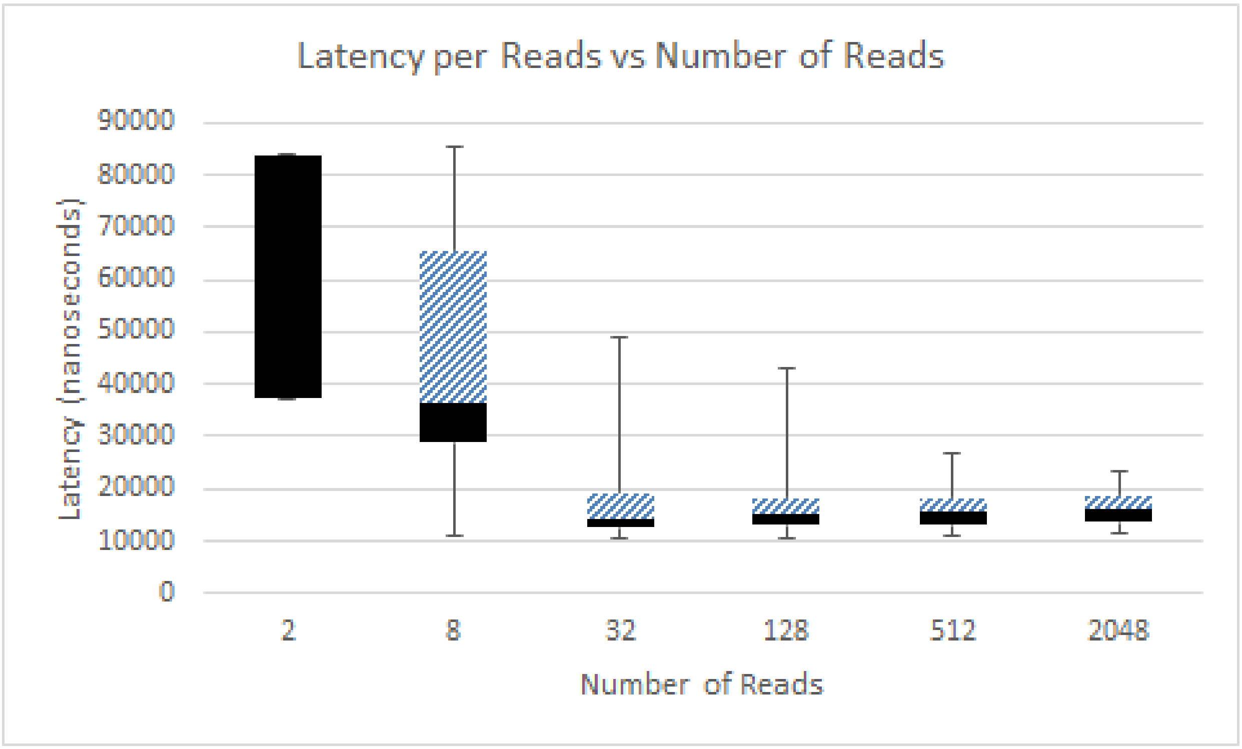

Figure 6: Latency per read vs number of reads of different data on configuration 1. This figure shows that it is increasing more efficient to read more data up to 32 reads. The average read speed is about 30 microseconds.

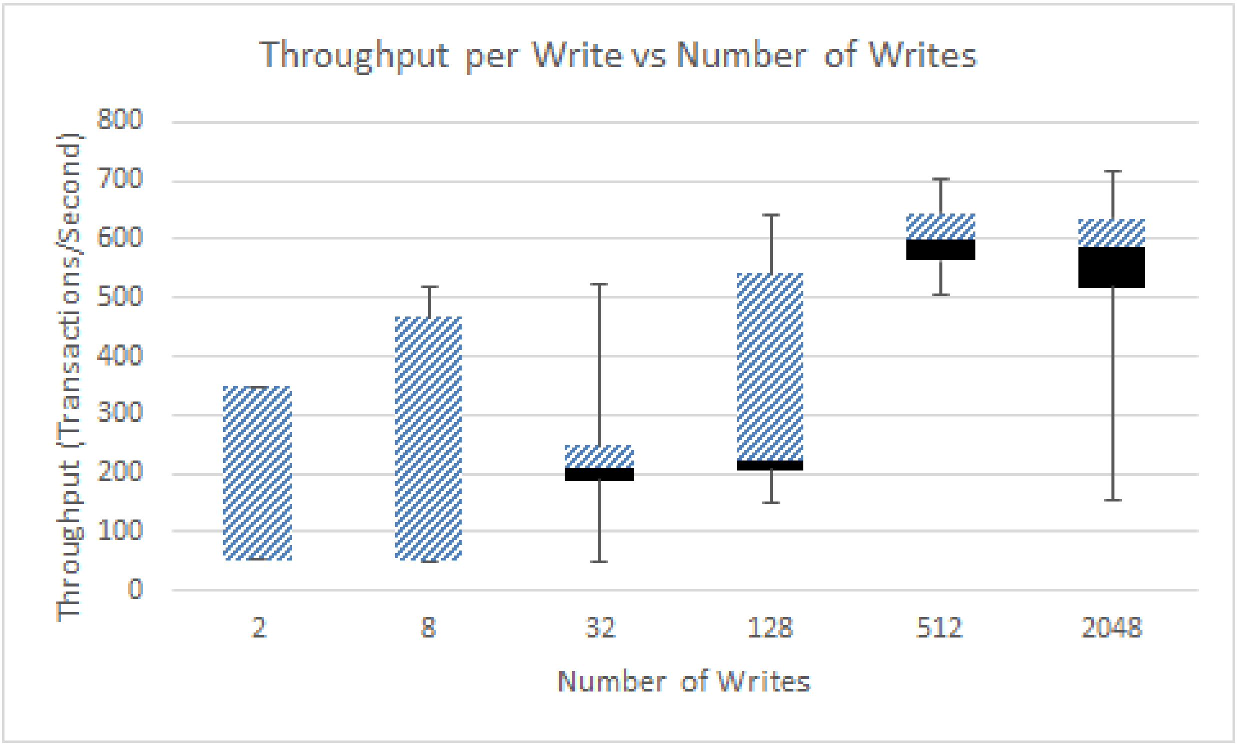

Figure 7: Throughput per write vs number of writes of different data on configuration 1. The average throughput is about 400 writes per second.

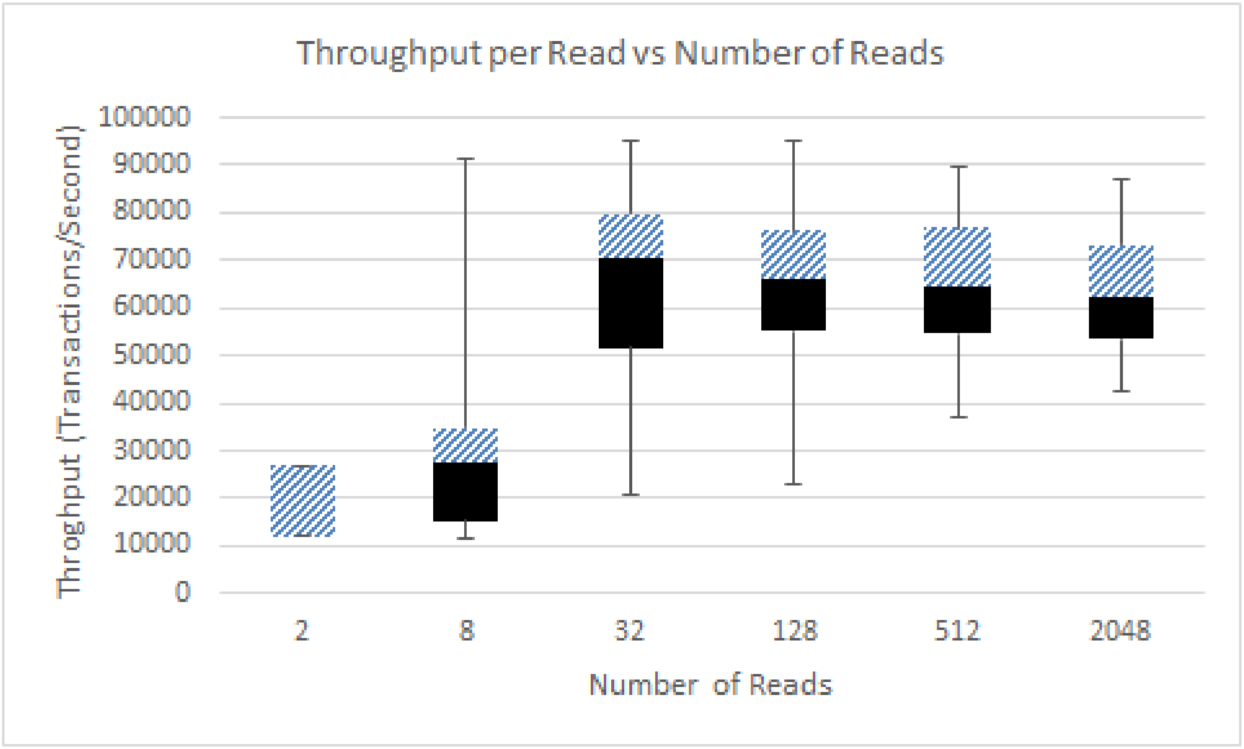

Figure 8: Throughput per read vs number of reads of different data on configuration 1. The average throughput is about 70,000 reads per second.

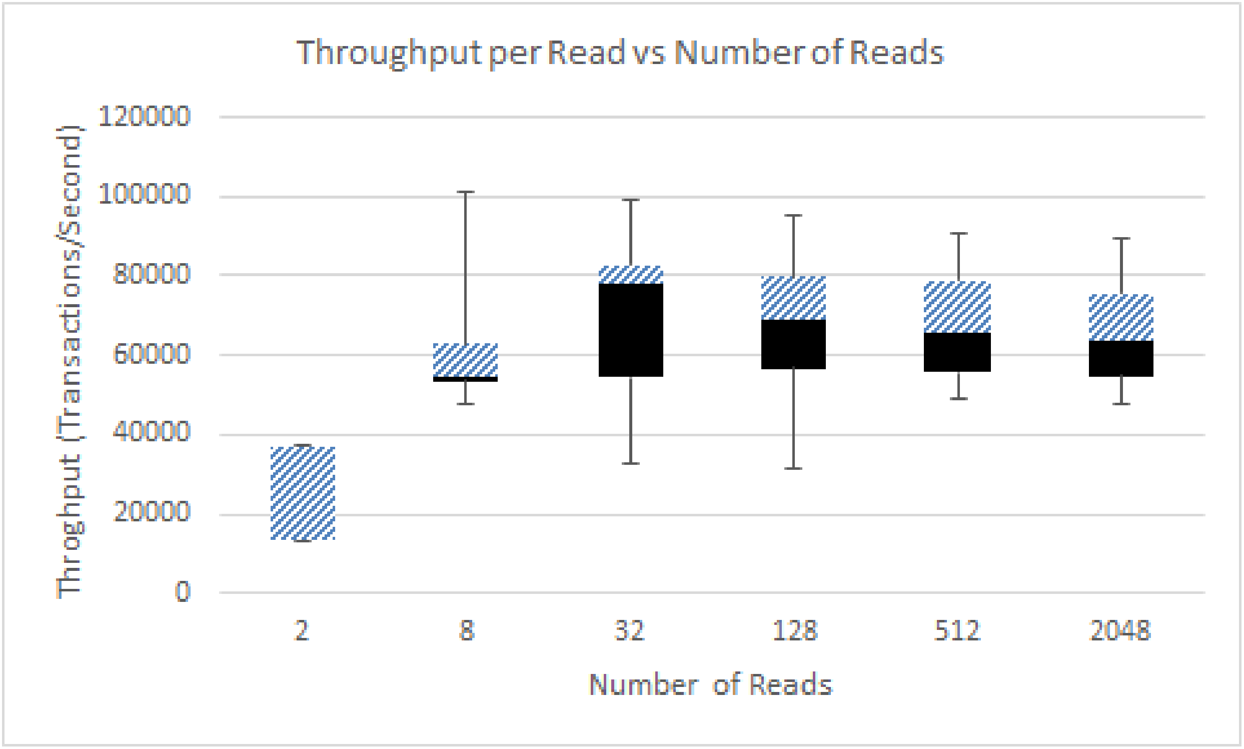

Figure 9: Throughput per read vs number of reads of different data on configuration 2. The average throughput is about 70,000 reads per second, which is the same as the results on configuration 1. Since the libraries only builds and parses data packets, having the same statistically equivalent data between different libraries makes sense.

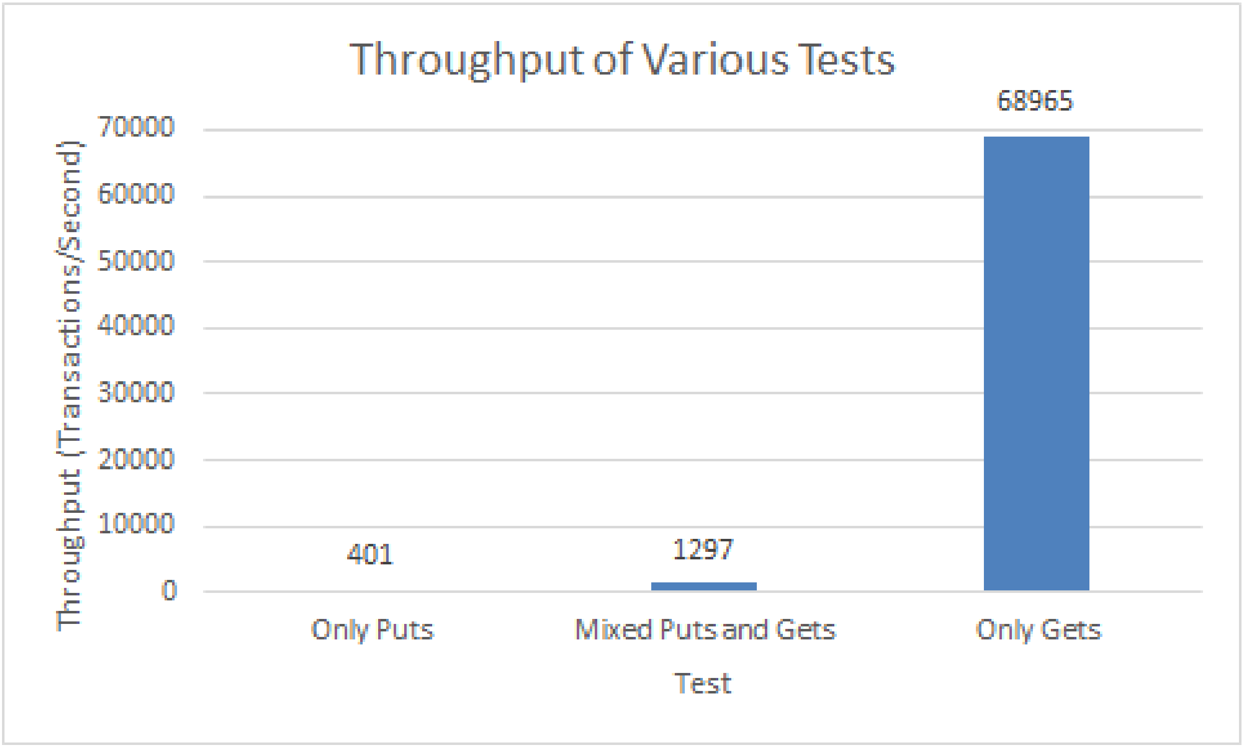

Figure 10: Throughput per different transactions of random data on configuration 3. The average throughput is about is about 401 puts per second. The average throughput is about 1,300 transaction for second for 50/50 reads and writes. The average throughput is about is 69,000 gets per second.

9. Conclusion and Lessons Learned

We designed and implemented consistent key-value store. We tested the key-value store for correctness and performance using three configurations. We found that due to similarity of client implementations between partner's group and ours, the test results in the configurations 1 and 2 are almost similar. The tests results of the partner's group tests also appear to be consistent with the tests results of our group.

It is difficult to claim any lessons learned as we did not test the performance difference of any one implementation. For instance, it is difficult claim how much faster is the system using an append only log because we did not compare to a system without an append only log. Furthermore, although our server's results is different from our partner's we are unsure of our partner's group's implementation, which again adds to the difficulty making any claims on our lessons learned. However, every implementation should cause a performance improvement from its explanation.

References

[1] Comer, Douglas. "Ubiquitous B-tree." *ACM Computing Surveys (CSUR)* 11, no. 2 (1979): 121-137.
[2] Hitz, David, Michael Malcolm, James Lau, and Byron Rakitzis. "Copy on write file system consistency and block usage." U.S. Patent 6,721,764, issued April 13, 2004.