CS 739 - Project 1

Saikat Raphael Gomes, Jason Feriante, Rahul Chatterjee
{saikat, feriante, rchat}@cs.wisc.edu

Partner Group-1: Anusha Dasarakothapalli, Adalbert Gerald Soosai Raj, Navneet Potti
Partner Group-2: Arkodeb Dasgupta, Brandon Davis, Chaitan Prakash

1. Implementation

We wrote our Key-Value store server in Python2.7 using Flask (a micro-framework) which assists with building, sending and receiving (parsing) HTTP Requests and Responses. We used SQLite3 to maintain persistent data storage for the key-value pairs. We also used the urllib2 python module to modify HTTP packets to comply with request standards on the client side. We implemented a cache layer in our application to make the get() operations fast. Our cache is an in-memory python dictionary which implements the least frequently used (LFU) eviction policy. All requests are routed to the cache first so that operations are kept completely in memory whenever possible.

Server	Flask-0.10.1 (a light weight pure Python server), with our in-memory cache.
Client	urllib2 (python2.7 module for sending HTTP request)
Storage	SQLite3 (lightweight, easily portable database with nice python integration)

Github Repository: https://github.com/rchatterjee/KeyValueStore_cs739

2. Protocol: *REST* API

Our teams agreed to use the Representational State Transfer (*REST*, as described by Roy Fielding in his dissertation) API protocol as our communication standard because, it is language agnostic and has a well known standardized protocol which was very compatible with our project requirements. For example, *HTTP GET* closely resembles our *get()* operation, while *put()* and *delete()* closely resemble *HTTP PUT* and *DELETE* operations respectively.

HTTP Packet requests and responses conform to the W3 Standard, where packets have a header and a body, and the first line of the header is the status line and subsequent lines give more details such as date, server, content-length, content-type, etc. A blank line separates the headers from the body. The body of our request packets contained ASCII encoded key and value strings. When a URL encoded query string was included in the body of an *HTTP* request, the content type header is specified as: *Content-Type application/x-www-form-urlencoded*. Diagrams of *HTTP* Request and Response packets are displayed in the appendix 1. Note: our heartbeat request and empty OPTIONS request for the server *REST* API 'sitemap' are not part of the multiple group configuration; they are unique to our server.

We decided to use *HTTP* status codes to denote success or error responses.
For example, the status line: "*HTTP/1.1 200 OK*" always means success.

1. 200 Success. GET returns a value, while *PUT* & *DELETE* return the old_value.
2. 201 Created. A new key-value pair was created with *PUT*.
3. 404 Not found, no value is available for the client.
4. 500 Error. There was some kind of internal server error.

Our *REST* API implementation are outlined in the Table 1.

Function	Procedure to Send	Response from server	Response Code
	HTTP OPTIONS request (empty)	Returns a JSON object describing all available HTTP Request operations, which effectively acts as a sitemap for the API	200: Ok

	HTTP OPTIONS request
	e.g. <server-ip:port>/?heartbeat=1
		JSON encoded HTTP response body:
		{
		"heartbeat": "I am alive!!!"
		}
			200: Ok
			500: Error

kv739_init(key)	HTTP GET request (no parameters) URI: <server-ip:port>/ e.g, seclab8.cs.wisc.edu:5000/	HTTP Response with an empty body	200: Server Up

kv739_get(key)	HTTP GET request with query string:
	URI:
	<server-ip:port>/?key=<key name>
	
	e.g.
	eclab8.cs.wisc.edu:5000?key='foo'
		JSON encoded HTTP response body:
		{
		"value": <actual-value>,
		"errors": [err-string1, err-string2...]
		}
			200: Key Found
			404: Key not Found
			500: Error

kv739_put(key, value)	HTTP PUT request with a url encoded query string body:
	URI: <server-ip:port>/
	body:
	key=<actual-key>&value=<actual-value>
		JSON encoded HTTP response body:
		{
		"old_value": <old-value>,
		"errors": [err-string1, err-string2...]
		}
		old_value is defined only if status_code is 200.
			200: Key Found, Replaced the old Value
			201: Key not Found, created new (key,value)
			500: Error

kv739_delete(key)	HTTP PUT request with a url encoded query string body:
	URI: <server-ip:port>/
	body: key=<actual-key>
		JSON encoded HTTP response body:
		{
		"old_value": <old-value>,
		"errors": [err-string1, err-string2...]
		}
		old_value is defined only if status_code is 200.
			200: Key found if old_value is returned, delete succeeded.
			500: Error

Table 1: Client Library Functions (vs Underlying Protocol & HTTP Packets)

Although requests were always simple URL encoded key value pairs, *JSON* was used for responses. This is because most languages can easily translate *JSON* into a vanilla object. Also, if an error is present, we provided an ***optional*** error specification where errors were always expected with the "errors" key, and the 'errors' body is a *JSON* array of error description strings: {"errors":["error string1", "error string2"]}.

3. Test Design

3.1 Correctness Tests

Test driven development was used as our methodology for building and implementing our features and we used python's *unittest* library to validate our assertions. As each part of the application was built, corresponding tests were created to confirm the various functions worked as expected.

Before releasing our client library we built unit tests and independently validated the database, cache, server and client operations. The following correctness tests were run on the 3 setup configurations to confirm that the server-client pairs behaved according to our agreed upon specifications and the project requirements:

Setup Type 1: our tests, our client library, our server.
Setup Type 2: our tests, other team's library, our server.
Setup Type 3: other team's tests, other team's client library, our server.

- get() a valid key, confirm success
- get() a blank key, confirm error
- get() a key with size>128, confirm error
- get() a key with bad char "[", "]", confirm error
- delete() a valid key, confirm success
- delete() blank key, confirm error
- delete() key size>128, confirm error
- delete() bad char "[", "]", confirm error
- get() vs a deleted key to confirm it is gone
- put() a valid key-value pairm confirm success.
- put() a blank key, confirm error
- put() with a key size>128, confirm error
- put() bad char "[", "]", confirm error
- put() a value >2048, confirm error
- get() vs a put() to confirm the value returned matches the value sent by put()
- Tests to confirm all correct *REST* API status codes match corresponding operations
- Tests to confirm all correct values are returned by the client library: 0, 1, -1
- get(), put(), and delete() tests were performed in a loop with random strings to confirm correctness in many varying situations

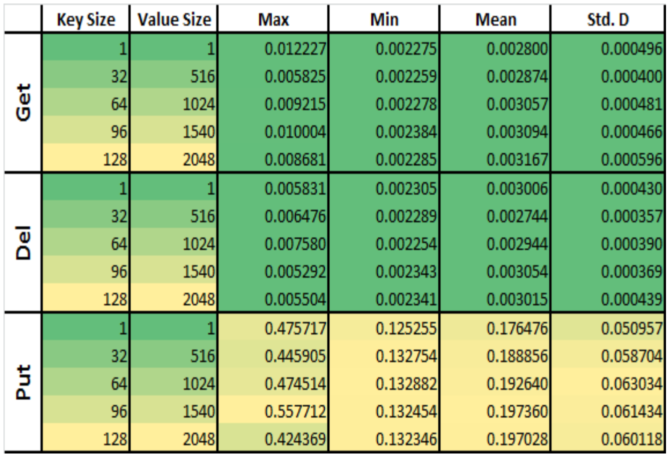
Some effort was required to validate all test cases, but eventually after communicating with the other teams we were able to validate all test cases with the three setup configurations above. This ensured that the agreed upon protocol was respected by all teams.

3.2 Performance Tests

Performance test was performed to measure latency and throughput.
Test Machine Specification: Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz (8 core)

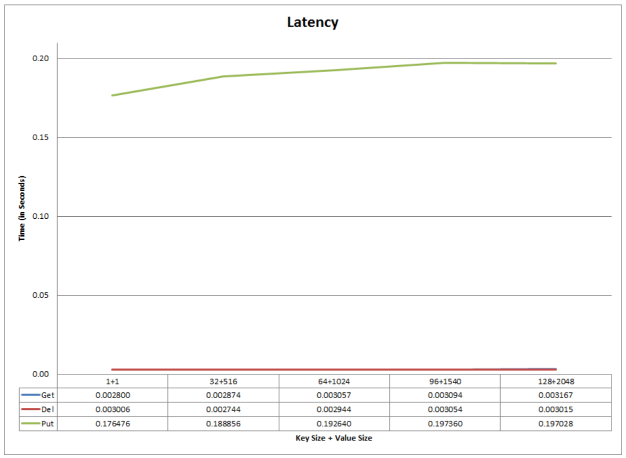
3.2.1 Latency

40,000 request of of put, get and delete each using our client library with out server were issued, the key size and value size was varied to measure the impact of key-value size on latency. Latency was measured for each individual operation. The results are summarized in figure 2-5. (Please see the Appendix for better resolution and more results.)



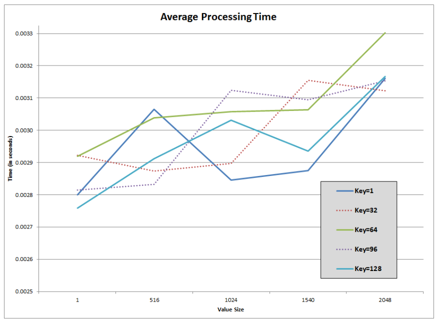
	Key Size	Value Size	Max	Min	Mean	Std. D
Get	1	1	0.012227	0.002275	0.002800	0.000496
	32	516	0.005825	0.002259	0.002874	0.000400
	64	1024	0.009215	0.002278	0.003057	0.000481
	96	1540	0.010004	0.002384	0.003094	0.000466
	128	2048	0.008681	0.002285	0.003167	0.000596
Del	1	1	0.005831	0.002305	0.003006	0.000430
	32	516	0.006476	0.002289	0.002744	0.000357
	64	1024	0.007580	0.002254	0.002944	0.000390
	96	1540	0.005292	0.002343	0.003054	0.000369
	128	2048	0.005504	0.002341	0.003015	0.000439
Put	1	1	0.475717	0.125255	0.176476	0.050957
	32	516	0.445905	0.132754	0.188856	0.058704
	64	1024	0.474514	0.132882	0.192640	0.063034
	96	1540	0.557712	0.132454	0.197360	0.061434
	128	2048	0.424369	0.132346	0.197028	0.060118

Figure 2: Latency statistics for different key and value sizes.

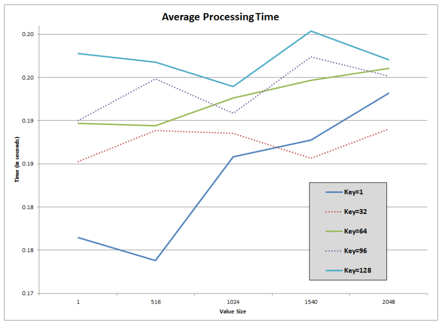


	1+1	32+516	64+1024	96+1540	128+2048
Get	0.002800	0.002874	0.003057	0.003094	0.003167
Del	0.003006	0.002744	0.002944	0.003054	0.003015
Put	0.176476	0.188856	0.192640	0.197360	0.197028

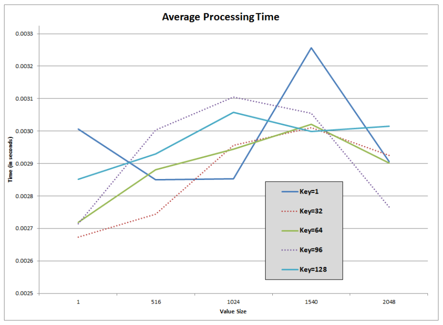
Figure 3: Latency for different key and value sizes.



Get

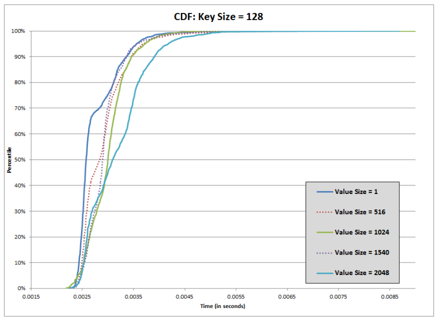


Put

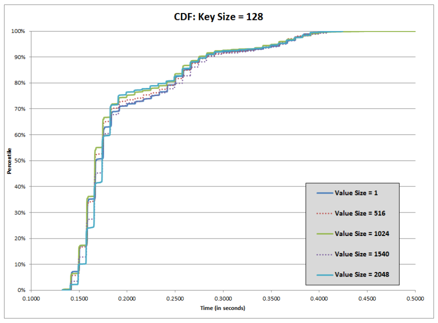


Delete

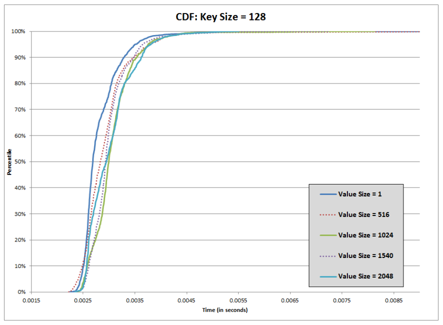
Figure 4: Average latency



Get



Put

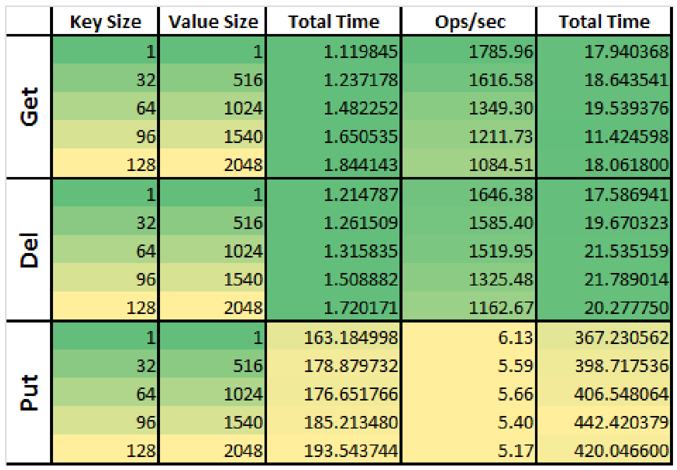


Delete

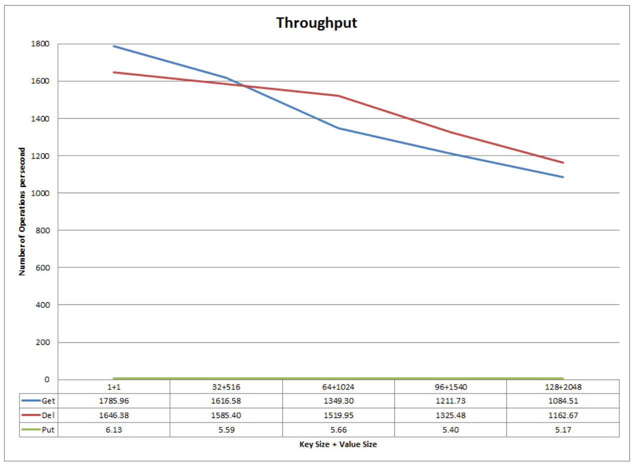
Figure 5: The Cumulative Distribution function

3.2.2 Throughput

A multi-thread pool sent 2000 get, put and delete requests to our server. The key-size and value-size was varied to measure performance differences with varying key sizes. The results are summarized in the figure 6. (Please see the Appendix for better resolution and more results.)



	Key Size	Value Size	Total Time	Ops/sec	Total Time
Get	1	1	1.119845	1785.96	17.940368
	32	516	1.237178	1616.58	18.643541
	64	1024	1.482252	1349.30	19.539376
	96	1540	1.650535	1211.73	11.424598
	128	2048	1.844143	1084.51	18.061800
Del	1	1	1.214787	1646.38	17.586941
	32	516	1.261509	1585.40	19.670323
	64	1024	1.315835	1519.95	21.535159
	96	1540	1.508882	1325.48	21.789014
	128	2048	1.720171	1162.67	20.277750
Put	1	1	163.184998	6.13	367.230562
	32	516	178.879732	5.59	398.717536
	64	1024	176.651766	5.66	406.548064
	96	1540	185.213480	5.40	442.420379
	128	2048	193.543744	5.17	420.046600



	1+1	32+516	64+1024	96+1540	128+2048
Get	1785.96	1616.58	1349.30	1211.73	1084.51
Del	1646.38	1585.40	1519.95	1325.48	1162.67
Put	6.13	5.59	5.66	5.40	5.17

Figure 6: throughput for different key and value sizes.

3.2.3 Observations

- Our *put* is significantly slower than *get* and *delete* because our database is indexed on keys which makes the *get* faster (as our assumption of read heavy load) at the cost of slower *put* and for persistence *put* and *delete* has to go through synchronous disk I/O.
- The server was the driving factor in terms of performance, and the client libraries for each respective group was too thin to make a significant impact. The performance difference between other groups' tests and our tests vs our own server were negligible, so we do not present these results.
- The correctness test script for all groups passed with small final reconciliation on the protocol amongst the groups.
- Introduction of the in-memory cache on the server improved the performance of get() since additional hits to the SQLite3 database was avoided if the key was already present in the cache.
- put() operations could be potentially improved with batching multiple operations. But we decided not to implement it to avoid complexity for this project.
- The test machine used to host our server and SQLite3 database was significantly slower than the other groups we worked with, evident with the RTT stats below:

Server	Min (in ms)	Average (in ms)	Max (in ms)	M. Dev (in ms)
Ours	0.286	0.338	0.483	0.065
Team 1	0.192	0.232	0.294	0.031
Team 2	0.201	0.217	0.253	0.018

Conclusion

Coming up with a common protocol for three groups was difficult and took a few iterations. Some protocol items that seemed like minor details during the initial discussion later became points of contention since they were causing our respective tests to fail vs the other group's servers. We can boil down the rest of our observations into a few points:

- Our implementation favors consistency and durability (and simplicity) over performance.
- Our solution is durable since *put()* operations are atomic and immediately written to persistent storage (SQLite3).
- The *delete()* operations are durable as well since they are immediately carried out.
- Also, the *REST* API is stateless which does remove some performance burden from the server (the only downside of a server restart is a cold cache).
- The *REST* API is language agnostic as well, so it can enable heterogeneous systems to interact seamlessly.
- We have implemented client-side and server-side logger for troubleshooting and maintaining a chronological list of operations performed. In the even of the database failure, logs can be consolidated to recover inconsistent data. A separate application would be needed to consolidate the logs.
- Our system is just a single node, so in the event of a hard-disk crash, everything could be lost.
- Our server is fault tolerant in the sense that if it crashed at any time, only the cached data is lost but since data is always persisted in the database before caching, there is no loss of data.
- Flask automatically restarts the server for us, and heartbeat requests (as specified in our protocol) can confirm the server is alive, so minor failures can be automatically detected and handled.
- Also, we know our implementation is very correct since it works with multiple client and test configurations, provided by us and the groups we cooperated with.
- We also made our *REST* API relatively easy to use by providing an effective 'site map' with an empty OPTIONS request (which returns a JSON configured object that specifies all available server operations).