Development Log:

1. This is a very busy week, so I have very limited time. I will try to implement in Python instead of C++.
2. Python has a very efficient data type for key/value storing, the dict. This will be the backbone of the data store. I’m working with Python 3.4.
3. To validate the assumption in 2) I wrote a script to time the access times to insert, read and delete in a dict. All are in the single digit micro seconds range. This leaves plenty of room to wrap it around a daemon structure and still meet the millisecond requirements. I tested inserting and reading 10,000,000 elements without any issues.
4. Again, because of time constraints, two design decisions are made
   1. Asynchronous processing on a single thread of requests. This is fairly efficient since it will handle bursts, but will underutilize the multiple cores on my computer
   2. UDP wrapper for the store, running “service-like” (no code included to register as an actual service)
5. store.py contains the key/value store implementation (based on dict)
6. storeServer.py contains the service
7. storeClient.py is the API to be called in order to use the key/value store
8. testSpeed.py tests the speed of the store object locally, and of the storeClient on a single persistent connection (ideal scenario, baseline used to meet designed requirements as concurrency wasn’t specified). This performance greatly exceeds the required performance:

\*\*\* Testing performance on 1000000 elements \*\*\*

Data Structure and Store Testing (no network or concurrency)

Time Per Hash (s): 1.0028639078140259e-05

Time Per Allocaiton (s): 8.188189111651026e-07

Time Per Read (s): 5.156078726330191e-07

Time Per Delete (s): 6.982233353998491e-07

Testing API service, persistent connection, no concurrency

Time Per Insert API (s): 6.921970661365802e-05

90%, 95%, 99%: 8.10623168945e-05 9.3936920166e-05 0.00012993812561

Time Per Get API (s): 6.641984715104833e-05

90%, 95%, 99%: 7.60555267334e-05 8.79764556885e-05 0.000121116638184

Time Per Delete API (s): 6.681732589058334e-05

90%, 95%, 99%: 7.70092010498e-05 8.89301300049e-05 0.000121116638184

1. Results from 8) show access times for Read/Write/Delete in the hundreds of micro seconds range (at least an order of magnitude lower than specified)
2. testCli.py is a command line utility to trigger a single connection request, with network connection, and times the response time of the actual requrest (without the network connection establishment). It is use in conjunction with xargs to produce a log, an example would to run 16 concurrent clients would be:  
     
   cat 2column.txt | xargs -n 2 -P 16 python testCli.py put > putlatency16.log
3. getStatsCli.py is a command line utility to consolidate the log files, and provides number of iterations, mean access time, variance of access time, top 90th percentile, to 95th percentile, and top 99th percentile. Multiple client scenarios were not part of the requirement, but are a critical part for the evaluation of a database solution. The following table present the results for 100,000 iterations. Somewhere in between the 4 to 8 client load (again, without

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Test Case** | **Task** | **Concurrent Clients** | **Mean** | **Std Dev** | **90%** | **95%** | **99%** |
| testlogs/deletelatency1.log | delete | 1 | 0.0004217 | 0.0000772 | 0.0004630 | 0.0004830 | 0.0005560 |
| testlogs/deletelatency2.log | delete | 2 | 0.0004227 | 0.0001600 | 0.0005398 | 0.0005989 | 0.0007620 |
| testlogs/deletelatency8.log | delete | 8 | 0.0010220 | 0.0021145 | 0.0020059 | 0.0038778 | 0.0099761 |
| testlogs/deletelatency16.log | delete | 16 | 0.0013105 | 0.0042314 | 0.0021583 | 0.0040525 | 0.0117867 |
| testlogs/getlatency1.log | get | 1 | 0.0004191 | 0.0001251 | 0.0004690 | 0.0005052 | 0.0006132 |
| testlogs/getlatency2.log | get | 2 | 0.0004499 | 0.0002117 | 0.0005890 | 0.0006511 | 0.0008350 |
| testlogs/getlatency4.log | get | 4 | 0.0005247 | 0.0005122 | 0.0007720 | 0.0009868 | 0.0018781 |
| testlogs/getlatency8.log | get | 8 | 0.0010511 | 0.0021436 | 0.0020103 | 0.0035252 | 0.0095899 |
| testlogs/getlatency16.log | get | 16 | 0.0015068 | 0.0048307 | 0.0025002 | 0.0044708 | 0.0161799 |
| testlogs/getlatency32.log | get | 32 | 0.0019486 | 0.0090673 | 0.0023389 | 0.0044817 | 0.0261209 |
| testlogs/putlatency1.log | put | 1 | 0.0004242 | 0.0001319 | 0.0004649 | 0.0004900 | 0.0005629 |
| testlogs/putlatency2.log | put | 2 | 0.0004715 | 0.0001981 | 0.0006249 | 0.0006809 | 0.0008719 |
| testlogs/putlatency4.log | put | 4 | 0.0004933 | 0.0003310 | 0.0007412 | 0.0008991 | 0.0012783 |
| testlogs/putlatency8.log | put | 8 | 0.0010887 | 0.0020527 | 0.0022005 | 0.0041333 | 0.0099746 |
| testlogs/putlatency16.log | put | 16 | 0.0015460 | 0.0052653 | 0.0024703 | 0.0046509 | 0.0182599 |
| testlogs/putlatency32.log | put | 32 | 0.0022016 | 0.0095384 | 0.0026214 | 0.0054360 | 0.0359478 |