Michael Laucella
Multicore programming
Lab 3

Analytics Report

**note - all graphs are located in the analyze folder just trail down to the respective folders
- also note that the labels haven't been switched so seconds refers to milliseconds in all the graphs (it would take way too long to re-generate all those graphs)

For the mean, median, max, min: you can run the program make sure to compile with the ST=1 flag, but to make life easier I also had python analyze my output and put all the stats from every run in one file stats.txt in the analyze folder. The order is the same as the file array in stats.py

The top sections are mostly speculative below the tables is an attempt at a more numerical understanding

From analyzing this file we notice first the lowest average is from the regular queue implementations. You would expect that the throughput, the amount it can serve in a period of time is lower than the higher threaded times, however for 1000 sessions with 100 requests each the finish time was 37s on 1 thread compared to 35s on 32 barely any difference in the overall client time. Potential reasoning behind this could be contention on the system, inferior algorithms for instance the queue class being used here has an enqueue and dequeue lock and also a semaphore which could be removed if a sentinel node model was used instead. Building on that point the lock free structure although having horrible performance was getting better and better as the threads increased both the client and server times were decreasing at a noticeable rate, likely due to the fact that each thread can perform some operation and is always either grabbing from, inserting into, or fixing the queue and CAS operations are themselves cheaper than mutex operations (in terms of time). This shows the impact the queue itself has on the run times observed.

. Also form observing the graphs there are many noticeable patterns within them where we can see how some outside influence such as other users on the machine may have drained performance, this can be observed in many of the graphs. Checking the machine with htop I found about 4 of the threads were running on 100% and more so for the server running the httperf. Another interesting pattern when viewing the stats.txt file the global locks for the most part offer very little to the time increase than the reader write locks. This leads me to think that the kv store itself doesn't cause the majority of the time differences, the queue is obviously having an impact as can be seen by viewing the lock frees graphs, I'm assuming though that also having to run more concurrent MD5's (which i suspect are making use of GPU's hence leading to minimal effect on timing especially as its IO it wouldn't affect the program's CPU timings) or sending and receiving connections may have played a role in the time shifts as well.

As for the client side there’s also the reality that the connections themselves are still bottlenecked at the very beginning they come in through the main thread meaning they face a period of serialization, and on top of that there is also the fact that they are highly reliant on the transfer speed of the network and the contention that may be on it. The system may take a long time to transfer all of those messages at any one time, there are many messages flooding the system trying to come in and go out, better performance would likely be gained by running two 8 thread servers and having them connect to two different servers running httperf with 500 sessions each. Then by one running 32 and one sending the 1000 sessions

--for times sake im excluding lock free queue with global lock
--server stats

Regular Queue, with global lock KV

Threads	Average (ms)	Max (ms)	Min (ms)	Median (ms)
1	.36803	3.591	.081	.258
2	.37859	3.464	.089	.269
4	.37296	4.44	.087	.263
8	.38436	3.774	.091	.263
16	.37145	3.939	.089	.259
32	.36382	13.875	.093	.254

Regular Queue, with reader/writer KV

Threads	Average (ms)	Max (ms)	Min (ms)	Median (ms)
1	.36177	3.467	.08	.252
2	.38399	3.637	.085	.266
4	.37270	6.494	.089	.262
8	.36873	3.769	.09	.256
16	.37250	3.583	.09	.256
32	.36282	6.043	.096	.254

--note that lock free queue has a 250ms sleep if queue is empty which may have impacted the max times for all the queues as they all see to be around the same time (this is further proved by the starting information in the graph of 32 thread lock free provided below, we see the very start runs the approx the max 250 then drops after the queue starts to fill up.

Lock Free Queue, with reader/writer KV

Threads	Average (ms)	Max (ms)	Min (ms)	Median (ms)
1	2.83668	258.311	.109	.25
2	1.50266	251.141	.101	.259
4	0.92605	212.703	.094	.253
8	0.66036	212.94	.091	.263
16	0.53205	212.76	.106	.247
32	0.45768	243.333	.075	.239

--client stats

Regular Queue, with reader/writer KV

Threads	Time (s)	Mean (ms)	Max (ms)	Min (ms)	Median (ms)
1	36.641	36.7	51.7	21.7	36.5
2	38.196	38.2	54.4	24.1	38.5
4	37.015	37	54.5	23.3	36.5
8	36.522	36.5	53.7	22.6	36.5
16	36.868	36.9	54.3	24.3	36.5
32	35.864	35.9	52.3	23.6	35.5

Lock Free Queue, with reader/writer KV

Threads	Time (s)	Mean (ms)	Max (ms)	Min (ms)	Median (ms)

1	286.709	286.7	303.1	76.6	286.5
2	151.960	152	293.9	27.4	148.5
4	93.718	93.7	245.5	28.8	82.5
8	66.875	66.9	240.8	27.3	60.5
16	46.415	46.4	282.5	25.9	42.5
32	53.915	53.9	240.0	26.5	48.5

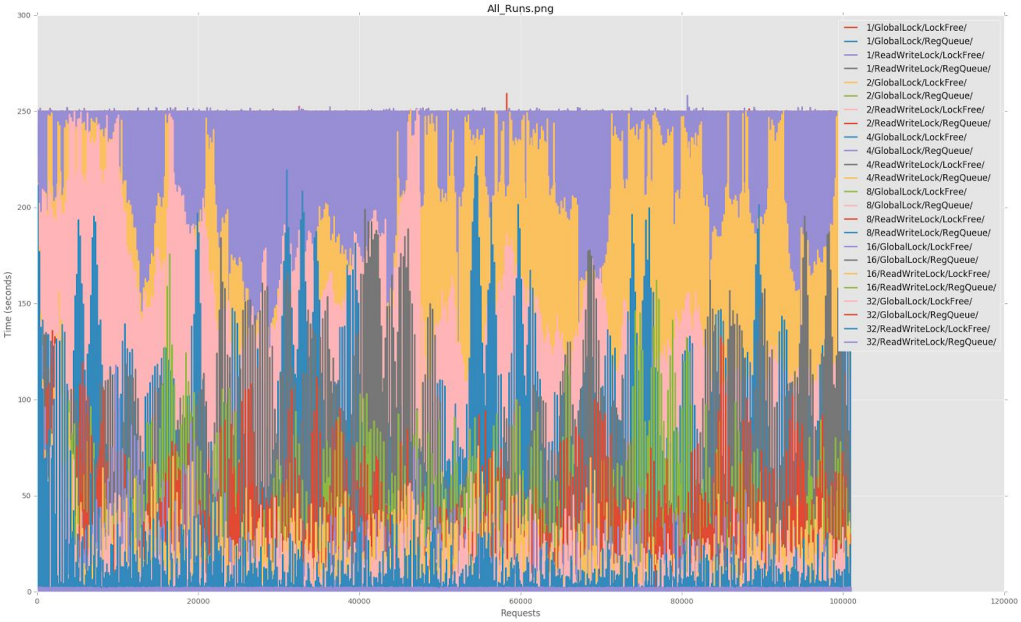
(analysis done using 2 threads regular queue)
Viewing the client tables shows that the average time is in milliseconds, so why is the total time so high? Well we see that each connection contains 100 requests, so 38.2ms/100 = .382 ms per request, Reviewing replies/s we see an average of 2620.5 which means that each reply took approximately 1/2620 of a second or .00038 seconds approx .38 ms which actually corresponds to the mean completion time per request of the 2 thread regular queue which is approx .38 ms. So there actually is a good amount of correspondence between the server and client with 2 threads. And from viewing all the connection means we see a general consistency one which reflects that of the server time per request we even see this relationship with the lock free numbers when viewing the tables above. This doesn't mean there is no network transportation effects because the server time taken also includes the sending of the data to the client. However it does show the time between accept and queue insertion has little effect.

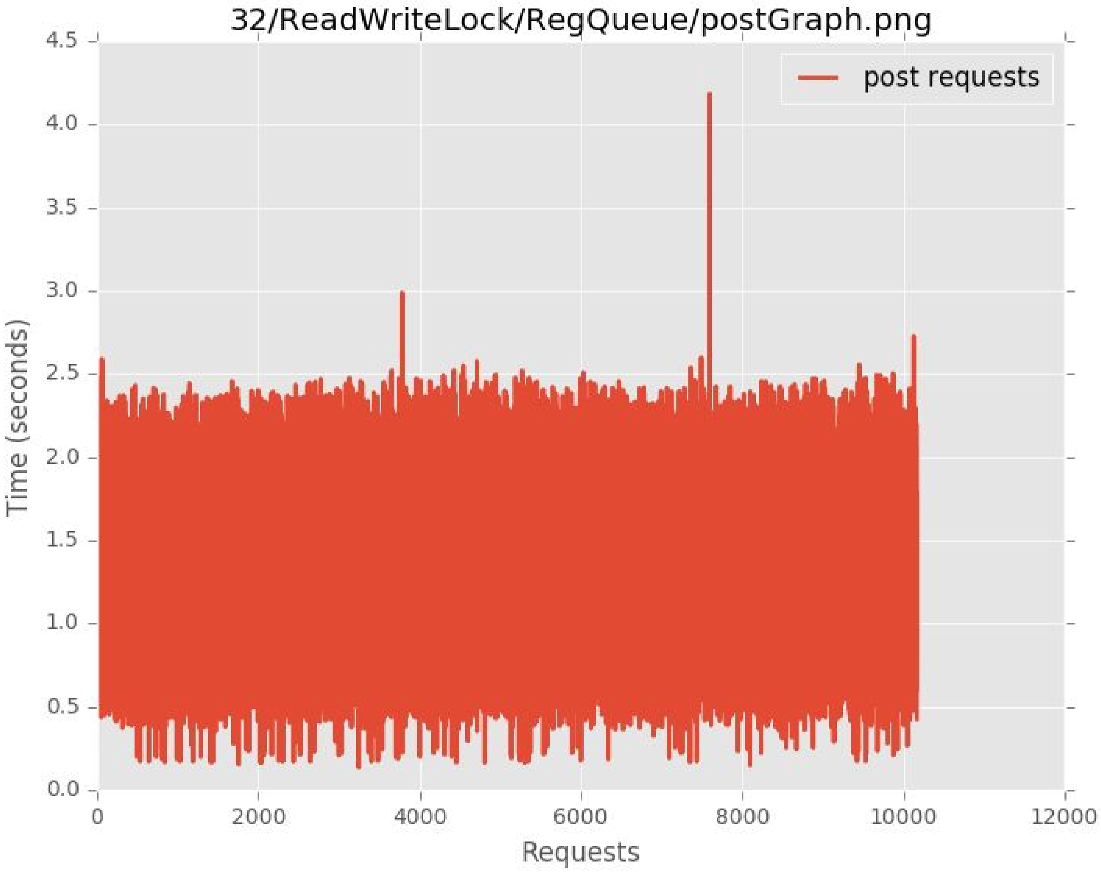
Below is a small subset of the graphs all of which are available through the analyze folder, These are specifically analysis graphs from the 32 thread times of lock free and regular queue with read write lock KV.

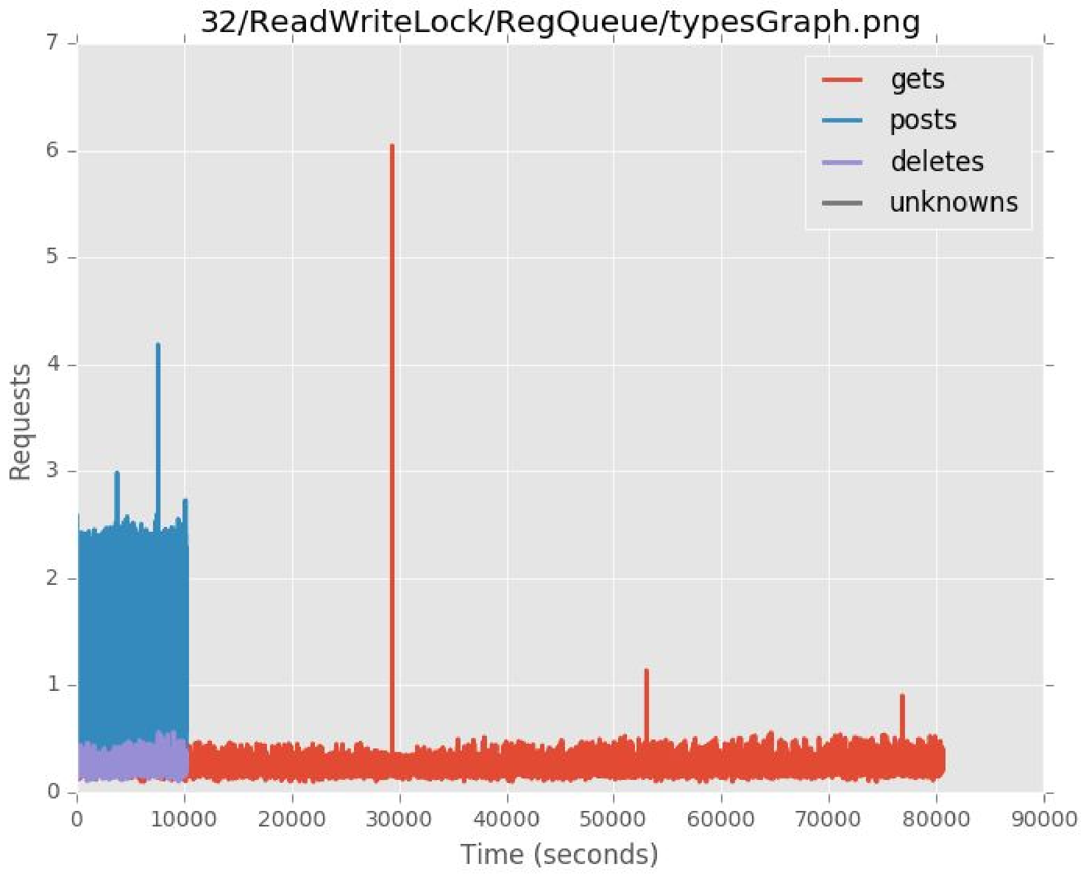
We see that posts/deletes as would be expected are the most expensive computations in both parts because they are writes which serialize the section.

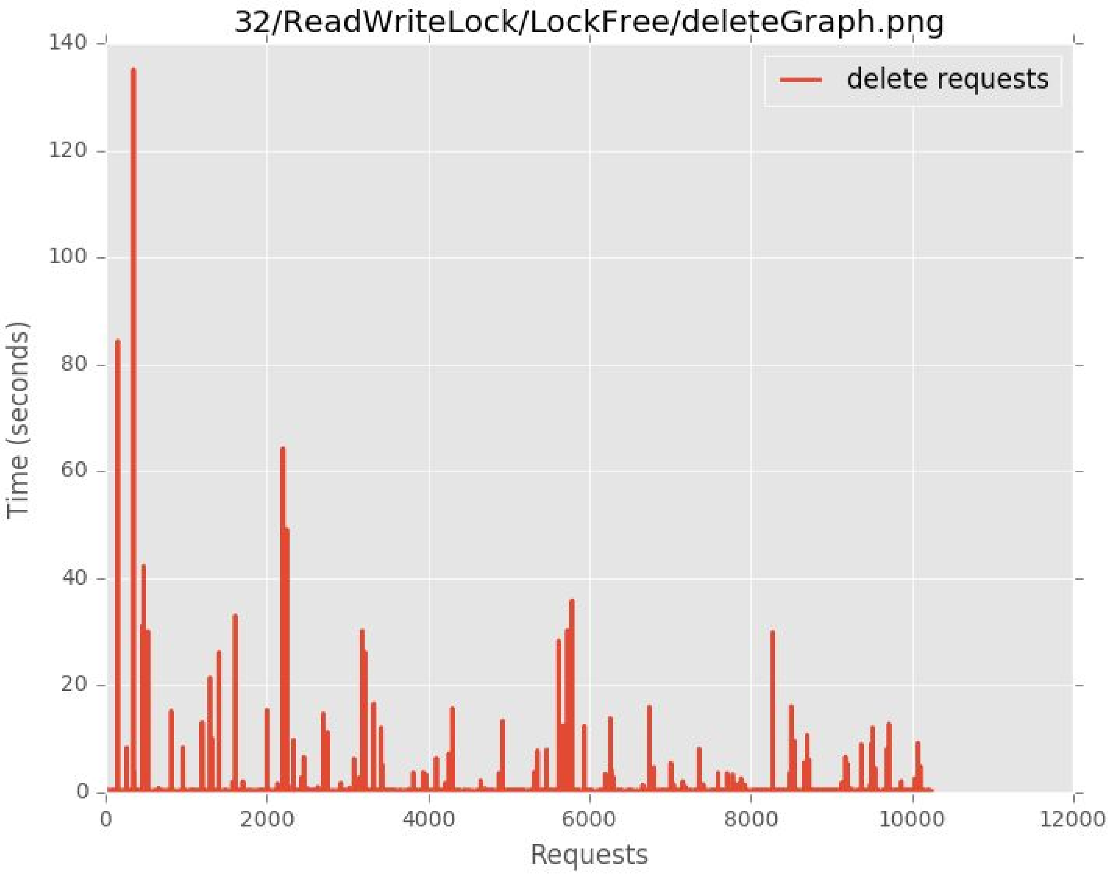
Also below is the graph from every run (better viewed from the folder), I would say it behaves as would be expected all the regular queue implementations cling pretty low on the graph a point of interest however is noting the decrease of time from the lock free structures as threads increase (the blue line which starts reaching the regular queues but has some high spikes) If the number of threads continued to increase the trend would lead to the idea that it will continue to decrease whereas the regular queue implementation stayed mostly stagnant.

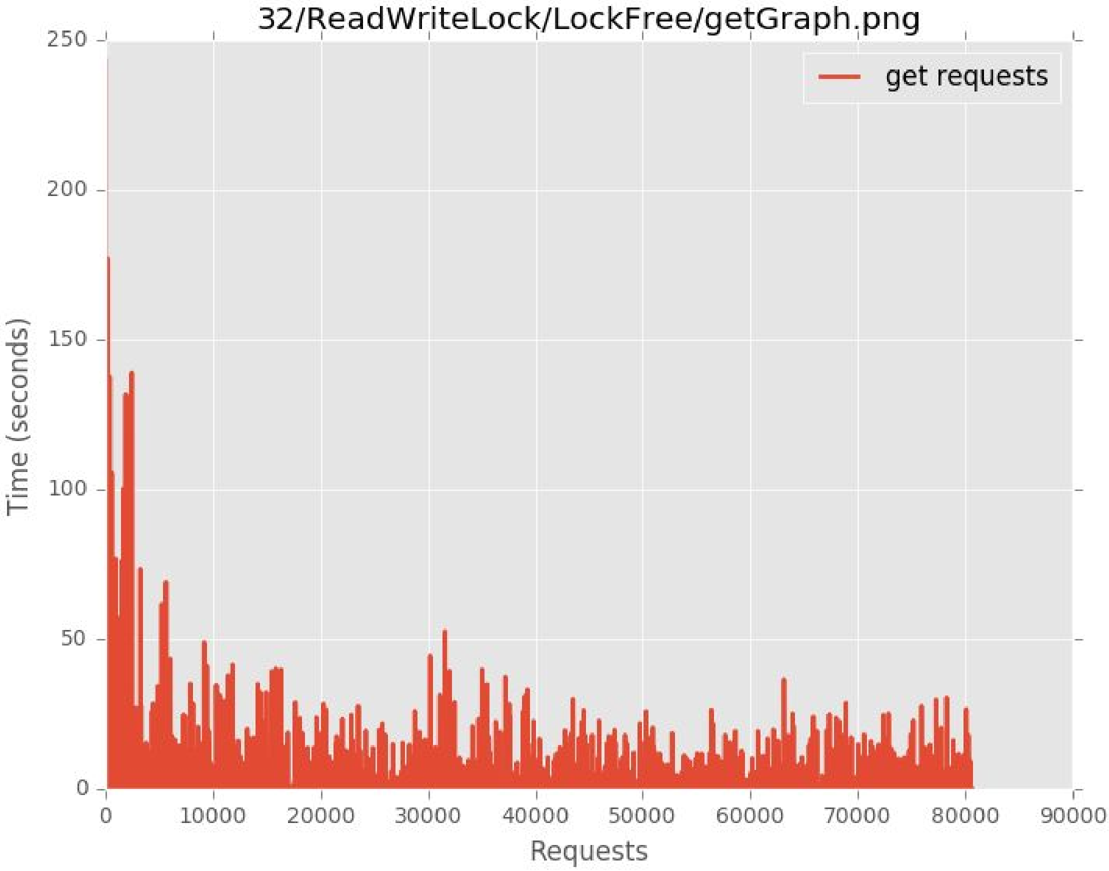
--once again ignore that time is set to seconds on the y axis it really is plotting milliseconds, also notice that two have the wrong axis labels (requests should be on x axis, times on y axis) the information is still posted to the correct axis just bad labels

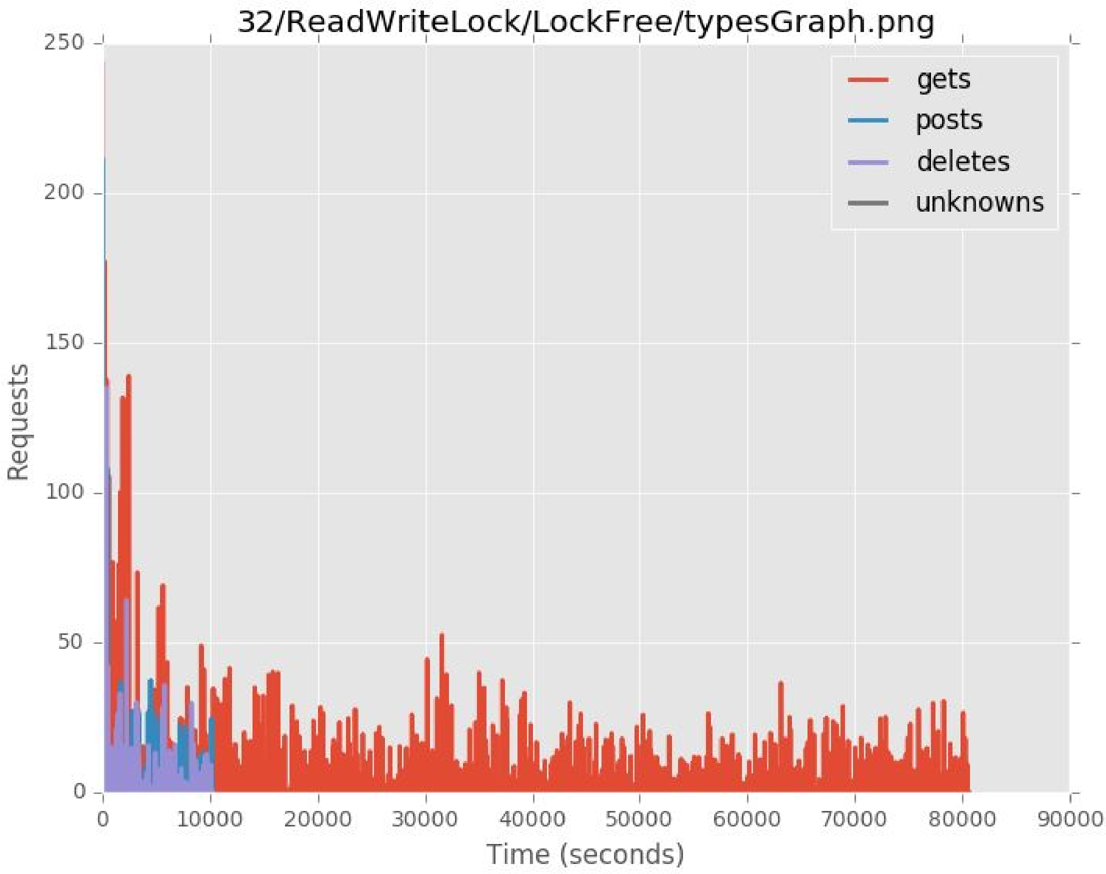

All_Runs.png
Time (seconds)
Requests
0
50
100
150
200
250
300
0
20000
40000
60000
80000
100000
120000
1/GlobalLock/LockFree/
1/GlobalLock/RegQueue/
1/ReadWriteLock/LockFree/
1/ReadWriteLock/RegQueue/
2/GlobalLock/LockFree/
2/GlobalLock/RegQueue/
2/ReadWriteLock/LockFree/
2/ReadWriteLock/RegQueue/
4/GlobalLock/LockFree/
4/GlobalLock/RegQueue/
4/ReadWriteLock/LockFree/
4/ReadWriteLock/RegQueue/
8/GlobalLock/LockFree/
8/GlobalLock/RegQueue/
8/ReadWriteLock/LockFree/
8/ReadWriteLock/RegQueue/
16/GlobalLock/LockFree/
16/GlobalLock/RegQueue/
16/ReadWriteLock/LockFree/
16/ReadWriteLock/RegQueue/
32/GlobalLock/LockFree/
32/GlobalLock/RegQueue/
32/ReadWriteLock/LockFree/
32/ReadWriteLock/RegQueue/

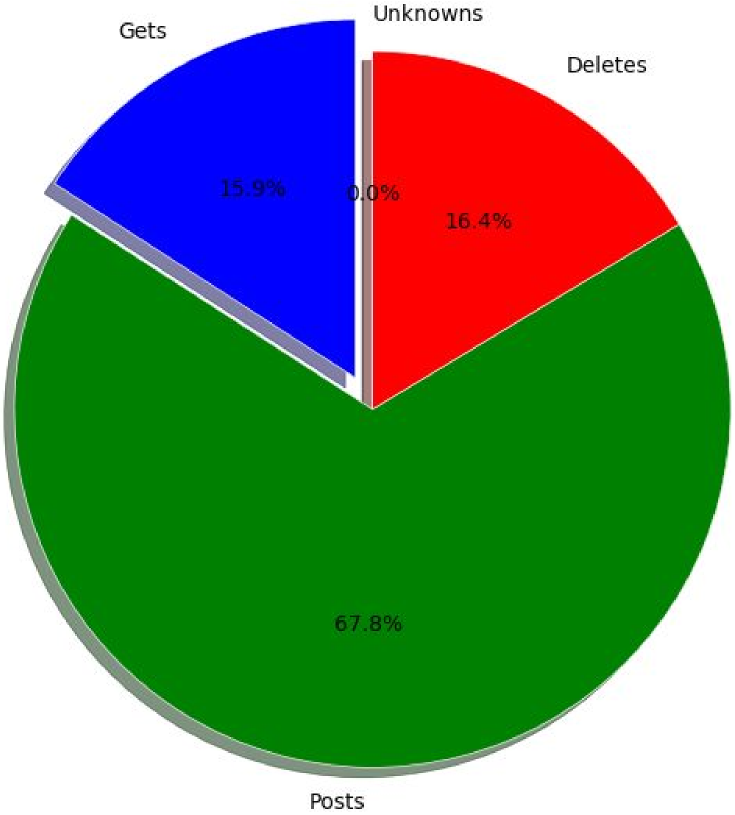

32/ReadWriteLock/RegQueue/postGraph.png
post requests
Time (seconds)
Requests
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
0
2000
4000
6000
8000
10000
12000

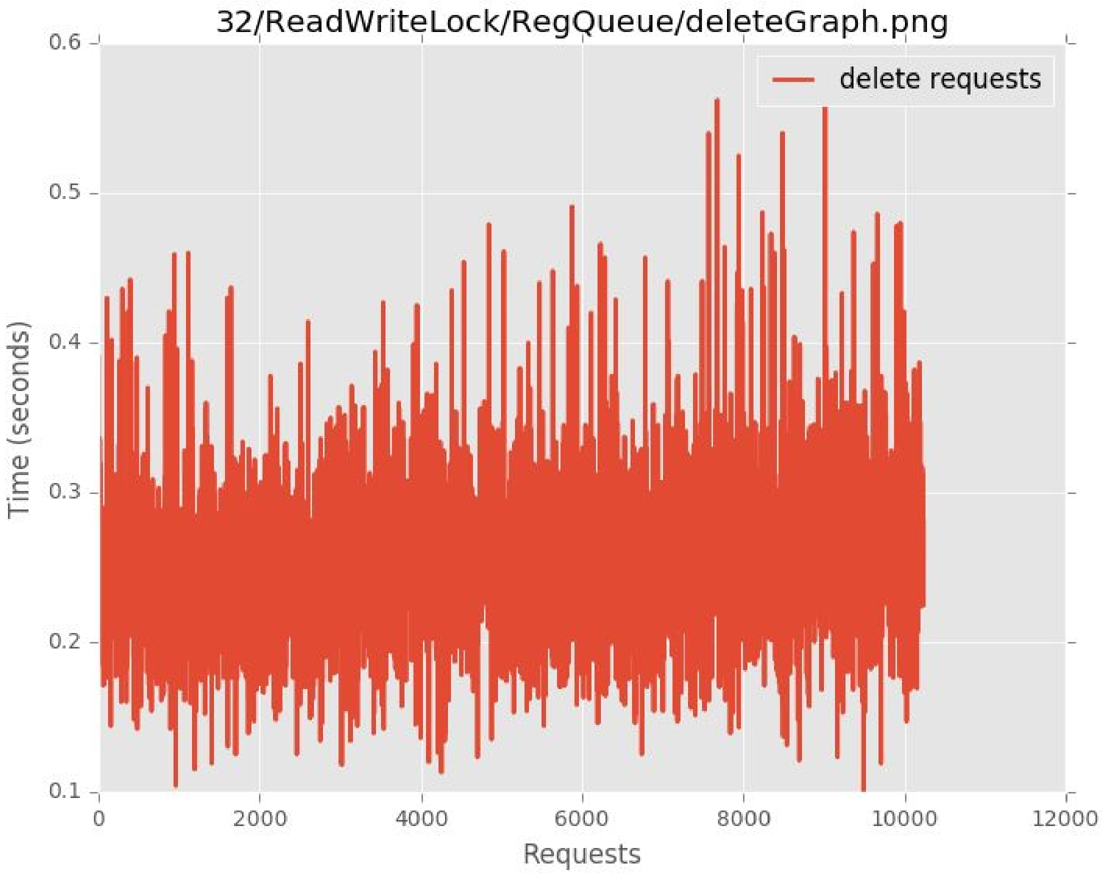

32/ReadWriteLock/RegQueue/typesGraph.png
gets
posts
deletes
unknowns
Requests
0
1
2
3
4
5
6
7
Time (seconds)
0
10000
20000
30000
40000
50000
60000
70000
80000
90000

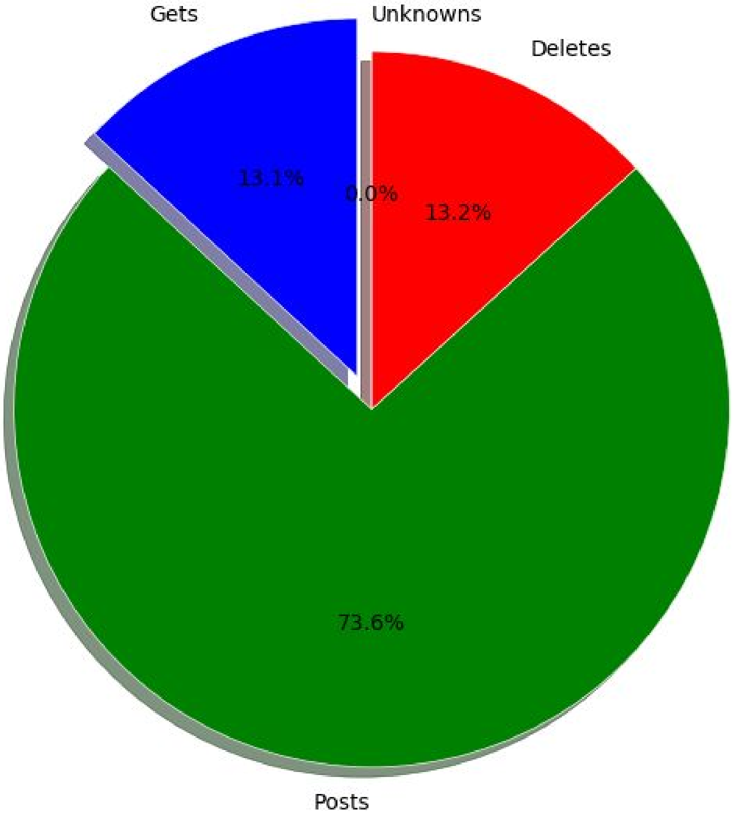

32/ReadWriteLock/LockFree/deleteGraph.png
delete requests
Time (seconds)
0
20
40
60
80
100
120
140
Requests
0
2000
4000
6000
8000
10000
12000


32/ReadWriteLock/LockFree/getGraph.png
get requests
Time (seconds)
0
50
100
150
200
250
Requests
0
10000
20000
30000
40000
50000
60000
70000
80000
90000


32/ReadWriteLock/LockFree/typesGraph.png
Requests
Time (seconds)
0
50
100
150
200
250
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
gets
posts
deletes
unknowns


Gets
Unknowns
Deletes
15.9%
0.0%
16.4%
67.8%
Posts


32/ReadWriteLock/RegQueue/deleteGraph.png
delete requests
Time (seconds)
0.6
0.5
0.4
0.3
0.2
0.1
0
2000
4000
6000
8000
10000
12000
Requests


Gets
Unknowns
Deletes
13.1%
0.0%
13.2%
73.6%
Posts