CAPS High Throughput Server Report

Overview

In this report I will be discussing the results of my throughput experiments using the multithreaded server and test client I developed according to the Assignment Specification and Message Board Protocol.

First, I must state that despite my best efforts, I was not able to complete the client-side test harness' functionality; though it does compile and partially run, either only one of its threads functions, or all poster threads only send one request then exit. Due to not being able to implement this part of the assignment in time, all the throughput experiments I have completed are using the Reference Client provided in the assessment materials.

I would like to stress however that a major reason for me struggling to finish the test harness is that I was told in an assignment support lecture by a lecturer who has now left the university that the client should calculate correct responses to every request it makes and maintain a mirror of the data structure that should be present in the server, thus the two can be compared after a test run and the functionality of the server can be verified. I only very recently found out that this is not a requirement listed in the assignment brief or anywhere on the mark scheme, and if I had not been told this by said lecturer, I believe I could have produced a fully functional test harness as without this requirement it is a much simpler task.

My server is fully developed and functional, running into no bugs (crashes, deadlock or livelock) during my testing on the PCs in Cantor 9341. **(I completed my tests on the machine with IP 10.72.84.62)**

Throughput experiments

Visible on the next page(s) is the raw data from the throughput experiments I completed. I varied the numbers of poster and reader threads from 1 to 3 and ran a set of stress-testing high throughput runs with 5 posters and 5 readers. My server survived every test and performed nearly on par with, sometimes faster than, the Reference Server provided in the assessment materials.

Below the data table are three graphs, labelled appropriately.

Fig. 1 shows the relationship between an increasing number of threads and a decreasing number of requests per thread per second. This disadvantageous relationship is a result of the use of mutexes: the more threads are actively manipulating the system's data structures, the greater portion of the time mutexes are locked and other threads have to wait. This is a great example of the diminishing returns of parallel programming - past a certain point of speedup it becomes incredibly difficult to see any more performance improvements when adding more compute units due to factors like these.

Fig. 2 outlines the difference in speed between poster and reader threads – there is almost none. This may vary per implementation of the solution but little to no difference between the two shows that both are nearly fully optimised, and certainly no part of the server is holding the rest back in terms of performance and response times.

Fig. 3 is an extension of Fig. 1 which displays the total number of requests per test run in relation to the requests per thread in the same run. It clearly demonstrates that, as the number of threads increases (from left to right) the overall performance of the system sees a large boost despite the fact that each individual thread is handling slightly less requests. This shows that while there are diminishing returns when adding threads, parallel programming is still a great way to boost performance.

How could I improve my code

Obviously, my project would be in a far better place if it had a functional client. If I were to remake the client, I would take the approach advised by the assignment brief and generate large amounts of random requests. The randomness would be designed such that, while readers generate some (maybe a large) portion of invalid read requests, they would also generate many valid requests. This would not only aid the test harness in stress testing the server at a greater throughput if so desired, due to the lack of slow-down from creating a mirror of the data, but would ensure that the server handles invalid requests correctly.

As for my server, one very scope-specific improvement I could make is to pre-allocate enough memory in the map of topics and topic vectors themselves so that during the 10 second runs, no extra memory would need to be allocated – this is a fairly slow process and could help improve the throughput significantly.

A more general and realistic improvement is to try and cut down on the use of mutexes where possible. While locking is a great method to ensure data does not become corrupted, it is slow. There are data structures that exist or can be designed which require few to no locks, depending on their application. If these were to be applied to the system, then I would expect to see a noticeable increase in performance and better parity between threads' throughputs.

Originally, I had planned to make an initial, 'lock-heavy' implementation and once it was fully implemented, work towards a solution using some of the data structures mentioned above. Unfortunately, due to time constraints this was not possible but would definitely be one of my highest priorities if presented with the challenge of optimising the multithreaded server further.

POST Threads	READ Threads	Total Threads	Runtime (sec)	Total POST Requests	POSTs/ Thread	Total READ Requests	READs/ Thread	Total Requests	Requests/ Thread	Requests/ Thread/Sec	Average Requests/Thread/Sec
1	1	2	10	211,220	211,220	221,431	221,431	432,651	216,326	21,633	
1	1	2	10	216,898	216,898	219,993	219,993	436,891	218,446	21,845	
1	1	2	10	209,475	209,475	210,984	210,984	420,459	210,230	21,023	21,180
1	1	2	10	206,246	206,246	211,030	211,030	417,276	208,638	20,864	
1	1	2	10	202,326	202,326	208,360	208,360	410,686	205,343	20,534	
1	2	3	10	176,021	176,021	368,590	184,295	544,611	181,537	18,154	
1	2	3	10	170,291	170,291	347,708	173,854	517,999	172,666	17,267	
1	2	3	10	177,610	177,610	363,751	181,876	541,361	180,454	18,045	17,546
1	2	3	10	166,756	166,756	342,169	171,085	508,925	169,642	16,964	
1	2	3	10	171,323	171,323	347,613	173,807	518,936	172,979	17,298	
1	3	4	10	143,187	143,187	464,710	154,903	607,897	151,974	15,197	
1	3	4	10	152,005	152,005	480,303	160,101	632,308	158,077	15,808	
1	3	4	10	144,308	144,308	471,157	157,052	615,465	153,866	15,387	15,517
1	3	4	10	149,076	149,076	463,445	154,482	612,521	153,130	15,313	
1	3	4	10	155,411	155,411	479,792	159,931	635,203	158,801	15,880	
2	1	3	10	357,906	178,953	188,646	188,646	546,552	182,184	18,218	
2	1	3	10	334,578	167,289	170,295	170,295	504,873	168,291	16,829	
2	1	3	10	348,227	174,114	117,181	117,181	465,408	155,136	15,514	17,138
2	1	3	10	344,141	172,071	180,808	180,808	524,949	174,983	17,498	
2	1	3	10	349,931	174,966	179,053	179,053	528,984	176,328	17,633	

2	2	4	10	311,622	155,811	302,717	151,359	614,339	153,585	15,358	
2	2	4	10	300,110	150,055	312,509	156,255	612,619	153,155	15,315	
2	2	4	10	304,984	152,492	316,560	158,280	621,544	155,386	15,539	15,339
2	2	4	10	306,692	153,346	321,514	160,757	628,206	157,052	15,705	
2	2	4	10	279,022	139,511	312,003	156,002	591,025	147,756	14,776	
2	3	5	10	259,324	129,662	448,454	149,485	707,778	141,556	14,156	
2	3	5	10	272,853	136,427	400,024	133,341	672,877	134,575	13,458	
2	3	5	10	261,751	130,876	414,347	138,116	676,098	135,220	13,522	13,599
2	3	5	10	261,006	130,503	394,507	131,502	655,513	131,103	13,110	
2	3	5	10	270,857	135,429	416,666	138,889	687,523	137,505	13,750	
3	1	4	10	453,968	151,323	152,874	152,874	606,842	151,711	15,171	
3	1	4	10	464,720	154,907	161,882	161,882	626,602	156,651	15,665	
3	1	4	10	442,157	147,386	163,350	163,350	605,507	151,377	15,138	15,298
3	1	4	10	439,011	146,337	159,259	159,259	598,270	149,568	14,957	
3	1	4	10	460,980	153,660	161,351	161,351	622,331	155,583	15,558	
3	2	5	10	421,359	140,453	286,384	143,192	707,743	141,549	14,155	
3	2	5	10	416,510	138,837	278,935	139,468	695,445	139,089	13,909	
3	2	5	10	411,924	137,308	292,024	146,012	703,948	140,790	14,079	13,909
3	2	5	10	381,993	127,331	284,364	142,182	666,357	133,271	13,327	
3	2	5	10	418,869	139,623	284,868	142,434	703,737	140,747	14,075	
3	3	6	10	393,770	131,257	392,759	130,920	786,529	131,088	13,109	
3	3	6	10	370,696	123,565	408,217	136,072	778,913	129,819	12,982	12,835
3	3	6	10	382,176	127,392	399,016	133,005	781,192	130,199	13,020	

3	3	6	10	359,887	119,962	370,634	123,545	730,521	121,754	12,175	
3	3	6	10	368,526	122,842	404,792	134,931	773,318	128,886	12,889	
5	5	10	10	496,903	99,381	442,170	88,434	939,073	93,907	9,391	
5	5	10	10	492,779	98,556	550,353	110,071	1,043,132	104,313	10,431	
5	5	10	10	517,763	103,553	553,507	110,701	1,071,270	107,127	10,713	10,103
5	5	10	10	536,538	107,308	510,479	102,096	1,047,017	104,702	10,470	
5	5	10	10	482,565	96,513	468,431	93,686	950,996	95,100	9,510	

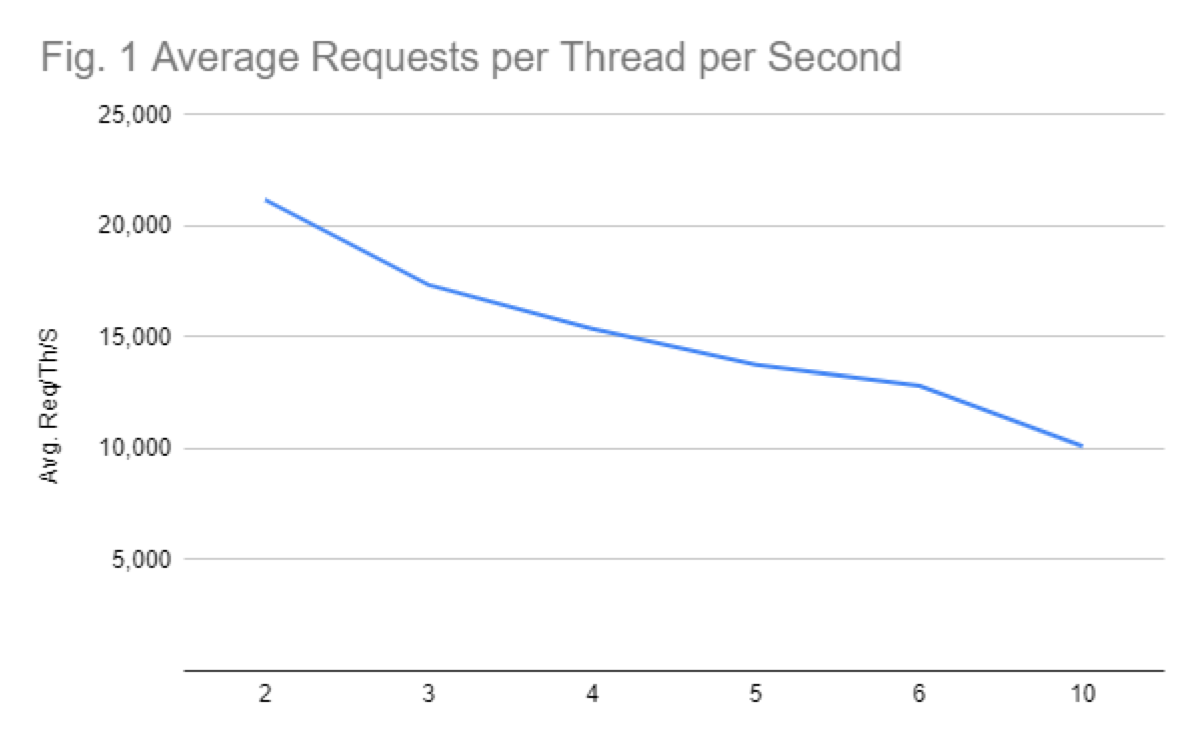
Fig. 1 Average Requests per Thread per Second
Avg. Req/Th/S
25,000
20,000
15,000
10,000
5,000
2
3
4
5
6
10

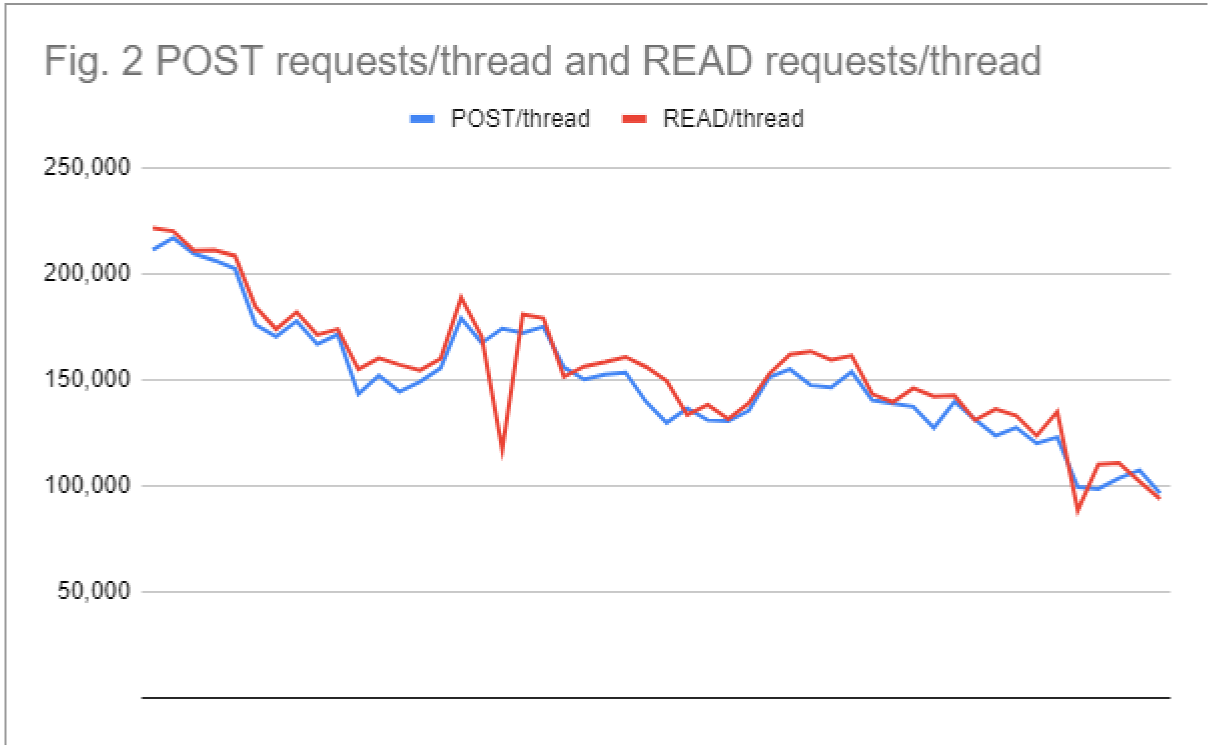
Fig. 2 POST requests/thread and READ requests/thread
POST/thread
READ/thread
250,000
200,000
150,000
100,000
50,000

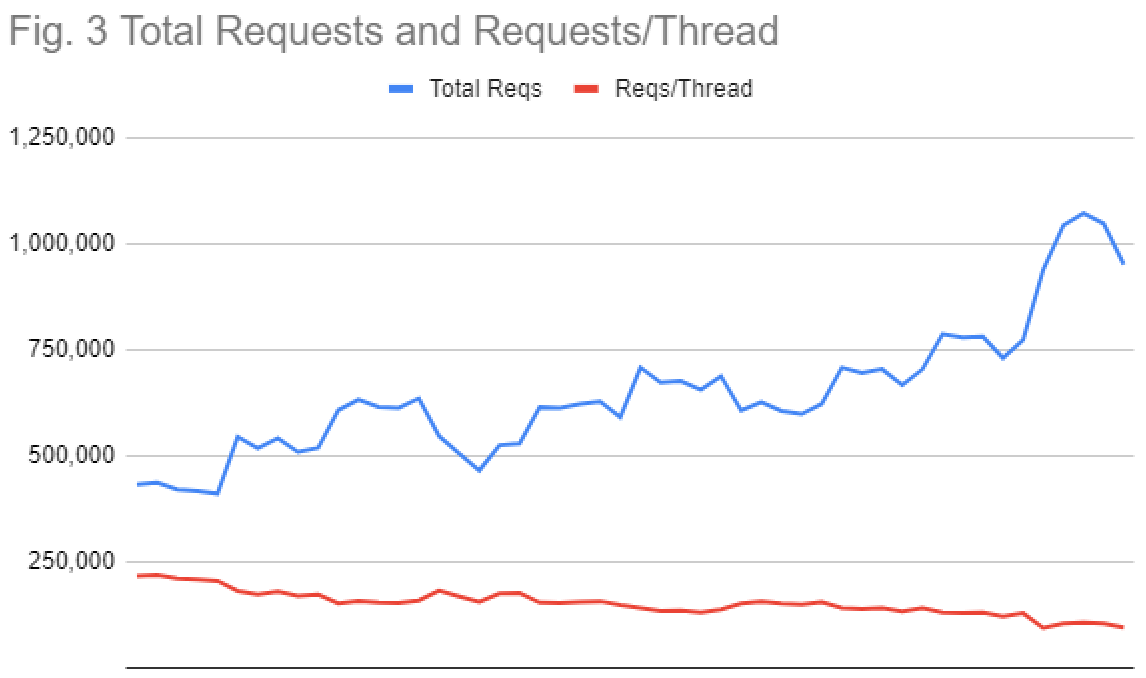
Fig. 3 Total Requests and Requests/Thread
Total Reqs
Reqs/Thread
1,250,000
1,000,000
750,000
500,000
250,000