ECE568 hw4 Scalability Test Write Up
students: xg77/hw269

1. multi-thread policy test and analysis

Upon accepting new socket connection requests, the server side can spawn a new thread do handle this newly accepted client. We implemented two strategies of server-side multi-threading and tested their performance.

multithread policy tests	one thread per client			40 thread pool		
# of concurrent clients / time to finish all client tasks(ms)	20	150	200	20	150	200
test 1	70	438	622	122	942	1010
test 2	98	415	849	99	1006	N/A
test 3	102	715	786	110	956	N/A
test 4	99	610	990	96	1220	N/A
test 5	96	895	651	116	899	N/A
test 6	81	572	570	90	927	N/A
test 7	88	428	323	133	1008	N/A
test 8	101	476	991	55	867	N/A
test 9	98	614	1401	101	929	N/A
test 10	74	625	789	174	1054	N/A
Avg total time	90.7	578.8	797.2	109.6	980.8	\
Avg delay per request	4.5	3.9	3.9	5.5	6.5	\
throughput = request per sec	221	259	251	182	153	\

As shown in the above table, the limitation of both versions was alike. We found that both policies met some connection errors when the number of concurrent clients went beyond 200. Sometime the ‘one thread per client’ policy can handle up to 800 clients gracefully, yet it was not stable. The limitation of the 40-thread pool policy was a bottleneck. Had we implemented it correctly, there wouldn’t be such a small number as its limitation. We will keep learning to make sure we gain a deeper understanding about thread-pool and its performance.

When it comes to delay and throughput, it was obvious that the 40-thread pool version paled. This was reasonable to us, because there was a restriction on concurrent server-side threads in this strategy compared to the first strategy.

For the ‘one thread per client’ strategy, there was no obvious divergence of throughput among different number of requests tested. We think this was because the number of threads was not large enough to push the VM to its limitation.

2. Multi-Core Tests

Multi-core tests	100 <create> requests			100 <transactions> requests		
# of running cores / time to finish all client tasks(ms)	1	2	4	1	2	4
test 1	377	462	304	3051	4094	3428
test 2	431	455	358	5210	1768	1901
test 3	396	382	423	4428	3980	2791
test 4	425	363	227	6012	2895	1628
test 5	459	300	393	3959	1908	1644
test 6	420	287	400	4401	2540	2624
test 7	443	268	412	3827	3659	2420
test 8	496	404	430	3708	1901	2600
test 9	540	253	420	3928	2727	3773
test 10	362	414	399	3991	2891	2590
Avg total time	434.9	358.8	376.6	4251.5	2836.3	2539.9
Avg delay per request	4.3	3.6	3.7	42.5	28.4	25.4
throughput = request per sec	229	278	265	23.5	35.2	39.3

As shown in the above picture, we did some more tests based on the content type of the requests and on the different number of cores utilized. When comparing <create> requests with <transactions> requests, the former took much less time. This was reasonable because the server will do more operations when dealing with a <transactions> request (this involves running order-matching algorithm and frequent query of the database).

When we limit number of the running cores using taskset command, we got the following conclusions: The throughput as well as average delay was almost the same for <create> requests when running on different number of cores. However, the throughput increased when running <transactions> requests on more cores. We think this was because the processing of a <transactions> requests is more complicated and thus requires more resource. In this case, the advantage of using more cores become clear.