Scalability Analysis

Yijun Mao, Chixiang Zhang, Qing Lu

1 Introduction

In order to simulate the scenario of a server handling multiple client requests and test its performance and scalability, we create client and server programs. To run the program, first you need to open two terminals, one is for the server and one is for the client.

server has three parameters:

- The first parameter “s” indicates it runs as server
- The second parameter indicates the way of creating threads. “0”: it uses thread-per-request, “1”: it uses thread pool method to handle coming requests.
- The third parameter indicates how many buckets you want to have for the server side. 1 for 32 buckets, 2 for 128 buckets, 3 for 512 buckets, 4 for 2048 buckets.

client has four parameters:

- The first parameter “c” indicates it runs as client
- The second parameter indicates how many request will be handled at the same time
- The third number represents whether it has small delay variation or large delay variation
- The fourth parameter indicates how many buckets you want to have for the server side. 1 for 32 buckets, 2 for 128 buckets, 3 for 512 buckets, 4 for 2048 buckets.

In this project, we used two methods to handle the request. One is creating thread per request. Once the request is received from the server, the server will create one thread to handle it. The other is creating multiple threads in a threadpool in advance. We utilized lzpong’s thread pool library (https://github.com/lzpong/threadpool) to realize it. The capacity of our thread pool was set to 512. Also, in order to evaluate the throughput of the two methods, we used a variable named count to count how many threads has been handled by the server. When computing the elapsed time, we set up a timer which starts when the server first time successfully accepts a request from the client. The throughput could then be calculated by (count) / (current time - start time) as shown in Fig 1.

```
double throughputCnt(struct timeval& start, int count) {
    struct timeval current;
    double elapsed_seconds;
    gettimeofday(&current, NULL);
    elapsed_seconds = current.tv_sec + current.tv_usec/1000000.0 - (start.tv_sec + start.tv_usec/1000000.0);
    return count/elapsed_seconds;
}
```

Fig. 1 Calculate Throughput

Next, we will discuss how the number of cores, buckets and requests, and variations in delay impact performance (throughput). The performance might be affected by other parameters, like network speed and virtual machines performance . In the assignment, we rule out outside parameters.

2 Performance Analysis

In order to analyze how the aforementioned parameters and two multithreading methods influence the performance, we only changed one parameter for each testing and keep other parameters unvaried. Also, we collected the data by running tests on each set of parameters five times, computed their means and standard deviations, and plotted them in test_plots.ipynb.

2.1. Throughput vs. number of cores

We fixed other parameters: Bucket size: 128; Number of requests: 1024; Small variation and only changed cores to run the program: 1 core, 2 cores and 4 cores, in order to see how the number of cores influences throughput of the system.

With the increasing numbers of cores, the average throughput of the system improves for thread per request. This result indicates that adding more cores can improve the performance of the system, which shows a good scalability of the server. However, for the thread pool method, the throughput is almost unchanged, which is 7 requests per second. Because threads in threadpool have already been created in advance and been assigned to handle requests, the performance is stable even though CPU increases.

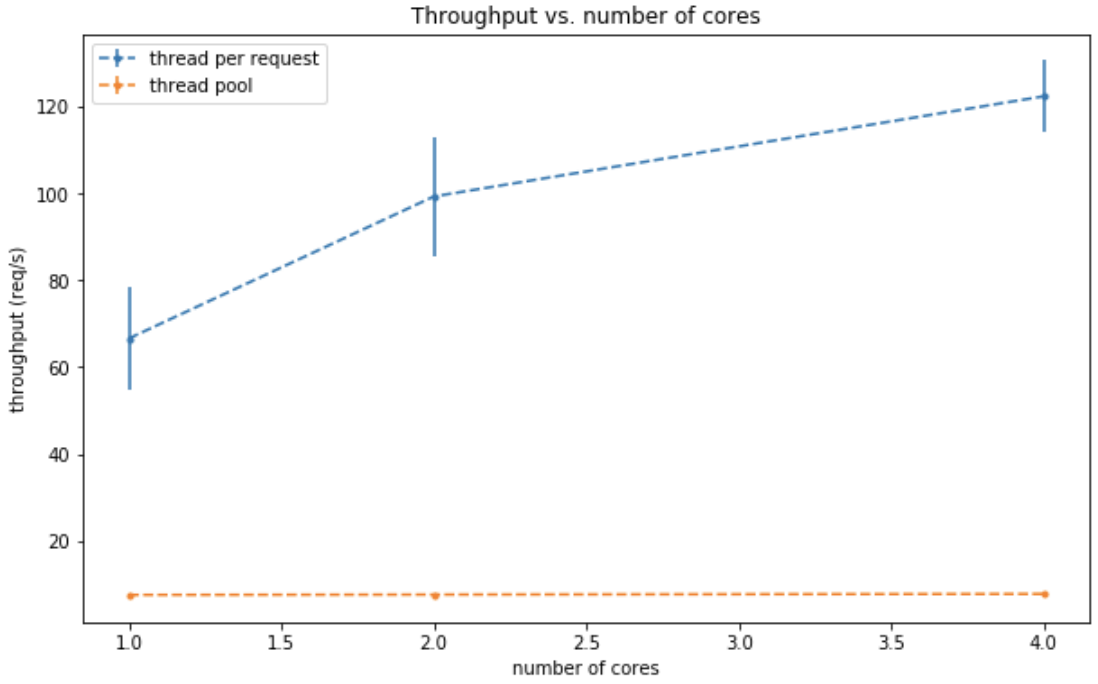

Fig. 2 Throughput vs. number of cores

2.2 Throughput vs. number of requests

We fixed other parameters: 4 Cores; Bucket size: 128; Small variation; and changed the number of requests to: 256, 512, 768, and 1024, in order to see how the number of requests influences throughput of the system.

When we increased the number of concurrent requests and the loads increased, the average throughput of thread per request increased in the first period. However, the increasing tendency is not very

obvious when requests become 1024, because the system tends to saturate and the throughput will decrease when sending more requests. Thus, with the increasing number of requests, the throughput might decrease and take longer time to complete the requests.

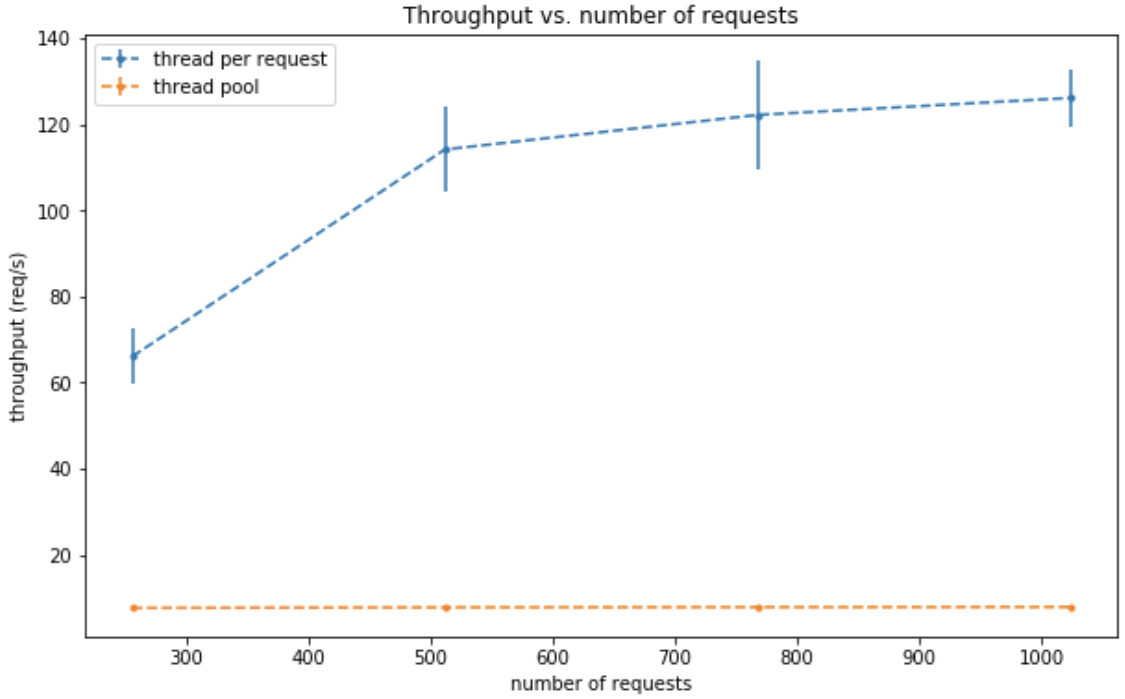

Fig. 3 Throughput vs. numbers of requests

2.3 Throughput vs. variations in delay

We fixed other parameters: 4 Cores; Bucket size: 128; Number of requests: 1024; and changed the variation of delay count to: 0 for small variation and 1 for large variation, in order to see how variation of delay influences throughput of the system.

When the variation of delay was changed from small to large, both methods' throughput decreased. Thread per request decreased from 145 req/s to 25 req/s, while thread pool method decreased from 7req/s to 2 req/s. The reason why throughput decreases is that average delay increases in request, which will reduce the throughput.

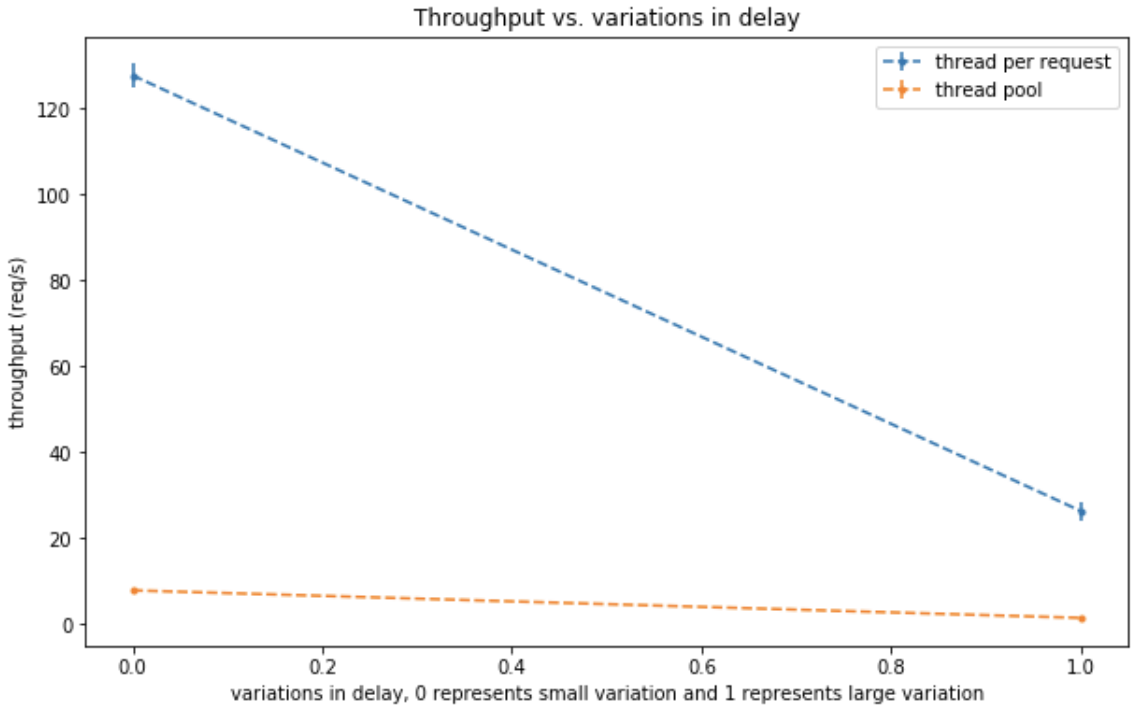

Fig. 4 Throughput vs. variations in delay

2.4 Throughput vs. number of buckets

We fixed other parameters: 4 Cores; Number of requests: 1024; Variations in delay: small; and changed the number of buckets to: 32, 128, 512 and 2048, in order to see how the number of buckets influence throughput of the system.

Actually, after 10 experiments, the change of number of buckets has little influence on the throughput of the system for both methods. Also, it may expect some contingency due to network speed. The average throughput decreased from 137 req/s to 128 req/s when the bucket numbers increased from 512 to 2048 for thread per create method, which slightly increased the throughput. For the thread pool method, it almost had no change, with the increasing buckets.

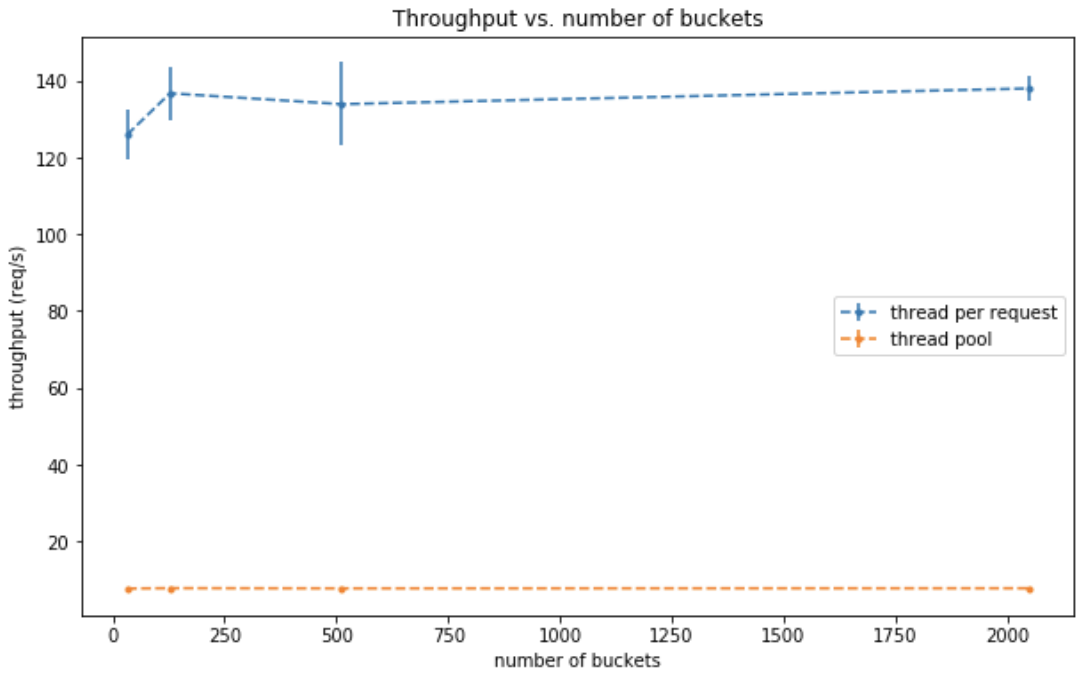

Fig. 5 Throughput vs. number of buckets

2.5 Create Per Request vs Pre-create

Under the same condition (same request, same variation), it is surprising to find the speed of thread pool (pre-create) is lower than thread-per-request method. The reason may be the frequent use of mutex to avoid race conditions and the switch of different threads in threadpool may take longer time than the thread-per-request method. However, the thread pool method is more stable than the thread-per-request method.

3 Conclusion

According to the analysis we made, the major facts that impact the scalability of our server are core number, request number and variation in delay. Bucket size has nearly no impact on the scalability since the throughput changes randomly when alternating bucket size. As for thread creation strategy, the thread per request strategy has a much higher throughput compared with the pre-create strategy. However, thread-per-request is less stable since the socket fd would crash if the client sends too many requests that exceeds the maximal number of file descriptors the Linux system can handle, which is 1024 in our case.

Appendix

Bucket size 128, Number of requests 1024, Small variation (1-3s)					
	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
1 core, thread/req	56.1906	52.1527	67.8798	71.0475	85.6392
1 core, thread pool	7.50762	7.47231	7.52384	7.77634	7.50509
2 cores, thread/req	86.04	118.712	82.9395	110.439	98.1843
2 cores, thread pool	7.62342	7.62838	7.66904	7.51487	7.68843
4 cores, thread/req	133.721	109.311	117.982	124.302	126.273
4 cores, thread pool	7.61735	7.81952	7.88469	7.66441	7.9092

Table 1 throughput vs. number of cores

Bucket size 128, Number of requests 1024, Small variation (1-3s)					
	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
256 reqs, thread/req	68.1574	70.2176	53.5012	71.1975	67.0115
256 reqs, thread pool	7.81941	7.18683	7.51291	7.89525	7.58903
512 reqs, thread/req	114.655	96.4493	114.934	117.364	127.239
512 reqs, thread pool	7.71399	7.77005	7.76198	7.76479	7.66421
768 reqs, thread/req	103.669	118.666	124.569	121.25	142.584
768 reqs, thread pool	7.70401	7.61062	7.82613	8.21644	7.59907
1024 reqs, thread/req	121.157	116.769	126.785	135.312	130.482

1024 reqs, thread pool	7.69528	7.74267	7.85524	8.00938	7.85673

Table 2 throughput vs. number of requests

Bucket size 128, 4 cores, Number of requests 1024					
	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
Small variation, thread/req	125.49	126.71	125.509	132.672	127.977
Small variation, thread pool	7.8713	7.85334	7.91612	8.02409	7.78058
Large variation, thread/req	28.3472	28.8373	24.1085	26.534	23.8825
Large variation, thread pool	1.49197	1.47374	1.4739	1.48713	1.50067

Table 3 throughput vs. variations in delay

4 cores, Number of requests 1024, Small variation (1-3s)					
	Test 1	**Test 2**	**Test 3**	**Test 4**	**Test 5**
32 buckets, thread/req	130.027	118.079	121.086	135.749	124.507
32 buckets, thread pool	7.81416	7.60265	7.81148	7.85737	7.82485
128 buckets, thread/req	137.97	123.668	143.745	139.943	138.151
128 buckets, thread pool	7.82995	7.88567	7.97907	7.83172	7.85842
512 buckets, thread/req	121.729	127.007	128.297	140.754	151.593

512 buckets, thread pool	7.93074	7.87185	7.78465	7.70104	7.83477
2048 buckets, thread/req	136.466	143.191	134.181	139.75	136.194
2048 buckets, thread pool	7.72621	7.72693	8.00165	7.93912	7.87113

Table 4 throughput vs. number of buckets