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 factors, like network speed and virtual machines performance . In the assignment, we rule out outside factors.

2 Performance Analysis

In order to analyze how the aforementioned factors and two multithreading methods influence the performance, we only changed one factor for each testing and keep other factors unvaried. Also, we collected the data from the running results and plotted the data in test_plots.ipynb.

2.1. Throughput vs. number of cores

We fixed other factors: 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.

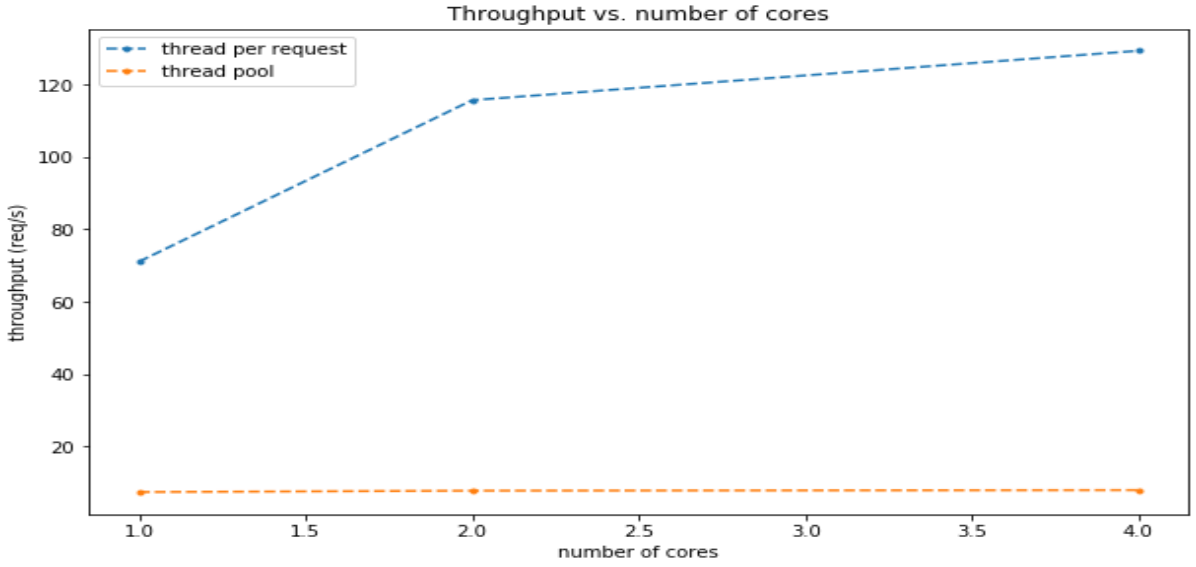

Fig 2 Throughput vs. number of cores

2.2 Throughput vs. number of requests

We fixed other factors: 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.

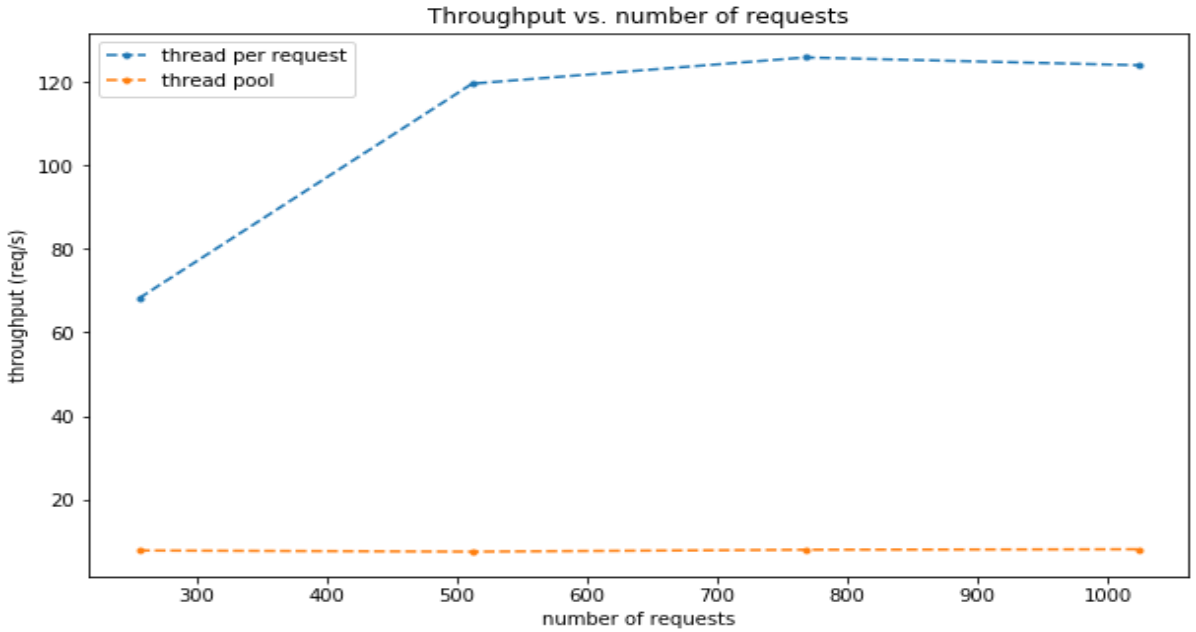

Fig 3 Throughput vs. numbers of requests

2.3 Throughput vs. variations in delay

We fixed other factors: 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 reduces the throughput.

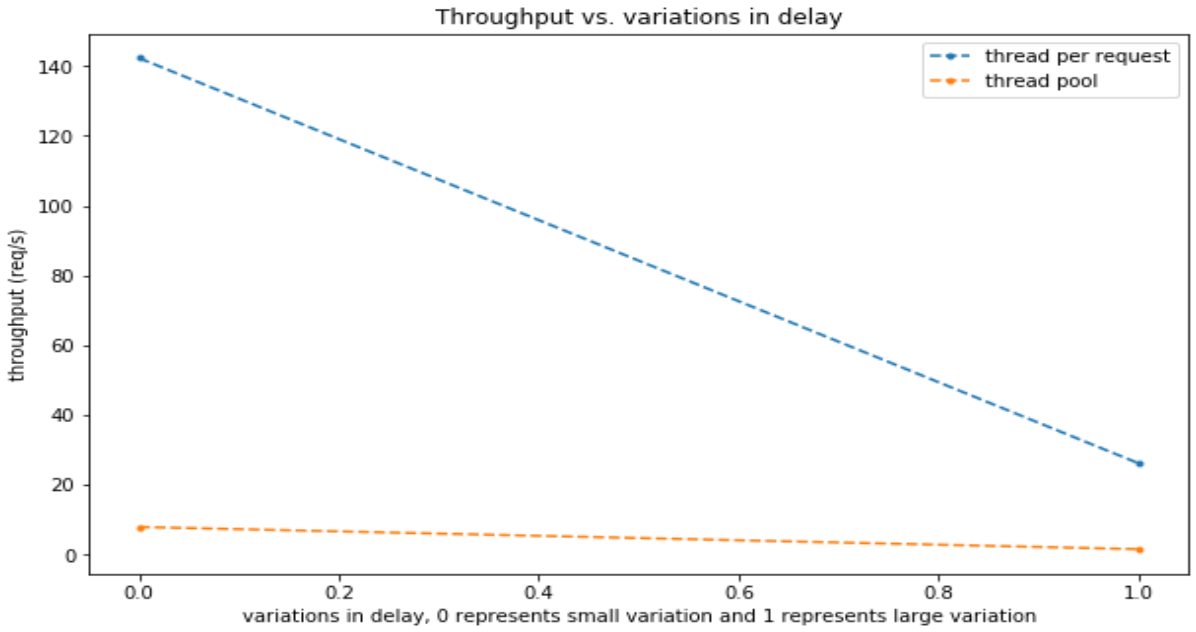

Fig 3 Throughput vs. variations in delay

2.4 Throughput vs. number of buckets

We fixed other factors: 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.

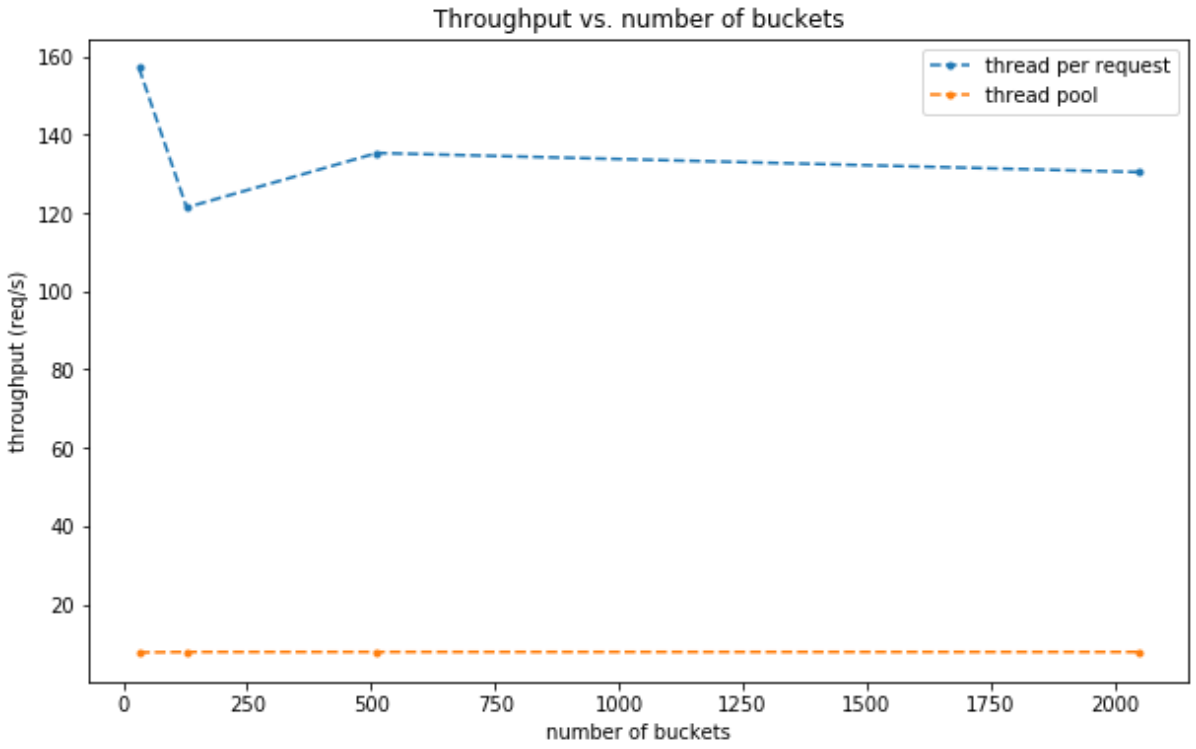

Fig 4 Throughput vs. number of buckets

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.