ERSS HW4 Report

Jingyi Xie (jx95), Yi Mi (ym154)

Introduction

In this assignment, we implemented a server software `server.cpp` that can receive requests, perform small tasks, and send back the result. We also built a testing infrastructure `test.cpp` in order to study the performance and scalability of the server software. The configurations can be easily altered to meet the different testing requirements.

Implementation

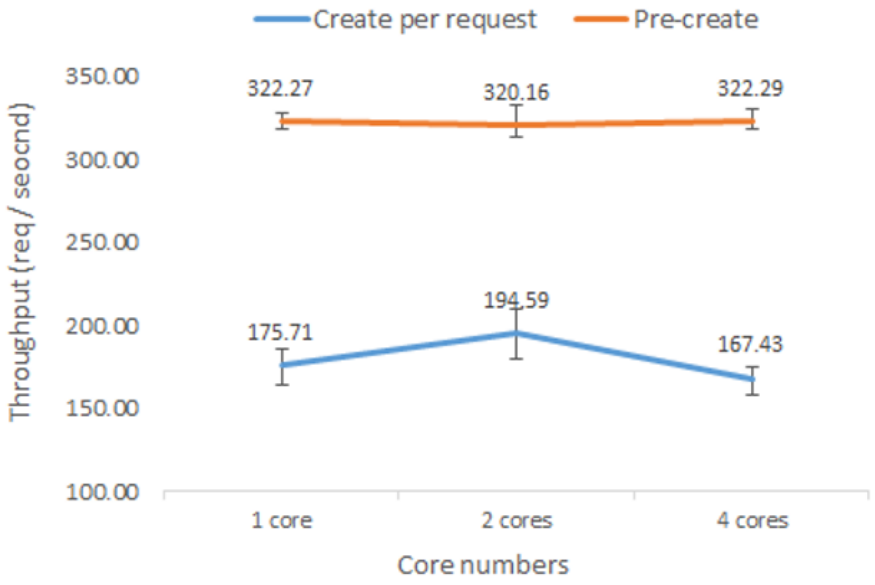
- **Server**
  - We use a `Server` class to implement the functionality. The constructor creates the `ServerSocket` and `listen` on incoming requests. Then we use `Server::per_run()` and `Server::pre_run()` to realize the two policies.
  - **Create a thread per request:** In this policy, the main thread has a `while (1)` loop and keeps accepting incoming requests. It then creates a new thread per request to do the rest of the work.
  - **Pre-create a set of threads:** In this policy, we first create `POOL_SIZE`s of threads. Then the main thread enters a `while (1)` loop which keeps accepting requests an adds the returned `client_fd` to the shared data structure `request_fds`. Meanwhile, each child thread also has a `while (1)` loop that pops the fd from `request_fds`, upgrades the bucket, and then sends back the new value.
- **Testing Infrastructure** We also use multiple threads in `test.cpp`, we first call the `getaddrinfo()` in the main thread. Then, `host_info_list` is passed to every child thread. The child thread just keeps connecting the server, sending the randomly generated requests, and then receiving the new value.

Performance & Scalability Analysis

For each combination below, we tested 3 times and made a line chart with error bars.

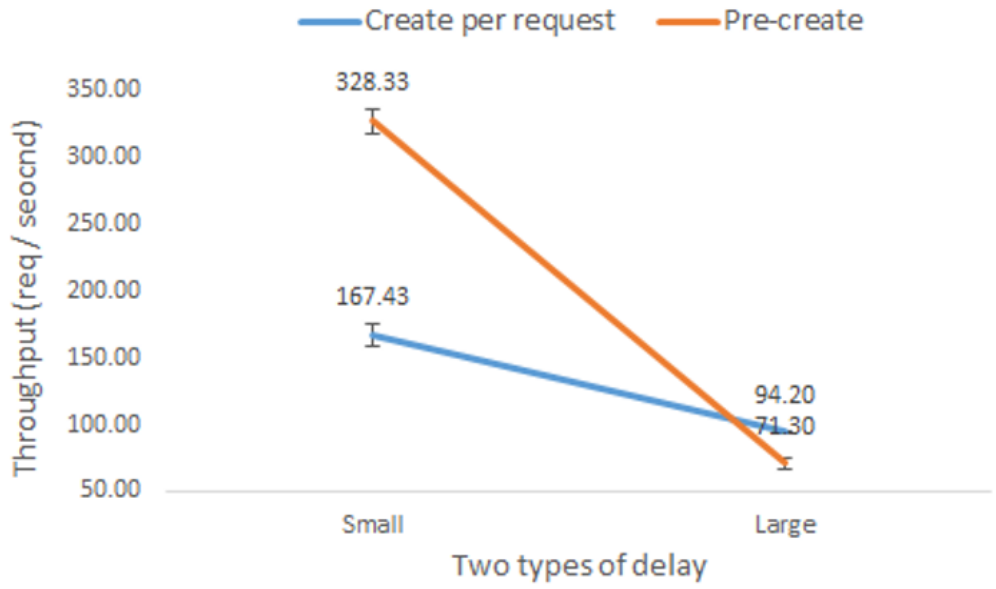
- **Different Core Numbers** (Note: Tested with **small delay and 32 buckets**)

	Core	Time	Req #	Throughput	Req #	Throughput	Req #	Throughput
PER	1	30 s	5335	177.83	4916	163.87	5563	185.43
PER	2	30 s	5364	178.80	5857	195.23	6292	209.73
PER	4	30 s	5082	169.40	4736	157.87	5251	175.03
PRE	1	30 s	9522	317.40	9842	328.07	9640	321.33
PRE	2	30 s	9461	315.37	9964	332.13	9389	312.97
PRE	4	30 s	9903	330.10	9555	318.50	9548	318.27



- We first test how different core numbers impact the performance of the two strategies. Overall, the graph illustrates that with the `POOL_SIZE` of 800, the pre-create strategy performs much better than the create-per-request strategy. The reason is that in create-per-request policy, there is no limit on the number of threads, which may be highly inefficient. Also, the overhead of `pthread_create()` is on the critical path. However, in pre-create policy, this overhead is upfront. We also learned that load balancing is also easier in this case.
- In addition, what is counter-intuitive is that the throughput does not monotonously grow as the core number increases. We considered that it's the delay of communication and allocation between cores that impacts the throughput. As for the variance, we found that the create-per-request strategy has a bigger variance at its best performance. Therefore, we believe that for this policy, while peeks at two-core, its performance is indeed unstable and subject to the influence of external factors.

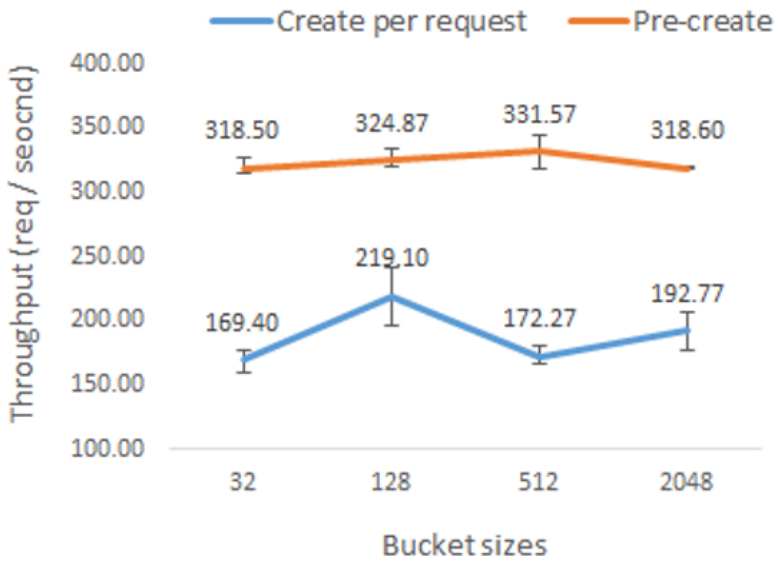
- **Different Delay Variations** (Note: Tested with **4 core and 32 buckets**)



	Delay	Time	Req#	Throughput	Req#	Throughput	Req#	Throughput
PER	Small	30 s	5082	169.40	4736	157.87	5251	175.03
PER	Large	30 s	2863	95.43	2806	93.53	2809	93.63
PRE	Small	30 s	9903	330.10	9555	318.50	10092	336.40
PRE	Large	30 s	2226	74.20	2000	66.67	2191	73.03

  - With 4 cores active and small delay count, the throughput of pre-create is twice as that of create-per-request.
  - However, when delay count grows larger (1 - 20 seconds), the throughput of both strategies plunges. It should be noted that the throughput of the pre-create strategy decreases at a faster speed, which means that this policy is not suitable for large delay count.

- **Different Bucket Size** (Note: Tested with **4 core and small delay**)



	Bucket#	Time	Req#	Throughput	Req#	Throughput	Req#	Throughput
PER	32	30 s	5082	169.40	4736.00	157.87	5251	175.03
PER	128	30 s	5912	197.07	6573.00	219.10	7271	242.37
PER	512	30 s	5168	172.27	5093.00	169.77	5515	183.83
PER	2048	30 s	5286	176.20	5783.00	192.77	6189	206.30
PRE	32	30 s	9903	330.1	9555	318.5	9548	318.27
PRE	128	30 s	9746	324.87	10178.00	339.27	9726	324.20
PRE	512	30 s	9947	331.57	10387.00	346.23	9583	319.43
PRE	2048	30 s	9558	318.60	9557.00	318.57	9612	320.40

  - We then studied the impact of different bucket sizes with 4 cores active, small delay count and 800 `POOL_SIZE`. Theoretically, the frequency of accessing the same bucket will be reduced with a bigger bucket size, thereby decreasing the lock time and increasing the throughput.
  - However, the performance of the pre-create strategy is minimally affected and rather stable. And the throughput of the create-per-request strategy is more unstable with

great fluctuation. There is no monotonic pattern. When bucket size is 128, it has the biggest throughput and the biggest variance as well. We believe this difference is due to the lack of testing time because the two policies both use `requestHelper` to access and update the buckets.