report

Team member: Boyan Hou(bh214), Kefan Lin(kl352)

This project focuses on studying different load distribution and multi-threading strategies' impact on server software's performance and scalability. The code is all written in C++ with well-thought OO Design, which is to improve the project's modularity and maintainability.

The server's task is to create N buckets and the beginning then continuously receive requests from the client, who specifies "delay count" and "bucket index" in each request, and use multiple threads to first simulate the process of "performing the task" according to the delay count and finally, add the delay count to the specified bucket.

In this project, the server-side code is written in a way that it is highly configurable regarding its threading strategy (pre-request-create and per-request-create), the number of pre-created threads and the server's running time. In the mean time, the core number allowed to be used by server can be configured in the provided docker yml file.

The client-side test infrastructure is built to be able to continuously send randomly generated requests to the server with a number of pre-created threads. The thread number and the range of randomly generated request can be configured by the client's input parameter. This provides convenience and accuracy for the testing procedure.

1. Test Data

All tests done during this project were proved to be under the condition that the client has fully saturated the server. All data acquired for analysis have been averaged over multiple repeated tests with exactly the same software and hardware to guarantee accuracy and credibility.

Data points 1~6: two threading strategies (pre & per-request-create) + running with 1,2,4 cores

Thread policy	Num of cores	Delay	Client threads	Num of bucket	Time(s)	Process requests	Throughput
pre-create	1	1-3s	1000	2048	300	90466.33	301.55
pre-create	2	1-3s	1000	2048	300	115481.67	384.94
pre-create	4	1-3s	1000	2048	300	137161.67	457.21
per-create	1	1-3s	1000	2048	300	69181	230.6
per-create	2	1-3s	1000	2048	300	81918.67	273.06
per-create	4	1-3s	1000	2048	300	99393.33	331.31

Data point 7~10: two threading strategies + small & large variations in delay count

Thread policy	Num of cores	Delay	Client threads	Num of bucket	Time(s)	Process requests	Throughput
pre-create	4	1-3s	1000	2048	300	134996	449.99
pre-create	4	1-20s	1000	2048	300	55489	184.96
per-create	4	1-3s	1000	2048	300	99382.67	331.28
per-create	4	1-20s	1000	2048	300	56572	188.57

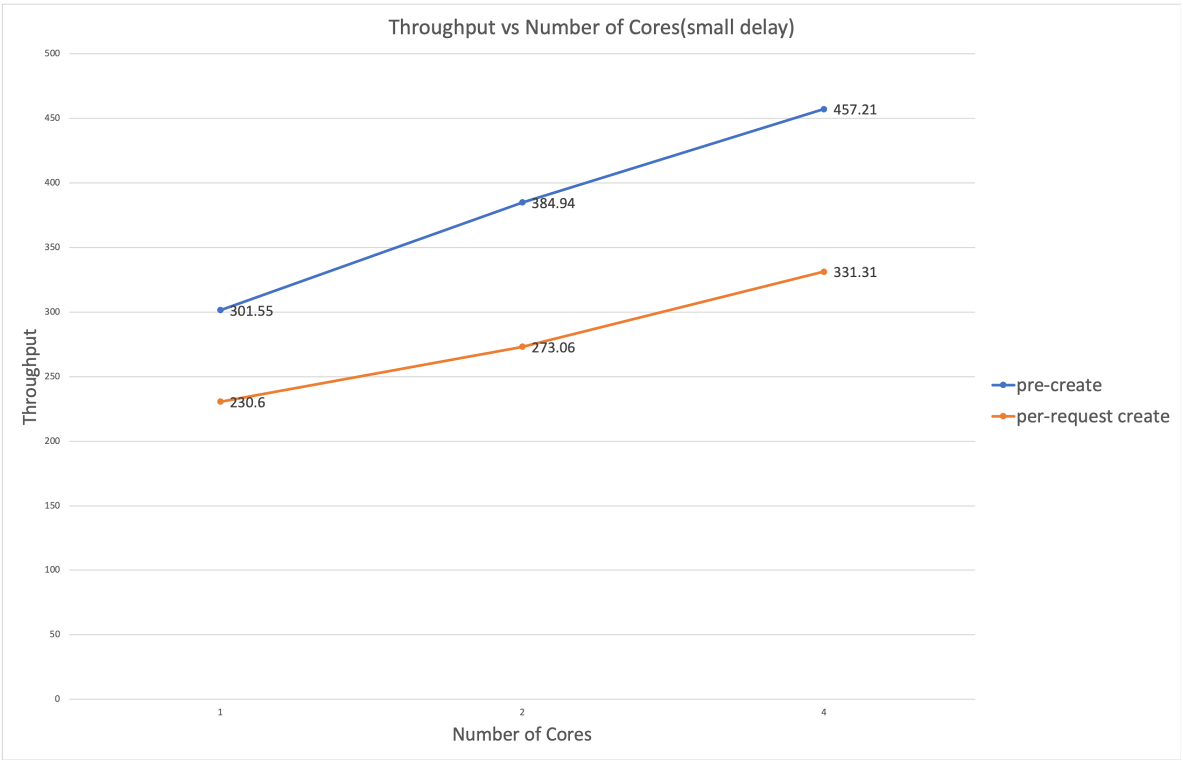
Data point 11~18: two threading strategies + different bucket sides

Thread policy	Num of cores	Delay	Client threads	Num of bucket	Time(s)	Avg throughput
pre(1000 threads)	4	1-3s	1000	32	300	437.828889
pre(1000 threads)	4	1-3s	1000	128	300	466.688889
pre(1000 threads)	4	1-3s	1000	512	300	464.894444
pre(1000 threads)	4	1-3s	1000	2048	300	463.805556
per	4	1-3s	1000	32	300	325.504444
per	4	1-3s	1000	128	300	325.967778
per	4	1-3s	1000	512	300	326.223333
per	4	1-3s	1000	2048	300	324.712222

2. Graph & analyze

In the next parts, the following dimensions are analyzed: performance scalability, two threading strategies, the performance difference with small vs large variations in delay count, and the performance difference with different bucket sizes.

part 1: performance scalability (server throughput when running with 1, 2, and 4 cores)



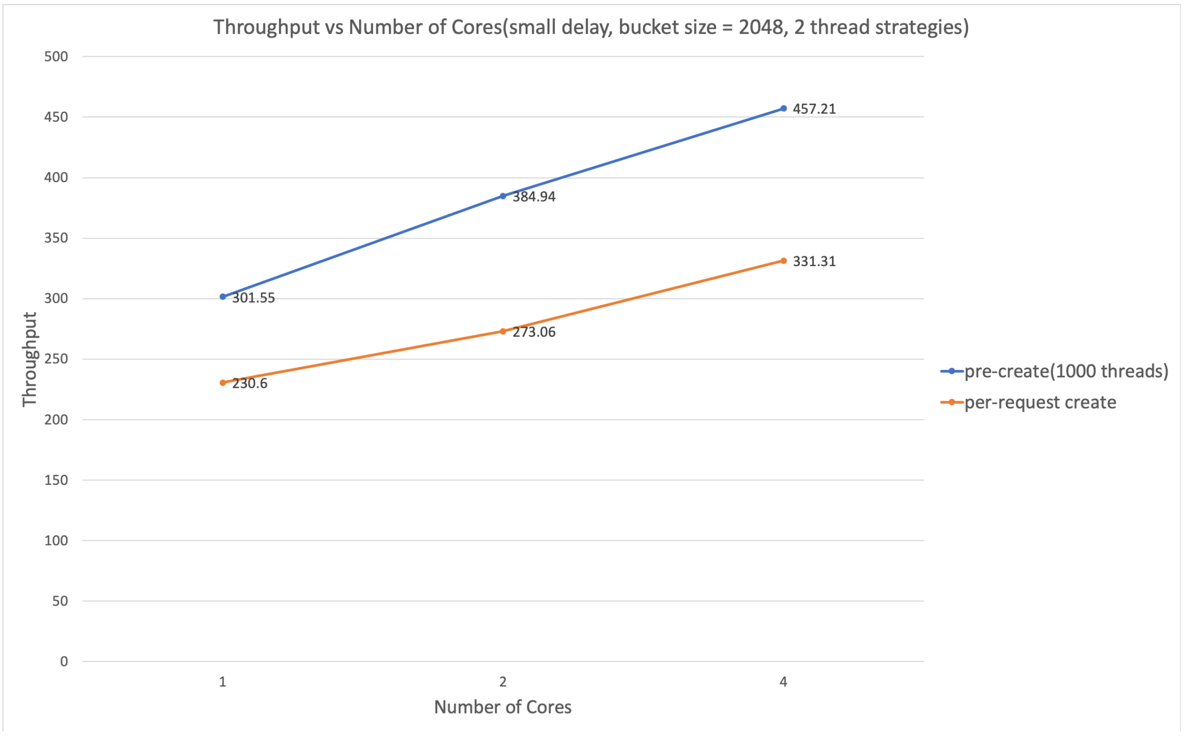
The above graph shows the throughput (processed requests / second) vs number of cores used by server. The delay amount is randomly generated from a range of 1~3 seconds.

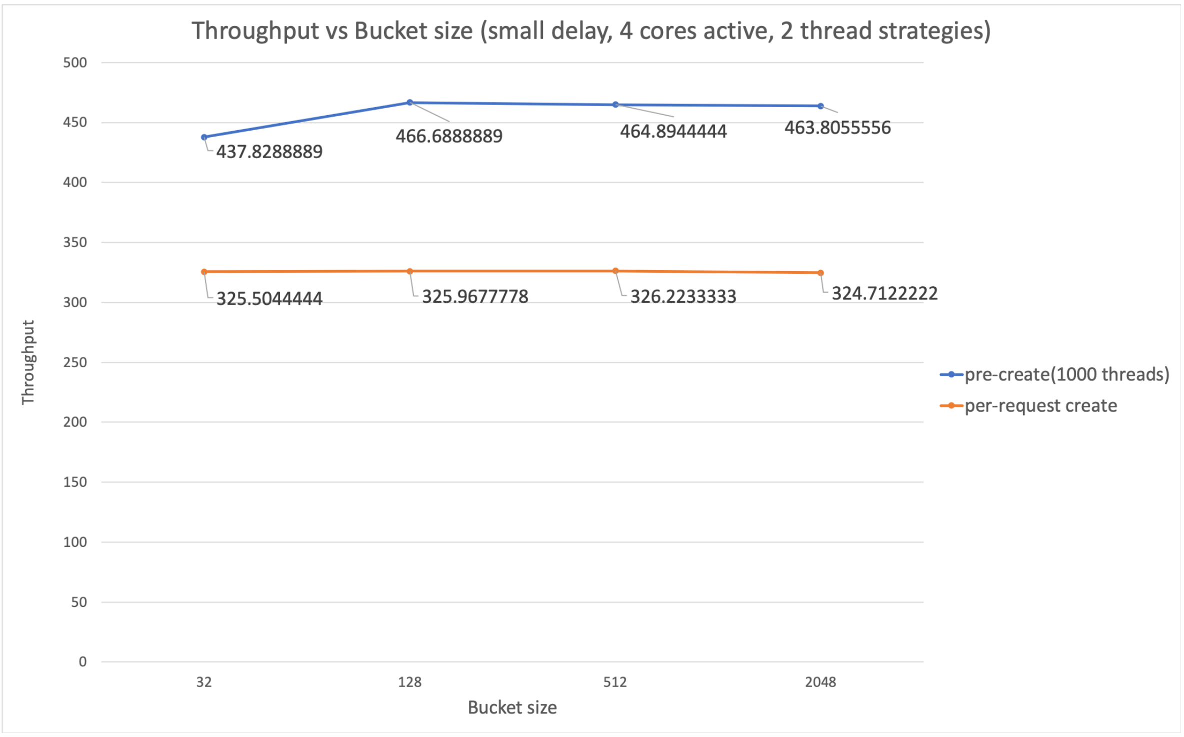
As shown in the graph, while the server is guaranteed to be fully saturated in all the tests, the number of cores used by server has a positive impact on the throughput.

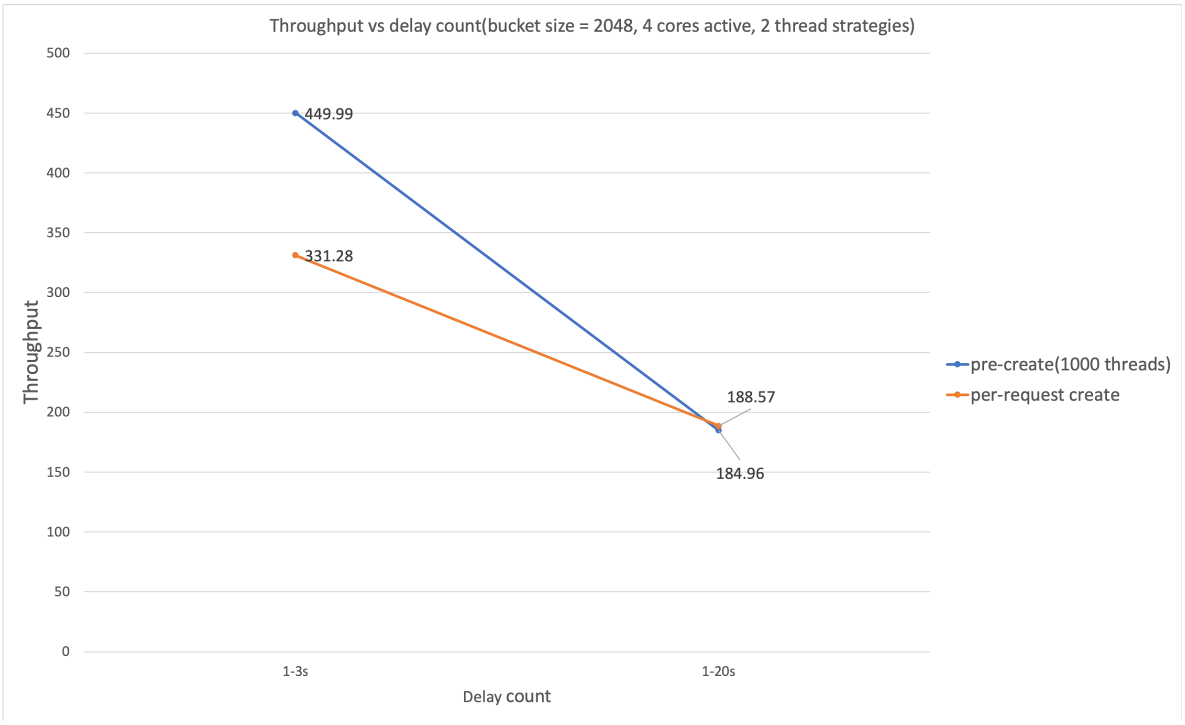
This is a reasonable result since each cpu core has fixed number of physical threads, hence puts a limit on how many software threads it can effectively carry on. The increase of core numbers increases the number of available physical threads, thus more software threads are able to run effectively in concurrent, which results in the increase in server throughput.

In the authors' opinion, this results in a good scalability for the server software. If more throughput is needed, adding in more cpu cores as hardware support would improve it effectively.

part2: Two threading strategies: (1) create per request and (2) pre-create





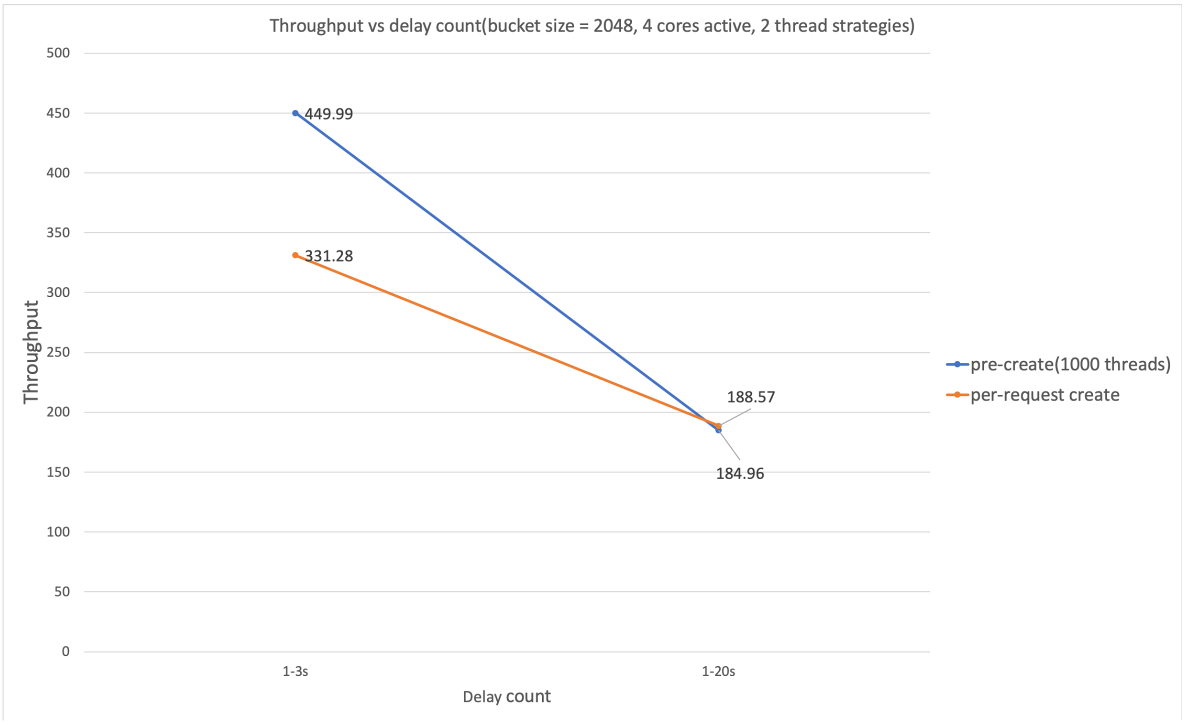


(The threading strategies are distributed in the whole experiment, so these three graphs are overlapping with the others in the other three parts.)

The above three graphs shows the server throughput under three different test conditions, with the blue lines being "pre-request-create" threading strategy and the red lines being "per-request-create" threading strategy.

From these graphs, one can analyze that under the same testing condition, pre-request-create can always out-perform per-request-create strategy regarding the server throughput. As far as the authors concern, this is due to the fact that per-request-create strategy does not re-use a thread when its job is done, hence has a larger overhead for creating a new thread for every received request. However, the advantage of per-request-create strategy is that it is relatively easy to write: unlike the pre-request-create strategy, pooling the threads is not necessary, and there is also no concern about load balancing between different threads.

part3: Small vs. large variations in delay count

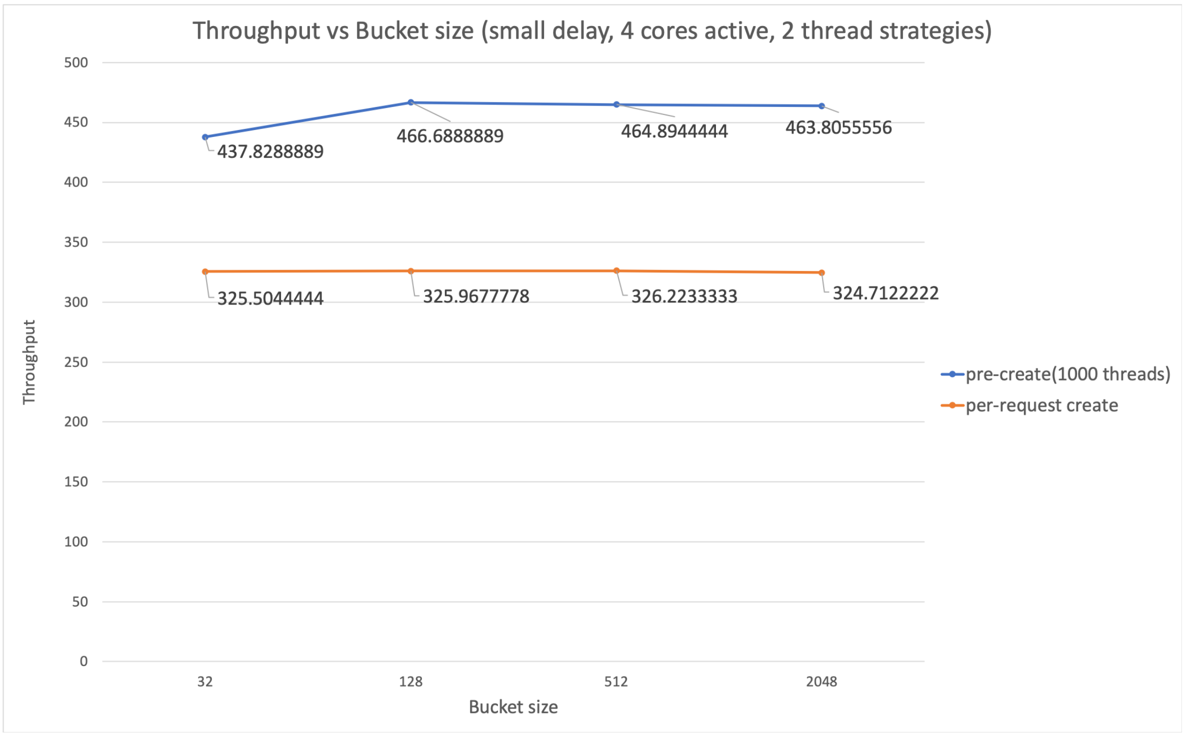


In this test, two set of delays are tested: a random delay range of 1~3 seconds, and another of 1~20 seconds.

The graph shows that the throughput of the server sees a decrease as the delay count variation increases. However, this is exactly as expected, because the average of a random range of 1~3 seconds would be 2 seconds, while the average of a random range of 1~20 seconds would be 10 seconds. This means that each request in the 1~20 seconds range would usually take a longer period for a thread to process, resulting in the degrade in the server's throughput.

However, if considering from the aspect of "delay count" alone, it is believed that the variation in it will not affect the throughput of the server in either a positive or a negative way. This is because the processing of each request are carried on in individual threads and would solely depend on its own processing time. This characteristic of the server software is recognized by the authors as good scalability.

part4: Different bucket sizes with 4 cores available



As shown above, the tests are carried out with bucket sizes being 32, 128, 512, 2048. The result shows that the bucket size does not have much of an impact on the server's throughput. As far as the authors concern, since the "bucket"s are implemented in a "vector of integer" data structure and the "add number" operation is just adding integer to an integer, the throughput would not be impacted by the increase in bucket size because the aforementioned operations are both O(1) in time complexity and will not scale up as the bucket size grows. This also indicates very good scalability of the server software.

3. Conclusion

During this project, a server software and its testing infrastructure are built. The server software's functionality and scalability have been tested in multiple key aspects. From the acquired data, the authors come into the conclusion that this server is robust enough and has good scalability. The knowledge and experience gained throughout the process will be applied to future study, research and work and will be part of a valuable asset of the authors.