ERSS Homework4 Report

Kuan Wang (kw300)
Yue Yang (yy258)

Introduction

In this assignment, we implemented a server that receives requests from clients and performs a certain task before responding to the client. The server uses two different threading strategies, “create a thread per request” and “pre-create a set of threads”, to service requests. We also implemented client software as the testing infrastructure to test the server on its functionality and scalability. The client-side infrastructure sends requests rapidly enough to saturate the server throughput, so that the server code is stressed to reach its largest throughput. We collected performance measurements in different dimensions by altering the arguments in client and server software, and analyzed the measurement based on the implementation.

Implementation

create a thread per request

This threading strategy indicates that one thread is created to handle a request when the server receives a new request, and this thread exits after processing the request. Hence,we use a *while(1)* loop to detect whether there is a new request and create a thread when a new request is detected.

pre-create a set of threads

This threading strategy indicates that a certain number of threads is pre-created. We use a *for* loop for pre-creation. When the server receives new requests, one thread handles it. Each pre-created thread executes a *while(1)* loop to keep processing the requests. Hence, larger number of ore-created threads results in larger throughput but more overhead.

testing infrastructure

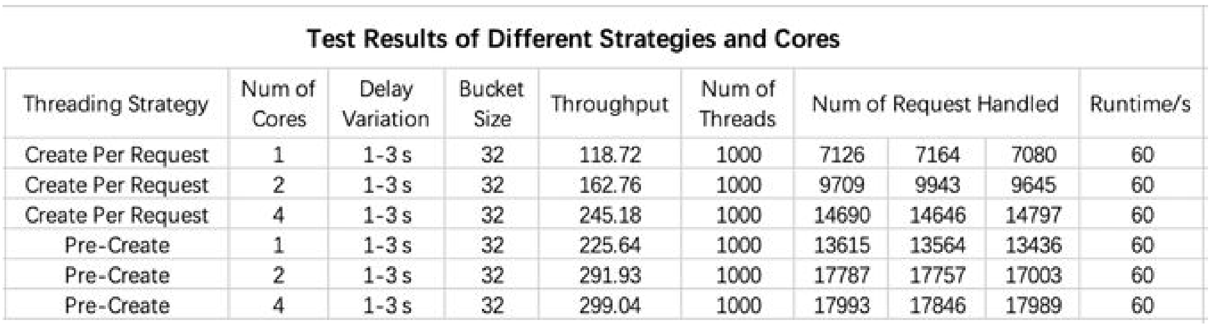
The client-side infrastructure takes five arguments, which are hostname, lower and upper bound of delay count, bucket size and number of threads to send requests, respectively. The client software first makes a socket connection with the server, and creates many threads to keep sending requests to the server simultaneously. We can increase the speed of sending requests by creating more threads. In this way, we stress the server code to drive the performance tests.

Performance Analysis

To better test the scalability of our program, we follow the rules of load testing when we write our code and test case. Specifically, we follow Rule 1 which is about generating a lot of load. We use multi-thread in our client-side code. Client has 1000 threads and each thread is in a *while(1)* loop to keep sending requests.

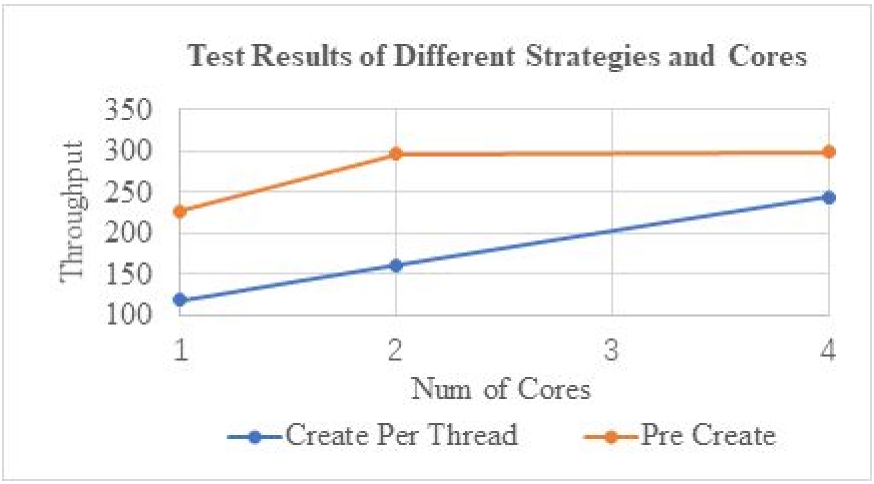
We test the throughput of server handling requests with combinations of different threading strategies, number of cores running the code, delay variations and bucket sizes.
For threading strategies, we test create per request strategy and pre-create strategy. For the number of cores running the program, we test running with 1, 2 and 4 cores. For delay variations, we test delays of 1-3 s which represents for small delay count variability and delays of 1-20s which represents for large delay count variability. For bucket sizes, we test 32, 128, 512 and 2048 buckets.
Below are our test cases.

Different Strategies and Cores



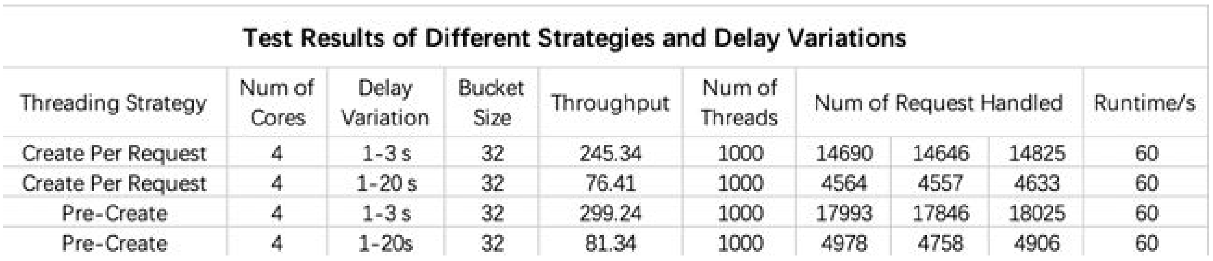
Test Results of Different Strategies and Cores

Threading Strategy	Num of Cores	Delay Variation	Bucket Size	Throughput	Num of Threads	Num of Request Handled			Runtime/s
Create Per Request	1	1-3 s	32	118.72	1000	7126	7164	7080	60
Create Per Request	2	1-3 s	32	162.76	1000	9709	9943	9645	60
Create Per Request	4	1-3 s	32	245.18	1000	14690	14646	14797	60
Pre-Create	1	1-3 s	32	225.64	1000	13615	13564	13436	60
Pre-Create	2	1-3 s	32	291.93	1000	17787	17757	17003	60
Pre-Create	4	1-3 s	32	299.04	1000	17993	17846	17989	60



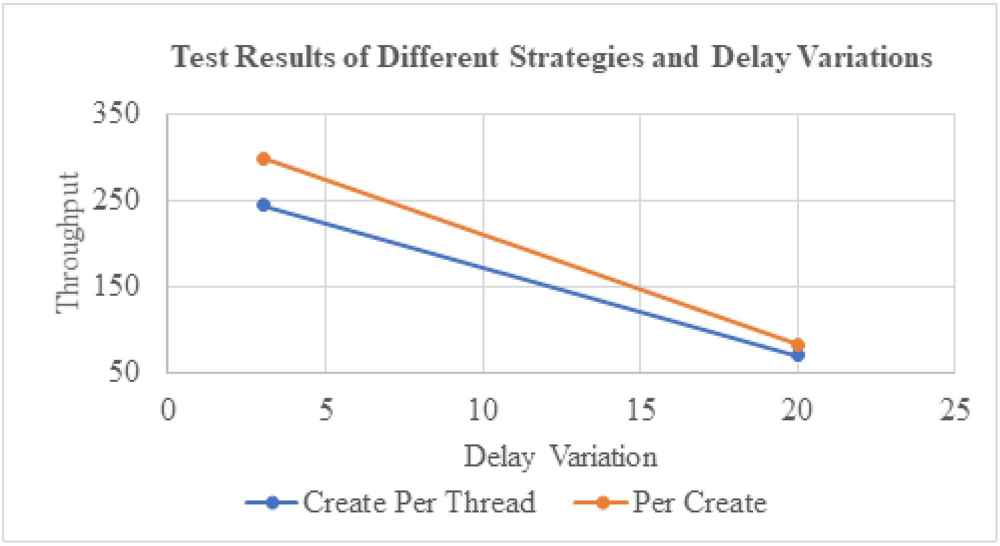
From the above results we can see that, as the number of cores which runs the code increases, the throughput increases. When the number of cores involved increases, the hardware resource contention decreases with the communication latency increases. From the results we get, we can conclude that, in our implementation, the resource contention is more dominating than communication latency, that explains why we get higher throughput when the number of cores increases.

Different Strategies and Delay Variations



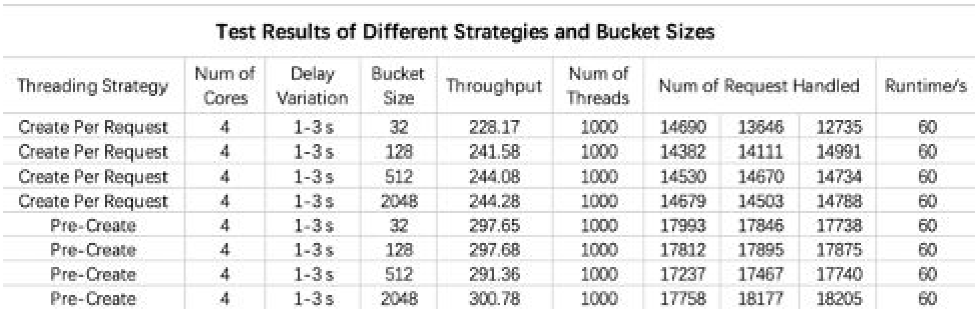
Test Results of Different Strategies and Delay Variations

Threading Strategy	Num of Cores	Delay Variation	Bucket Size	Throughput	Num of Threads	Num of Request Handled			Runtime/s
Create Per Request	4	1-3 s	32	245.34	1000	14690	14646	14825	60
Create Per Request	4	1-20 s	32	76.41	1000	4564	4557	4633	60
Pre-Create	4	1-3 s	32	299.24	1000	17993	17846	18025	60
Pre-Create	4	1-20s	32	81.34	1000	4978	4758	4906	60



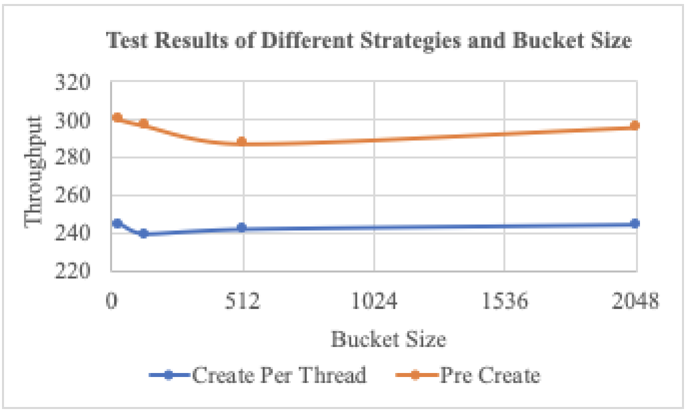
From the above results we can see that, as delay variation increases, the throughput decreases for both Create Per Thread Strategy and Pre-Create Strategy. That is because as the delay time increases, each thread spends longer time in handling one request, resulting in more requests waiting to be handled. When a large number of requests come in, the number of available threads decreases, thus the throughput of the program decreases.

Different Strategies and Bucket Sizes



Test Results of Different Strategies and Bucket Sizes

Threading Strategy	Num of Cores	Delay Variation	Bucket Size	Throughput	Num of Threads	Num of Request Handled			Runtime/s
Create Per Request	4	1-3 s	32	228.17	1000	14690	13646	12735	60
Create Per Request	4	1-3 s	128	241.58	1000	14382	14111	14991	60
Create Per Request	4	1-3 s	512	244.08	1000	14530	14670	14734	60
Create Per Request	4	1-3 s	2048	244.28	1000	14679	14503	14788	60
Pre-Create	4	1-3 s	32	297.65	1000	17993	17846	17738	60
Pre-Create	4	1-3 s	128	297.68	1000	17812	17895	17875	60
Pre-Create	4	1-3 s	512	291.36	1000	17237	17467	17740	60
Pre-Create	4	1-3 s	2048	300.78	1000	17758	18177	18205	60



From the above results we can see that, as the bucket size increases, the throughput does not change much for both of the two strategies. Theoretically, a program with smaller bucket size has more possibilities to write to the same place in the bucket and race condition is more easily to happen. That will cause the program to run for a longer time. However, our results do not meet with our expectations. We speculate the reason is that the smallest delay time is way longer than the synchronization time. As delay time is more dominating, the effects caused by bucket sizes are not visible.