Concurrent and Parallel Systems

Throughput Evaluation on a Multithreaded TCP Server

Assignment 1

Andy Le
2602034

Submitted on the 23 January 2020

1 Introduction

This report discusses about the throughput evaluation of the developed TCP Client (test harness) that has been deployed on the PC in Cantor 9341 that goes by the Ethernet IPV4 address 10.72.85.143. To further elaborate on the evaluation, a data table and a graph has been supplied as a means of interpreting the performance of the TCP Server based on the message exchange rate with the test harness. The data table and graph have been generated with the R programming language using the formattable and ggplot2 libraries.

2 Experiment Strategy

The strategy devised for this experiment relies on a "tweaked" version of the TCP Client. "Tweaking" predicates the removal of redundant operations with particular focus on echoing strings since they are contributing to significant performance loss in relation to calculating the throughput. This has been accomplished by commenting them out after the developmental finalisation the TCP Server and Client to ensure that a fully functional TCP Client can perform the throughput calculation accordingly.

2.1 Request Generation

The way how POST and READ requests are generated is arbitrary as each topic and message are hardcoded in distinct data structures. Each topic is stored in a ***map<string, integer>*** where string represents the topic name and integer the amount of requests sent related to the topic whereas the messages are strings stored in a ***vector<string>***. Since the message generation is arbitrary, it means that the retrieval of topics and messages is realised by the ***RandomRange()*** function inside the ***MockRepository.cpp*** file that takes advantage of the Mersenne Twister Randomiser for random number generation. Lavavej (2013) argues that the use of the rand() is harmful due to non-uniform random number generation but advocates the use of uniform random number generators that promise an increase in performance. [2]

2.2 Throughput Calculation

To calculate the throughput, the automated sending method *(renamed to **CalculateThroughput()** in the final revision)* in the ***TcpClient.cpp*** file had to be tweaked to the extent to which an integer representing the count of fulfilled requests is incremented for each message sent. The method will continuously send messages for the duration of 10 seconds and calculate the throughput based on the quotient of the total amount of fulfilled requests and the time duration in seconds. This can be visualised with the following formula where *Messages* represents the total amount of fulfilled requests, *Start* and *End* start and end time in seconds respectively, and *Throughput* as self-explanatory the throughput in messages per seconds:

$$\frac{Messages}{End - Start} = Throughput$$

The formula represented above only represents the throughput calculation of a threaded process. In order to calculate the throughput of the test harness, the average throughput of every threaded process must be calculated. Once calculated, it will yield the average throughput of all threads which is crucial for recording a set of individual result for the throughput experiment.

3 Throughput Experiment

3.1 Tabular Representation of Throughput Averages

The following table represents 10 attempts on average throughput evaluation per experiment where ***avg_thrput_n*** represents an average throughput of an attempt n:

Experiment	Poster	Reader	avg_thrput_1	avg_thrput_2	avg_thrput_3	avg_thrput_4	avg_thrput_5	avg_thrput_6	avg_thrput_7	avg_thrput_8	avg_thrput_9	avg_thrput_10
1	1	1	18418.23	18946.88	18506.16	18344.38	18471.14	18627.31	18168.05	18872.27	18494.54	18556.86
2	1	2	16880.64	16709.87	17092.31	16845.42	17179.61	17132.73	17600.88	17352.53	17585.34	17161.93
3	1	3	16138.73	16719.65	16540.95	16152.55	16143.25	16423.27	16522.02	16527.23	16210.07	16393.63
4	2	1	17731.19	18132.70	17751.83	17837.60	17734.31	17234.64	17454.86	17788.60	17944.39	17971.02
5	2	2	16244.38	16228.61	16657.99	16176.39	16252.32	16273.95	16233.56	16433.77	16178.98	16381.82
6	2	3	14048.96	14013.50	13943.44	13953.84	13360.41	13744.43	13904.74	14018.20	13148.48	13207.17
7	3	1	16522.87	16779.11	16256.12	16196.85	16647.21	16850.04	16613.46	16585.52	16587.01	16598.38
8	3	2	13965.90	13848.67	13985.02	13936.33	13800.63	13844.43	13879.34	14037.65	13526.23	13495.43
9	3	3	11648.05	11711.53	11676.91	11823.84	11635.78	11765.21	11471.12	11778.98	11831.63	11857.43

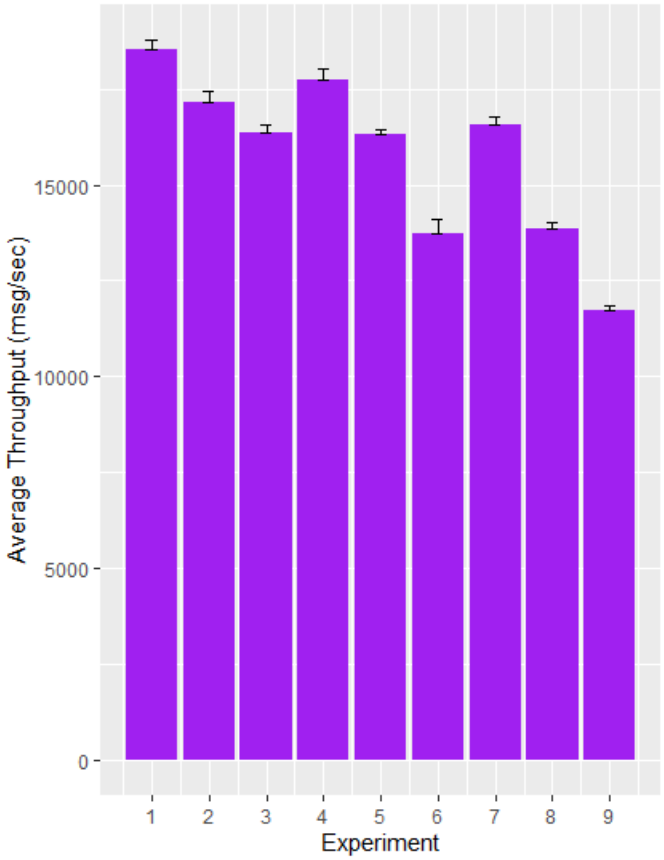
3.2 Tabular Representation of Throughput Statistics

The following table represents statistical evaluation on throughput evaluation per experiment where ***avg_throughput*** represents the average throughput, ***var_throughput*** the throughput variance, and ***sd_throughput*** the standard deviation of the throughput:

Experiment	Poster	Reader	avg_throughput	var_throughput	sd_throughput
1	1	1	18540.58	53695.10	231.7220
2	1	2	17154.13	88592.90	297.6456
3	1	3	16377.14	42217.53	205.4691
4	2	1	17758.11	66101.73	257.1026
5	2	2	16306.18	21983.26	148.2675
6	2	3	13734.32	126610.11	355.8231
7	3	1	16563.66	41212.61	203.0089
8	3	2	13831.96	33883.37	184.0744
9	3	3	11720.05	13755.10	117.2821

3.3 Graphical Representation of Throughput Averages

The following bar plot represents the mean value of each throughput experiment, with additional representation of the standard deviation:



3.4 Data Interpretation

Based on the throughput experiment conducted, it is evident that a higher throughput has been achieved based on the minimum amount of threads used as described in the table in Section 3.1 Experiment 1. The following observations have been made:

- 2 threads yielded a region of 18000 msg/sec
- 3 threads yielded a region of 17000 msg/sec
- 4 threads yielded a region of 16000 msg/sec
- 5 threads yielded a region of 13000 msg/sec
- 6 threads yielded a region of 11000 msg/sec

The observation predicates that a stepwise increase in thread number ranging from 2 to 4 resulted in a throughput loss of 1000 msg/sec whereas any number exceeding 4 resulted in significant throughput loss since the PC in Cantor has 4 hardware threads in total. In conclusion, it can be deduced that using more threads than those specified in the hardware will significantly impact the throughput negatively. The average experimental throughput enables the calculation of the throughput variance in order to highlight how each individual average throughput of an experiment relate to each other. Since the variance itself does not reveal significant information, it can be used to calculate the standard deviation which indicates the distance of individual experimental throughput from the average experimental throughput. Based on table in Section 3.2 Experiment 6, the variance and standard deviation is higher than any other

experiments. This is due to the fact that the standard deviation influences the variance. The higher the standard deviation, the higher the throughput variance.

4 Potential Performance Improvement

In order to improve server performance, it is vital to change the architectural design of the server. The current server follows an object-oriented design and changing it to a procedural style may benefit performance. According to Chatzigeorgiou (2003), OOP leads to performance penalty since it can result in a significant increase of both execution time and power consumption. [1] However, OO design can also be enhanced by taking advantage of inline functions in order to speed up the execution of smaller functions. Furthermore, decreasing code size can contribute to performance acceleration. To gain performance optimisation through the decrease of code size, the abolition of the OO design is the best choice due to the removal of class dependency. To gain additional increase in performance, the data structure used should be changed to the extent to which it rejects the insertion of duplicate values. This is to ensure that the data structure does not grow at the expense of duplicate values so that data structure-specific operations can take advantage of the best case scenario in the long run in Big O terms. Therefore, space is being traded off for time. However, empirical tests on threading should be conducted by comparing the performances of different modes of threading *std::thread, std::async, std::future* to identify the most suitable mode that may influence the decision of abolishing the use of a thread pool in favour of manual threading.

References

[1] Alexander Chatzigeorgiou. Performance and power evaluation of c++ object-oriented programming in embedded processors. *Information and Software Technology*, 45(4):195 – 201, 2003.

[2] Stephan T. Lavavej. rand() considered harmful, goingnative 2013. `https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful`, 2013. [Online; accessed 11/01/2020].