Iterative Server Assignment

Chris Hoyt, Chris Deardeuff

CNT505-Computer Networks & Distributed Processing, Professor Kelly

Table of Contents

[Introduction 3](#_Toc33985624)

[Client-Server Setup and Configuration 3](#_Toc33985625)

[Testing and Data Collection 4](#_Toc33985626)

[Data Analysis 7](#_Toc33985627)

[Conclusion 7](#_Toc33985628)

[Lessons Learned 8](#_Toc33985629)

# Introduction

The purpose of this project was to get an introduction into programing and limitations of an iterative server. The goal of the assignment was to demonstrate the increase in response time of the server when more and more clients sent requests and waited on the server to respond. This paper will outline the process we took in programing, the results we got from the server and other collected data, and the lessons that we learned from this project.

# Client-Server Setup and Configuration

The design of the server-side program was simple, it was a single threaded server that processed client requests one at a time. When starting up the server program, it prompts the user for an IP Address and then a port for the server to listen on. The server then sets up a single socket and waits for a client to connect to it, when a client connects, it notifies the server console that a client has connected and then proceeds to listen for an input from the client. When the client sends a request over, the server program takes the request into a switch case and executes the corresponding action and returns what was requested to the client, printing it onto the client’s console. After doing so, the server socket reopens and continues to wait for another client to connect and continues to repeat over and over.

The purpose of the server program is to sit and listen for a client or multiple clients on a specified IP Address and port. When the server receives a request from a client, it processes that request and then sends it back and continues to wait for another client. If multiple clients ask for a process at the same time, the server will process each request one by one, completing the first and moving on to the next when it finished with the previous.

The client program is more complex as it takes the user input for how many, and what type of request, and then will create that many clients and start their threads. Each client thread connects to the server and receives its information from the request and then prints to the console. The client generator is coded so that it uses a do while loop to continuously run the program until the user says quit. Inside the do while loop, it prompts the user for the server address, the port that the server is listening on, the number of desired clients, and the type of request to send. After the user enters all that information the program will create that many clients using a client object that inherits the Thread class which allows them to run in parallel. After the client objects are all done running, the do while loop executes again.

A design decision that was discussed was the specifics of what the request to the server was going to consist of. We needed to determine what the client would send and what the server would want. For example, if the client sent just an integer that represented a certain command, but the server wanted the command itself, there would be a problem and nothing would have been returned. We decided to send one integer that the server would then use in a switch statement to process the request.

# Testing and Data Collection

The Iterative Server was tested by pushing one to twenty five clients at it, each client would record the time it took the server to return the process and then give the client side console a total time and an average time that the server took to send the process it was asked for.

The data generally shows an exponential growth for each of the processes, being quick for one to ten clients, and then drastically increasing the time it took the server to finish returning each process to all clients when it reached fifteen clients. The following charts show the time taken for each request a client could send.

# Data Analysis

Increasing the number of clients not only affects the total turn-around time but it also affects the individual client’s turn-around time. The more clients there are, there longer the turn-around time is for clients that are waiting on clients before it to finish.

Increasing the number of clients affects the average turn-around time just as it affects the individual client’s turn-around time. As the number of clients increases, the greater the average turn-around time becomes.

The primary cause of the effect on the individual client turn-around time and the average turn-around time is that the server is only able to handle one client at a time and because of this, the server starts to become overwhelmed and lags behind when dealing with each following client.

# Conclusion

The data shows that as more and more requests are sent to the server, the higher and higher the response time takes, growing at an exponential rate. The conclusion that can be drawn from this data is that while the iterative server can be quick with 10 to 15 requests, after that amount, the time starts getting exponential and will take longer to serve requests as it is flooded with them.

# Lessons Learned

Many lessons were learned in the coding of the client program. While we had experience with multithreading before, we never had problems with using non-thread safe java classes such as the basic Integer. The basics of thread safety is only allowing one thread to access a shared process at a time. For example, total turnaround time is calculated by summing all client turn-around times. But when a client would finish and add their times, another client could be accessing the variable at the same time and write on top of the other client. This would cause very inaccurate total turnaround times. To solve this, we looked up some different thread safe methods and eventually went with the Atomic Long java class. This class provides a thread safe long type variable to use. Other methods used included a token counter that signified if a client was finished or not. This was used for preventing the user prompt from running before all the threads were finished. However, we also had to make this variable thread safe.