CSC358 A1 Report

February 11, 2022

httperf Testing Data

Simple Server

Number of Connections	Connection Rate	Connection time (avg)	Connection length	Reply rate	Reply time	Reply size	Reply status
100	1	1.2	1	1	response: 0.9 transfer 0.1	72492.0	2xx=100
100	100	1.4	1	10	response: 1.1 transfer; 0.2	72492.0	2xx=100
1000	100	4.4	1	98.4	response: 2.3 transfer: 0.4	72492.0	2xx=1000

Persistent Server

Number of Connections	Connection Rate	Connection time (avg)	Connection length	Reply rate	Reply time	Reply size	Reply status
100	1	1.2	1	1	response: 0.8 transfer: 0.2	72492.0	2xx=100
100	100	1.2	1	10	response: 0.9 transfer: 0.2	72492.0	2xx=100
1000	100	2.9	1	100	response: 2.2 transfer: 0.3	72492.0	2xx=1000

Pipelined Server

max-piped-calls = 10

Number of Connections	Connection Rate	Connection time (avg)	Connection length	Reply rate	Reply time	Reply size	Reply status
100	1	1.5	1	1	response: 1.1 transfer: 0.2	72493.0	2xx=100
100	100	1.3	1	10	response: 1.0 transfer: 0.2	72492.0	2xx=100
1000	100	2.2	1	100	response: 1.3 transfer: 0.2	72493.0	2xx=1000

Apache Server

max-piped-calls = 10

Number of Connections	Connection Rate	Connection time (avg)	Connection length	Reply rate	Reply time	Reply size	Reply status
100	1	0.7	1	1	response 0.5 transfer 0.1	72493.0	2xx=100
100	100	0.9	1	10	response 0.6 transfer 0.1	72492.0	2xx=100
1000	100	1.0	1	100	response: 0.6 transfer: 0.1	72493.0	2xx=1000

Notes

When choosing the number of connections and connection rate to use as options with the httperf tool we wanted to find a combination of values that gave us the biggest breadth of cases to simulate a real world situation, while also keeping our performance testing concise and clear. The first test is an example of 100 sequential connections to our server which would be rare in a real life situation. The second test is an example of 100 connections and requests being created at once. This exhibits a low scale load test to view how the server handles many connections at a single moment which is very common. Lastly, this is very similar to the last test which multiplies test two by 10 (100 concurrent connections, 10 times). This displays how each server can handle dealing with high loads of connections and requests sequentially. This is explained in more detail below.

Individual Server Comparison

This section will cover the logic behind each server and how the tests within each server compare to each other and the reasons behind the differences.

Simple Server

Logic

Simple server is a simple HTTP server that supports the HTTP 1.0 protocol. It can handle .txt, .cc, .html, .jpg and .js file extensions as well as conditional get requests.

For this Server a simple Server architecture is used where a single process loops continually until a connection is made with the server. Once a connection is made, a thread is spawned to deal with the connection (a single request). This is done such that concurrent requests can be handled without excessive queuing overloading of a single process. Once a request is processed, the socket is closed and the thread returns.

Tests

For the first performance test, the number of connections is 100 and the connection rate is 1. This means that 100 connections will be created sequentially one after another. For this test, we can expect the connection time (the average time of the full lifespan of a connection) to be unaffected. This is because connections are not being queued up and are being made sequentially. Furthermore, only one thread is being created at a time, therefore response time should be at its lowest across all three tests

For the second performance test, 100 connections are being made all at the same time. Based on the httperf data, it is evident that this caused an increase in the average lifespan of a connection. This is because with more concurrent connections it is highly probable that requests were queued up and threads were not created fast enough to process all concurrent requests. Furthermore, the reply time (time from request sent from sender to response received from sender increased slightly. This is likely due to the increased load the server is dealing with at a single moment.

For the second performance test, it is clear that the consecutive load on our simple server slows down the response time and connection time. This is likely due to the increase in queuing caused by the sequential load of 100 connections per second.

Persistent Server

Logic

The logic for this server is very similar to Simple Server, however, it is able to handle persistent connections. Once a thread is spawned to deal with a certain connection, the thread will then process requests from this socket, until a custom timeout time has passed. For our server, it is 5 seconds. Once this time has passed and nothing has been available to read from the socket, the thread returns and a new connection must be created if the user wants to send more requests.

Tests

For the first test, we are dealing with 100 unique connections, thus, it is clear that adding persistence should not have a large impact. However, on each request, a file is being requested that has references to other files on the server. Thus using a persistent connection here for each connection can reduce our connection time by reducing the overhead of closing and reopening connections for each object we need from the server.

For the second and third tests, the response time increases significantly which is likely due to the increase in processing for 100 concurrent connections compared to test one. Although threads are being used to handle each unique connection, the process which creates the threads may be overloaded when too many connections are created at the same time.

Pipelined Server

Logic

The pipelined server is able to handle pipelined requests in a pipelined fashion. The logic of handling a request is the same in this server when compared to the two previous servers. However, when a thread is spawned to deal with a connection, this server now deals with the case where this client may have sent multiple requests, and it must deal with all of them. To do this, a thread is spawned with an entry function such that it loops over all of the requests sent by the client and responds to all of them in a pipelined fashion. That is, while there is some data to read, process the request and respond accordingly.

Tests

For the pipelined server performance tests, the max pipelined requests was set to a max of 10.

For the first test, we can expect the connection and reply time to be unaffected as requests are made sequentially. Thus the first test does not make use of pipelining requests which the server was modified for.

For the second and third tests, it is clear that the pipelined server is very effective when comparing response times. Since 10 requests are pipelined together, the response time shown in the httperf performance data is the response time for 10 pipelined requests. Thus, it is clear that the response time for a single request is greatly decreased when handling multiple requests

at a time in a pipelined fashion, rather than a single request at a time (simple server)/single requests with persistence (persistent server).

Apache Server

Logic

The Apache web server deals with connections very similar to how the pipelined server does. For each connection made, the apache web server creates a new thread to process the requests from the thread. Furthermore Apache also supports pipelined requests. Similar to pipelined server, Apache processes pipelined requests sequentially.

Tests

See **Cross Server Comparison**

Cross Server Comparison

This section will cover the differences in test data between servers and how their logical differences caused these distinctions.

Simple Server

The Simple Server has the highest response time and connection time when comparing it to all other tests at 2.3ms. This is because it does not support persistent connections. Thus, once the index page is grabbed, a new connection must be created to grab all other resources. In this case this includes images, CSS and JavaScript.

When comparing Simple server tests to Persistent Server tests, the reply times for persistent server are slightly lower. This is because instead of a new request being processed by the main process by spawning a thread, in persistent server, when the subsequent requests are made, the thread is already spawned and is ready to reply. Thus this can cause higher efficiency when responding to requests.

When comparing Simple server to pipelined server, it is clear that the reply times for the pipelined server are more consistent, while the reply times for the simple server have a much larger range. This is because the pipelined server does a much better job at handling large loads of requests (it is set to a max of 10). Thus from test one to test three, it is evident that the response times for the pipelined server is steady and does not have a high variance. However, the Simple server cannot handle persistent connections and responds to requests one at a time. Thus it cannot handle a high load, and its reply time seems to grow exponentially.

Persistent Server

Persistent Server is able to reduce its response time by grouping requests within a certain timeout period onto the same thread and connection. However, the growth of response time does seem to grow at the same rate as the Simple Server as the tests we use request a single object with multiple other files. Thus the overhead that a persistent server avoids by supporting persistent connections seems to be a constant in our case, but may not be the case in a real world situation where different files can depend on many other files, varying on each request.

When comparing persistent server to pipelined server, because of the points mentioned above, it is a very similar comparison to that of simple server. Pipelined server is much better at dealing with high load as a group of requests can be processed on a single thread without having the client wait to request them one at a time. It is also clear that the connection time for each client is reduced as, a connection does not have to be open for RTT*num_files, but rather, in some cases one or two round trips can deliver all necessary files to the client.

Pipelined Server

As explained above, the pipelined server is much better at dealing with high loads of connections which shown by the data leads to much more consistent reply times. Furthermore, it is important to note that the connection times are on the larger side because when processing pipelined requests, the connection stays open until all pipelined requests are processed and responded to. Thus, the connection time may be larger than other servers depending on the amount of pipelined requests/files being requested.

Apache Server

The Apache server has much lower connection times and response times than the pipelined server. This is expected as it is an open source HTTP server that has been in development for over 20 years. Therefore there are many reasons that could lead to these differences. The first is the timeout time. Apache likely uses a dynamic timeout time for each connection using sampleRTT, estimatedRTT and devRTT. This can ensure high efficiency when dealing with lost packets and determining how long to wait before closing a connection. Furthermore, over its development, Apache has likely optimized processing requests and formulating responses which may not be the case in our pipelined server.

Other Notes

In general, conditional GET requests make significant developments from HTTP 1.0 to 1.1. For 1.0, they provide the bare bones functionality of requesting a file with some condition that the file being requested should satisfy. The condition is specifically querying the date of file's modification, based on which it responds accordingly, giving rise to the response type '304 Not Modified'. 1.1 further builds on this functionality, by adding the header fields: If-Unmodified-Since, If-Match, If-None-Match, and If-Range, to the existing If-Modified-Since field. As the name might suggest, If-Unmodified-Since checks if the requested file has not been modified since its date value, so the condition is met when this date is strictly older than the file's last modified date. Similarly, If-Match accepts potentially a list of such validating features which are known as 'entities'. For simplicity's sake, our server expects this header to hold only 1 entity, a sha256 checksum, further defining our protocol and limiting the header's values to '*' or a sum. A star loosens the condition, allowing it to mangle with the outcomes of other conditions, extending the complexity of a response. Furthermore, If-Match's counterpart, If-None-Match allows the sender to specify that the request should succeed only if all the field's entities don't match, and further builds on the complexity of a request. On the contrary, If-Range adds unique functionality where its value is specified by the sender when the sender has a partial copy of an entity and wants the entire entity, which may lead to getting the entire entity if the partial copy's version is older. If-Range was not implemented for this server, but all others were.

Run Server Instructions

1. CD into root directory
2. Run `make`
3. Run each Server by running
    a. ./SimpleServer <port number> <root directory path>
    b. ./PersistentServer <port number> <root directory path>
    c. ./PipelinedServer <port number> <root directory path>