Performance Testing Report

Vipul Taneja - tanejav1

William Li - liwill13

Eric Galego - galegoer

How the Servers Work

Simple Server:

Our simple HTTP server will begin by looking for connections, accept an incoming connection from a client and process the clients request(s). If the simple server receives a HTTP 1.1 request, we handle the request as though it were an HTTP 1.0 request, which means our server sends back a HTTP 1.0 response. After handling a single request from the client, our simple server, unlike Persistent and Pipelined Server, will close the connection between server and client. Afterwards, the server will resume into a listening state, in which the client must connect and have the server accept the connection again before the server can handle the next request. This specified behavior is what would *theoretically* cause our simple server to be slower than our other implementations when handling a multitude of requests. However, the httperf num-calls option, which specifies the number of requests per connection, does not work for simple server, so simple server when tested with 10,000, 100,000 and 1,000,000 connections actually runs slightly faster than our other implemented servers.

Compared to Apache Simple Server, our simple server actually runs slightly faster than Apache. We believe this is due to the fact that our simple server has less functionality than Apache. For example, we did not implement functionality to handle every single conditional request. In this way Apache is running more code in the background and is shown to work better when we increase the number of requests per connection and increase connections in general.

Persistent Server

Our persistent server will also begin by looking for connections, and accepting an incoming connection from a client. Unlike the simple server the connection between

server and client is not closed between the handling of requests. That is, it is expected for the persistent server to be able to handle a multitude of requests faster than the simple server because there is no additional RTT caused by the reconnections between the handling of requests. Our persistent server, unlike the simple server implementation, can now handle HTTP 1.1 requests along with 1.0. HTTP 1.1 requests from the client, unless specified by the Connection header, will be handled as a persistent connection. HTTP 1.0 requests will be handled exactly like the simple server (i.e close connection), unless the Connection header specifies that we should keep the connection alive. This means our implementation also assumes that an HTTP 1.1 request can specify for the connection to be closed instead of remaining persistent and our server should close the connection accordingly.

Looking at the test results, because we set num-calls=1 when comparing our persistent server with our simple server, each connection will only send 1 request before closing the connection. In this case, it makes sense to see no significant difference in speed between the two implementations because they are technically handling each connection the same way. (i.e accept connection, handle 1 request, close connection).

Comparing our persistent server results with Apache's persistent server, our persistent runs significantly faster for 100,000 and 1,000,000 connections with 2 requests per connection. However, it seems our persistent scales well based on number of connections but extremely poorly based on how many requests each connection has, as seen when our server takes ~132 seconds to run 1000 connections with 5 requests per connection as compared to Apache's 0.2 seconds.

Pipelined Server

Our pipelined http server looks for connections similar to our persistent server. After it accepts a connection with a client, our pipelined server will create a new thread to handle the client's requests. After creating the thread, the server will proceed to handle client requests in a pipelined fashion. If the client has 3 requests, it will handle each request in order. Despite adding threads to simulate pipelining in our pipelined server, we believe that the overhead caused by the creation of threads is why our pipelined server is slower than persistent.

Comparing our pipeline with Apache's pipeline, Apache runs significantly faster in all scenarios. In fact, our pipeline is extremely slow when dealing with the case where we have 1000 connections and 5 requests per connection. We believe the issue may be due to the fact that our pipelined server does not limit the amount of threads that can be created; however there are timers set on these threads to close them after being inactive for some time. We believe another issue is that the pipelined server is based on

our persistent server implementation but with the addition of threads. Since a similar time issue occurs in our persistent server (in regards to multiple requests per connection), this issue was most likely carried over when we tested our pipelined server.

Issues Section

We ran into many issues with our servers runtime and memory so we utilized valgrind to detect any memory leaks and resolve them. We got a report of 0 memory leaks in our final version of the code. This is how we cleared out all of these issues.

We did not get a chance to implement eTags due to time constraints however if-modified / if-unmodified headers are currently implemented.

Testing with Httperf

httperf --server localhost --port 8089 --uri /test.html --rate 150 --num-conn 27000 --num-call 1 --timeout 5

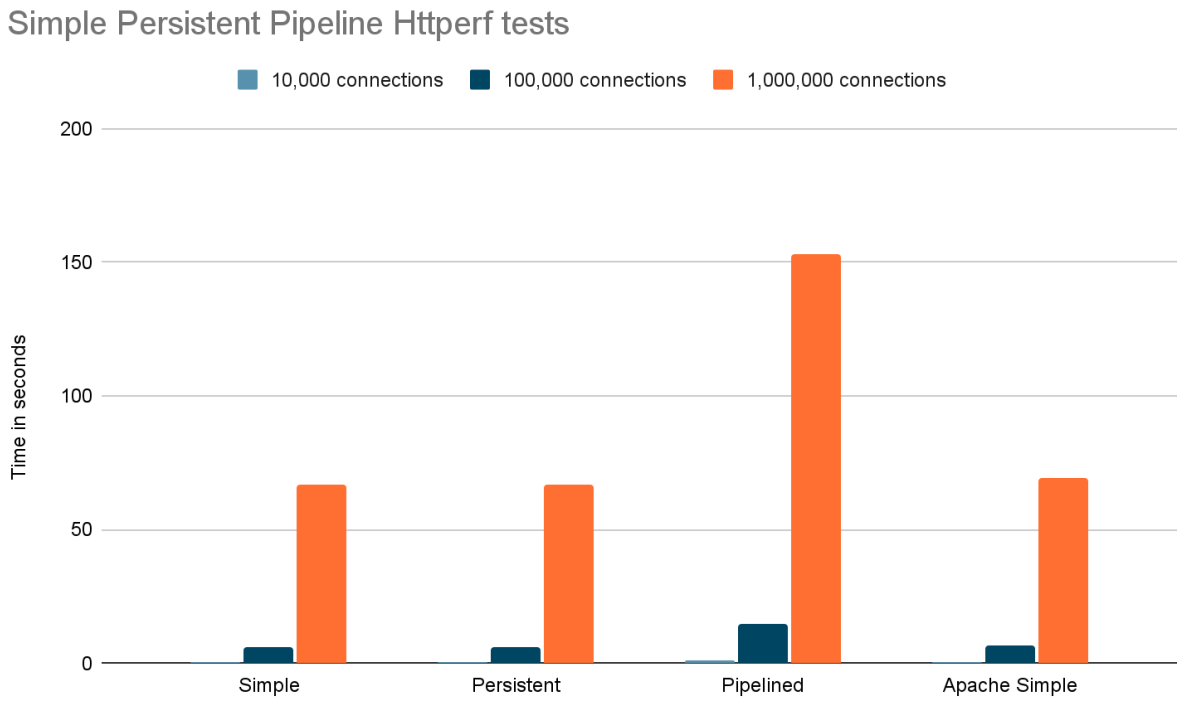
We used the following httperf commands with some argument adjustments to test our web servers. This command accesses the web server on the host with IP name *localhost*, running at port 8089. The web page being retrieved is ``**/test.html"** and the same page is retrieved repeatedly. The rate at which requests are issued is **150 per second**. The test involves initiating a total of **27,000 TCP connections** and on each connection one HTTP call is performed (a call consists of sending a request and receiving a reply). The timeout option selects the number of seconds that the client is willing to wait to hear back from the server. If this timeout expires, the tool considers the corresponding call to have failed. Note that with a total of 27,000 connections and a rate of 150 per second, the total test duration will be approximately 180 seconds, independent of what load the server can actually sustain. (Quoted from : https://www.hpl.hp.com/research/linux/httperf/wisp98/html/doc003.html)

Our results using Httperf

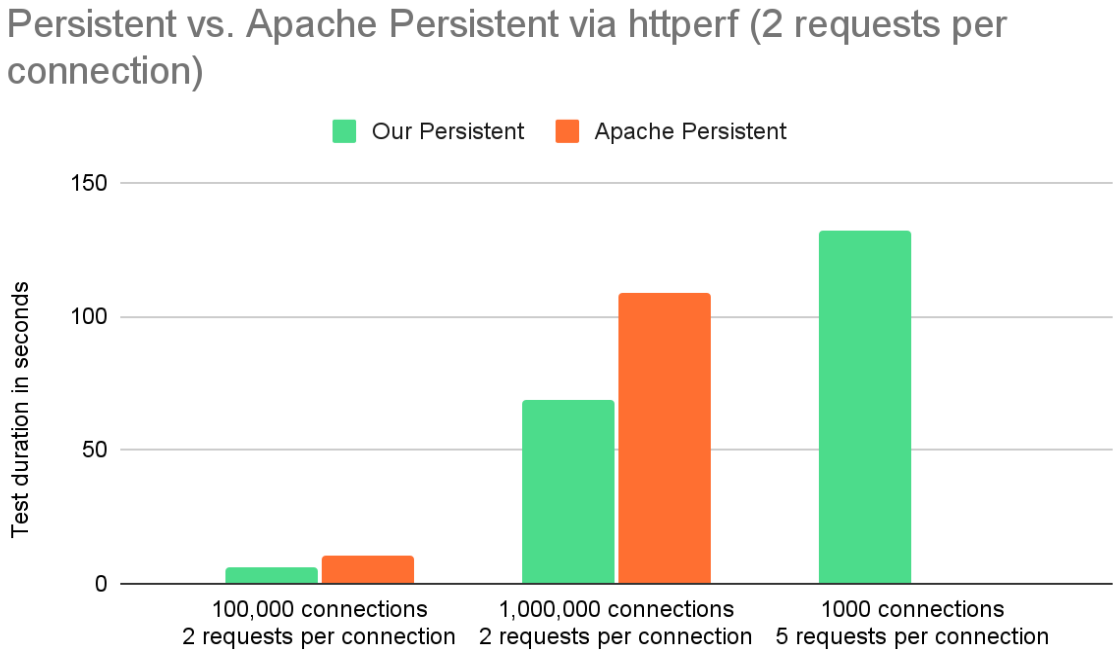
In the below bar graph we have the results of the time it took for these number of connections within each of our 3 implemented servers as well as Apache. As mentioned we believe Apache is slightly slower because our simple server has much less functionality than Apache. This would explain the difference and we have also noticed it is relatively small.
We have the following times for 10,000 connections which could not be accurately described within the graph however they indicate to us a similar pattern with the other results in the below bar graph.

Simple	Persistent	Pipelined	Apache Simple
0.303 s	0.313 s	0.883 s	0.493 s

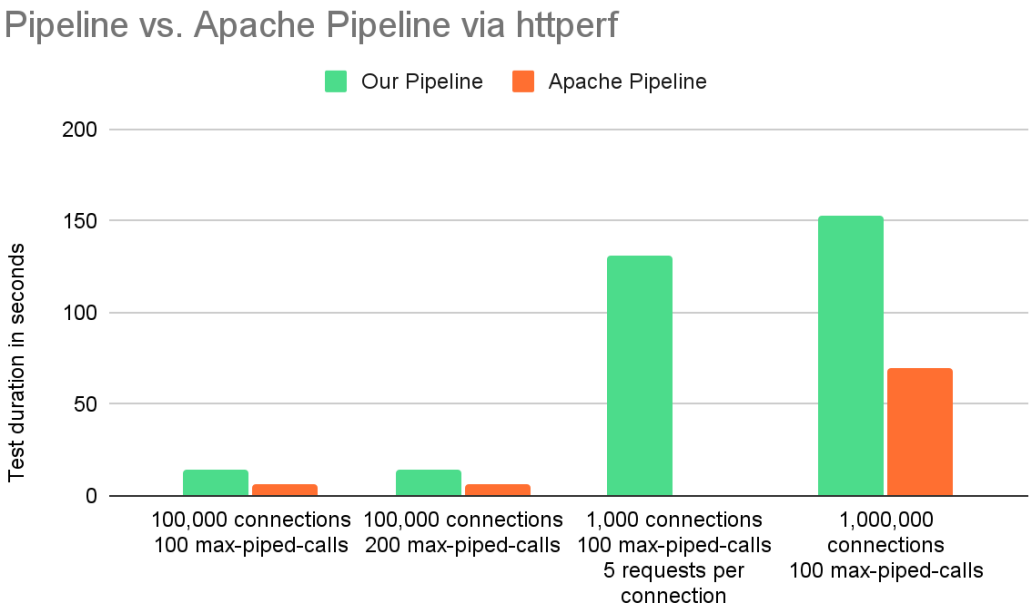
And lastly we believe that despite adding threads to simulate pipelining in our pipelined server, the overhead caused by the creation of threads may be why our pipelined server is slower than persistent.



In our persistent server comparison we can see that with 2 requests our server still does significantly better than Apache, however with the increase to 5 it does significantly worse. Refer to the explanation under persistent server regarding our belief in why this is occuring.



In our pipeline server comparison we can see the substantial difference between our pipeline server and the Apache server. When there are multiple requests per connection it is again significantly slower, similar to our persistent server which we believe is due to an issue carried over from persistent. (As both use relatively similar code)



Works Cited

References for our implementation:

https://www.geeksforgeeks.org/socket-programming-cc/

https://stackoverflow.com/questions/1858050/how-do-i-compare-two-timestamps-in-c

https://developer.mozilla.org/en-US/docs/Web/HTTP/Conditional_requests

https://stackoverflow.com/questions/11952898/c-send-and-receive-file

https://www.binarytides.com/server-client-example-c-sockets-linux/

https://stackoverflow.com/questions/14111524/how-to-set-sscanf-to-ignore-characters-at-start-of-string