COMP 8005

Assignment 2 Report

By: Derek Wong

Instructor: Aman Abdulla

Due: 12 PM on March 1st, 2021

Table of Contents

Introduction

The main goal of this assignment is to obtain a measure of scalability and performance of an epoll-based client-server implementation. I gathered a record of statistics on the number of requests, total bytes sent/received, total elapsed time for a full client-server data exchange, and average time for each request to be serviced from the client from various test cases. Along these results, I compared them to Wireshark statistics to analyze and verify the different conditions in which the epoll-based client-server implementation is more scalable and more performant.

Testing

The general idea is to keep adding workload to the server until the server shows a significant degradation in performance. There are two ways to increase workload, either by adding more clients or by sending more frequent requests. At the end of each test case, the result of each test case will be grouped together and plotted into a line graph to show:

1. Average time to service a request over an amount of clients
2. Average time to service a request based on frequency of requests

The Makefile in the “epoll” directory contains a target to execute a test script with user configured parameters.

Test Script

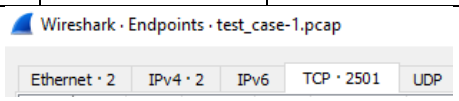
Refer to the User Manual on how to execute the test script.

Test Cases

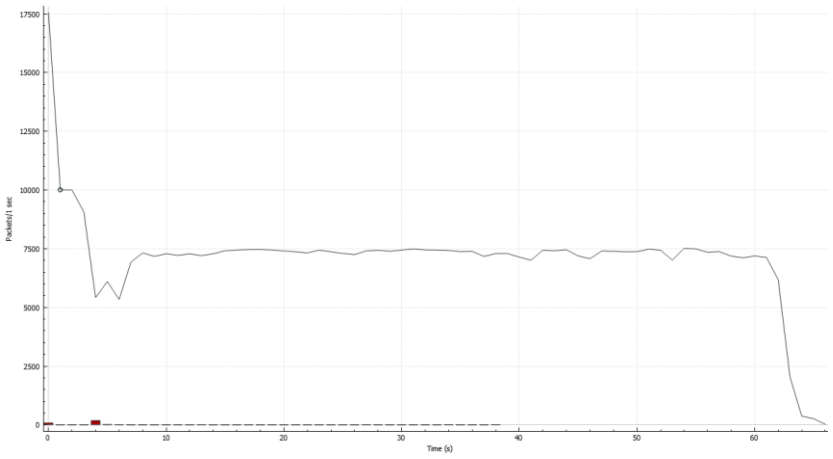
Case 1-5 Results

In these cases, we try to determine the maximum number of active sockets the server can handle in one session. We do this by increasing the number of clients until we can no longer establish all client endpoints on the server and the performance on the server side drops.

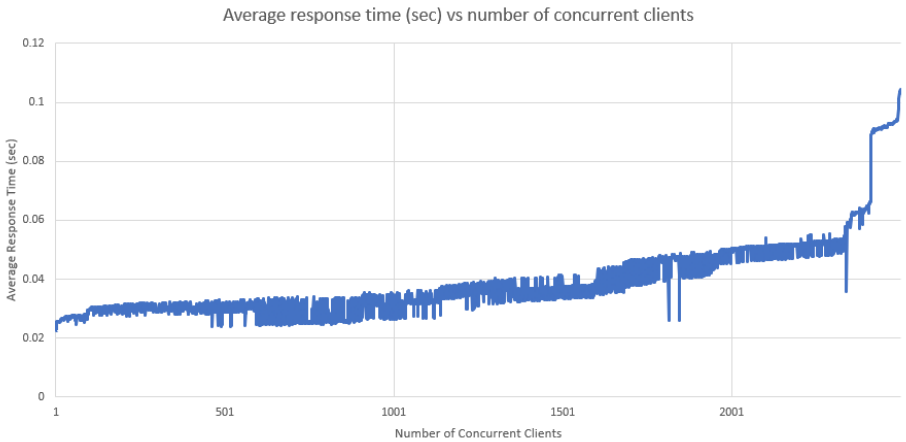
Test Case	Number of clients	Request/Client	Requests Delay (ms)
1	2500	60	1000



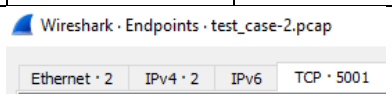
This confirms that we can scale to 2500 clients since the number of TCP endpoints matches the number of clients + server.



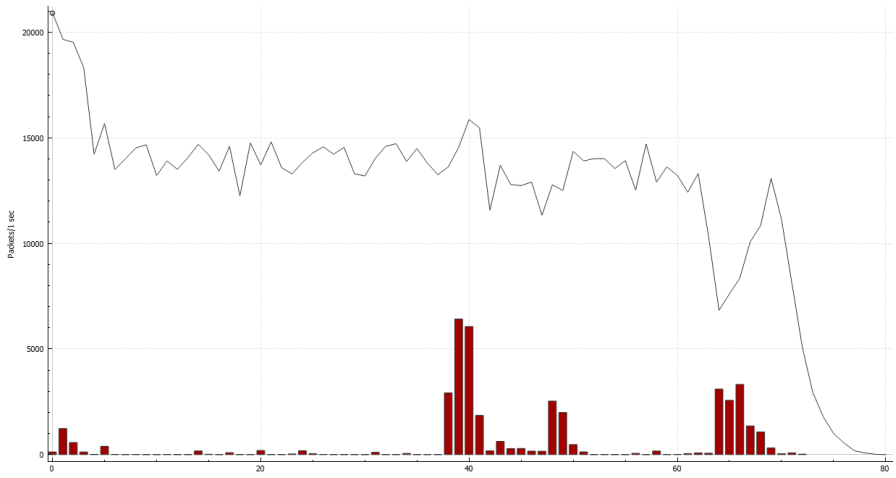
The IO graph is relatively constant with little TCP error flags, aside from the influx of IO in the beginning of the transfer.



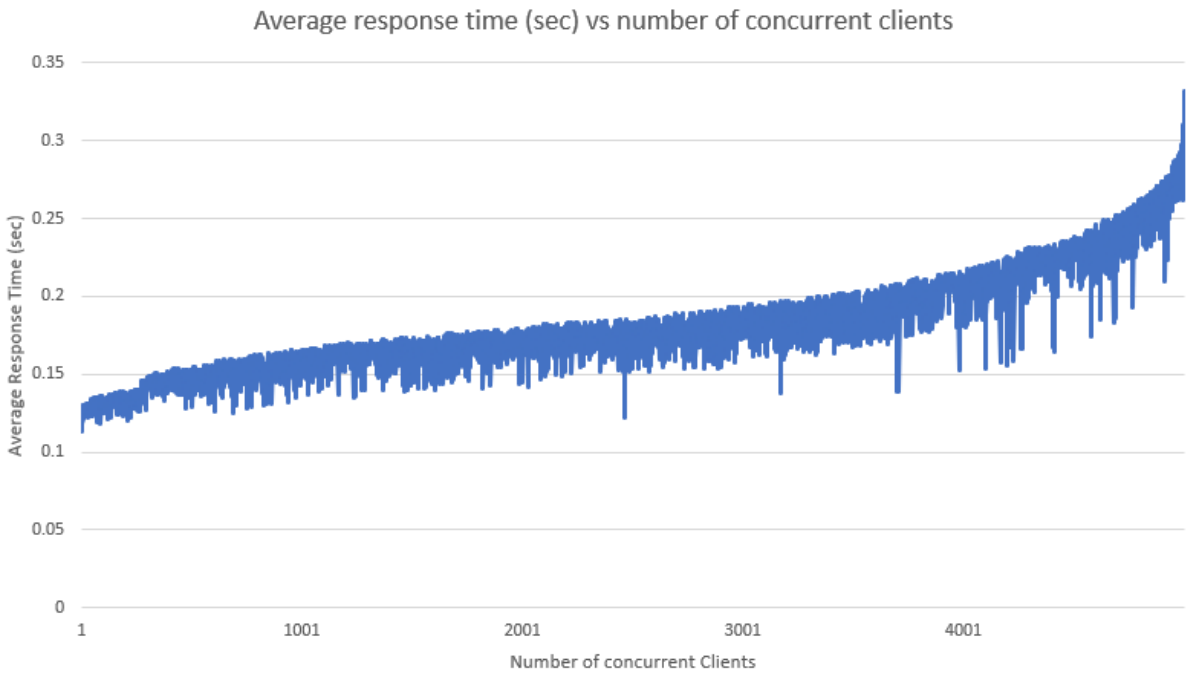
Test Case	Number of clients	Request/Client	Requests Delay (ms)
2	5000	60	1000



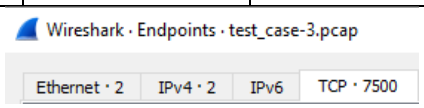
This confirms that we can scale to 5000 clients since the number of TCP endpoints matches the number of clients + server.



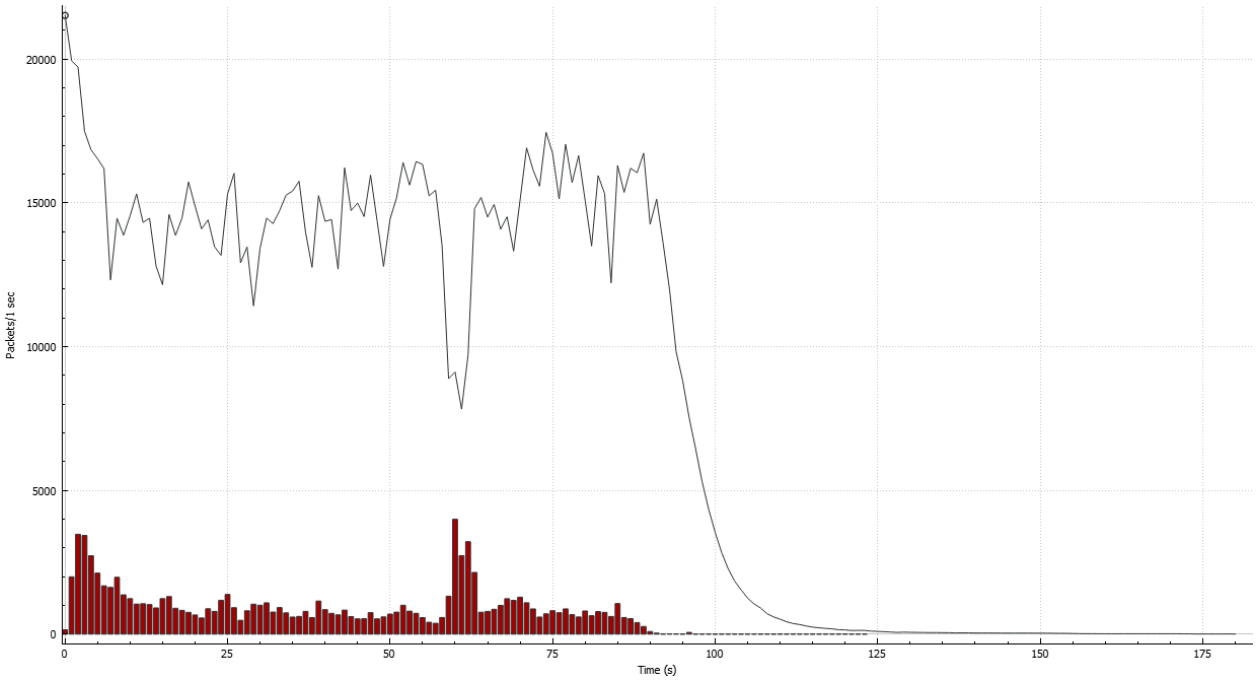
The IO graph has noticeably more variation in IO rate with more TCP error flags, and averages about twice the amount of IO as test case 1.



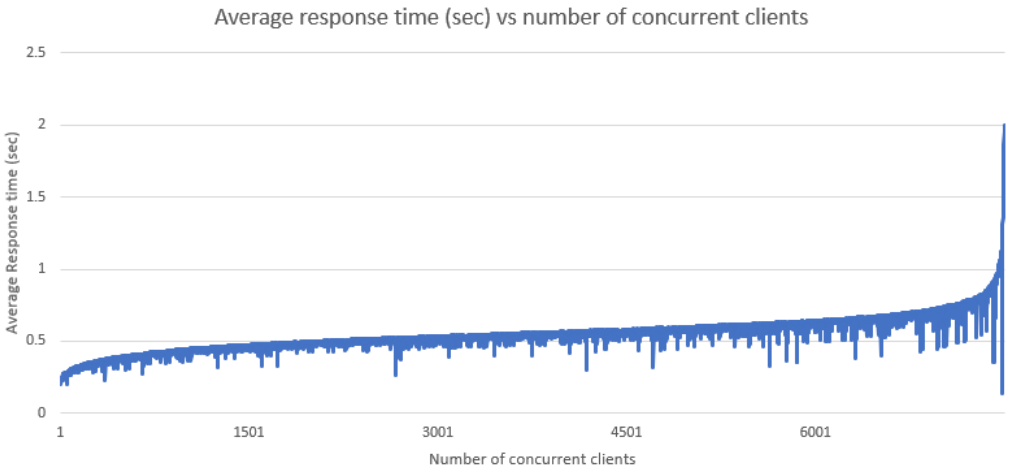
Test Case	Number of clients	Request/Client	Requests Delay (ms)
3	7500	60	1000



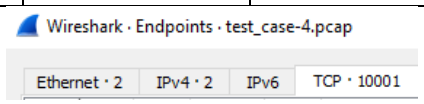
This confirms that we can scale close to 7500 clients since the number of TCP endpoints almost matches the number of clients + server. However, it is more likely that the connection request may not have ever been received given the numerous amounts of TCP retransmissions.



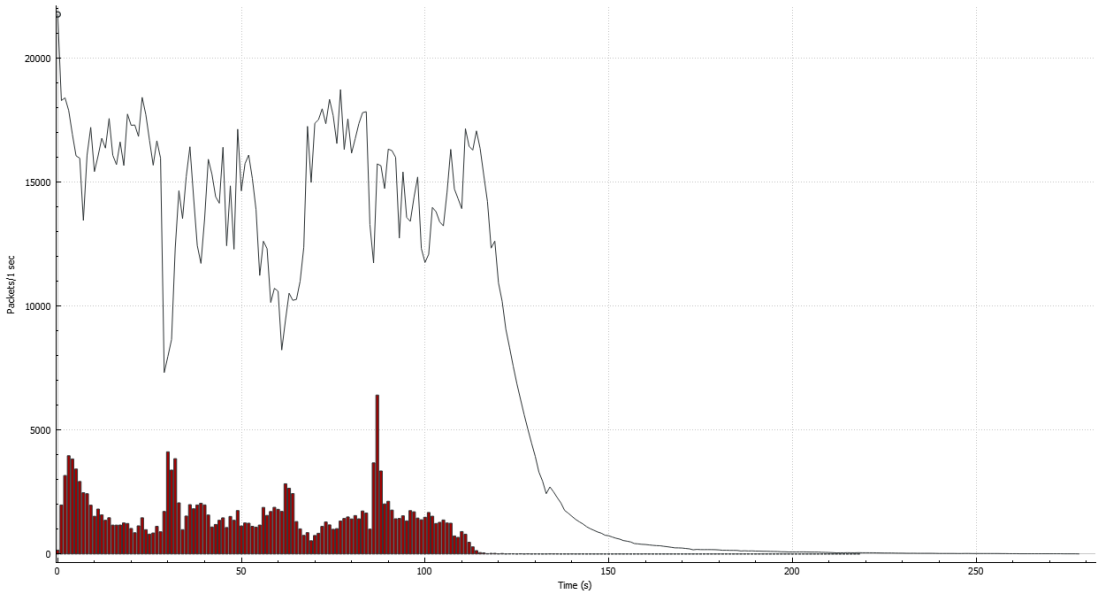
The IO graph has noticeably more variation in IO rate with more TCP error flags but maintains a similar IO rate and the complete data transfer for all clients take significantly longer.



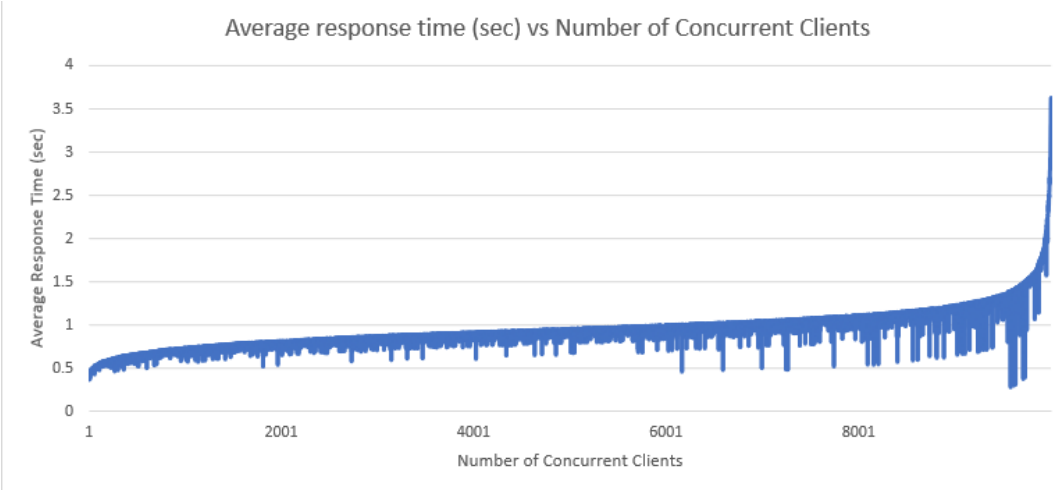
Test Case	Number of clients	Request/Client	Requests Delay (ms)
4	10000	60	1000



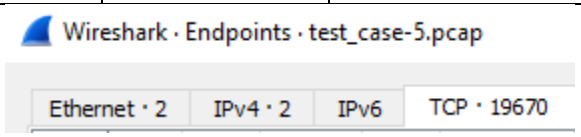
This confirms that we can scale to 10000 clients since the number of TCP endpoints matches the number of clients + server. Since we can achieve this number, test case 3 reporting only 7500 rather than 7501 endpoints is most likely an inaccuracy.



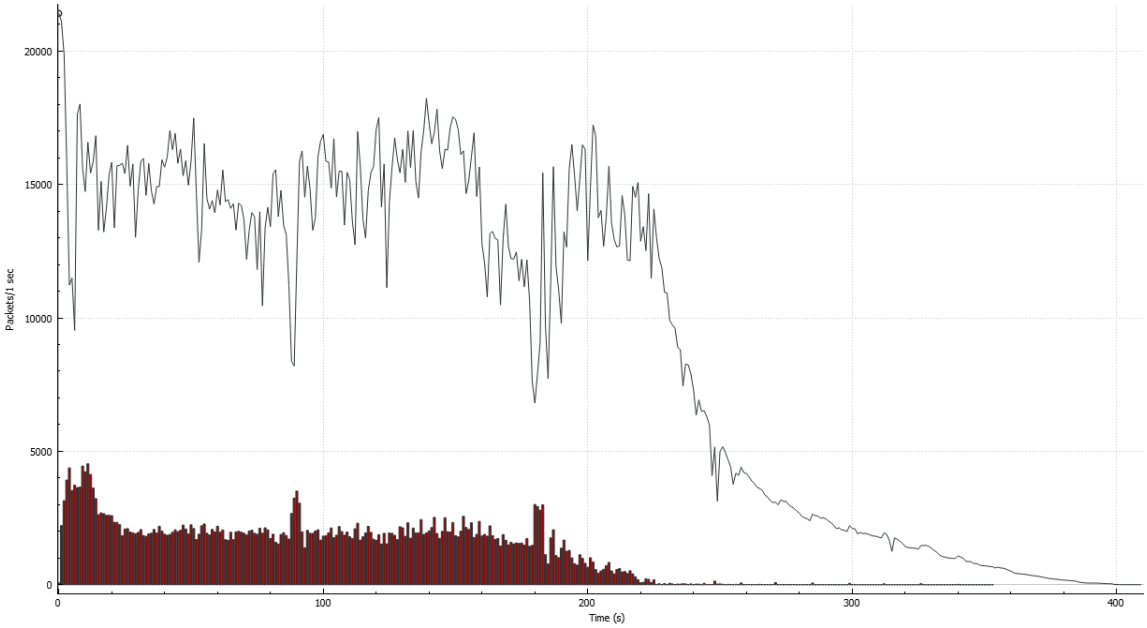
The IO graph has noticeably more variation in IO rate with more TCP error flags but maintains a similar IO rate and the complete data transfer for all clients take significantly longer.



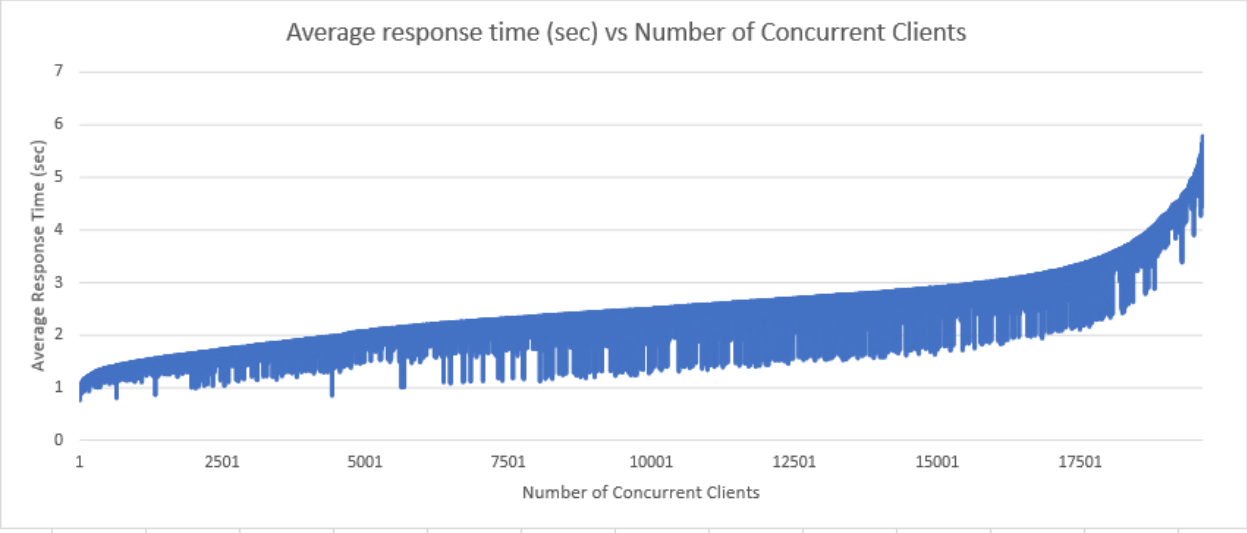
Test Case	Number of clients	Request/Client	Requests Delay (ms)
5	20000	60	1000



This confirms that we can scale to close to 20000 clients concurrently, with few clients never establishing connection.



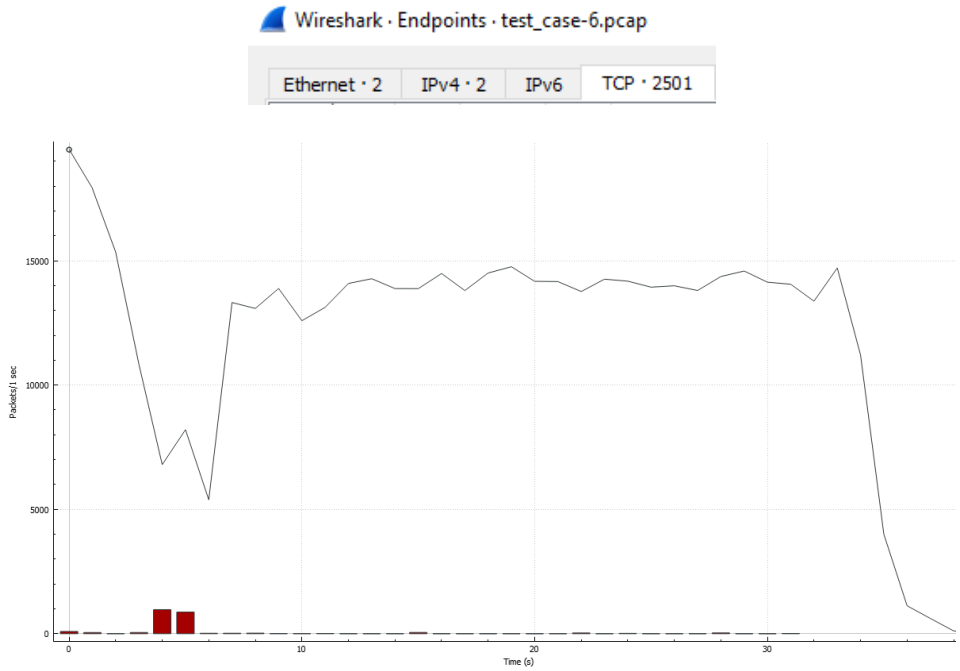
We should note that the variation in IO rate is significant, changing as much as 50% throughout the wireshark capture.



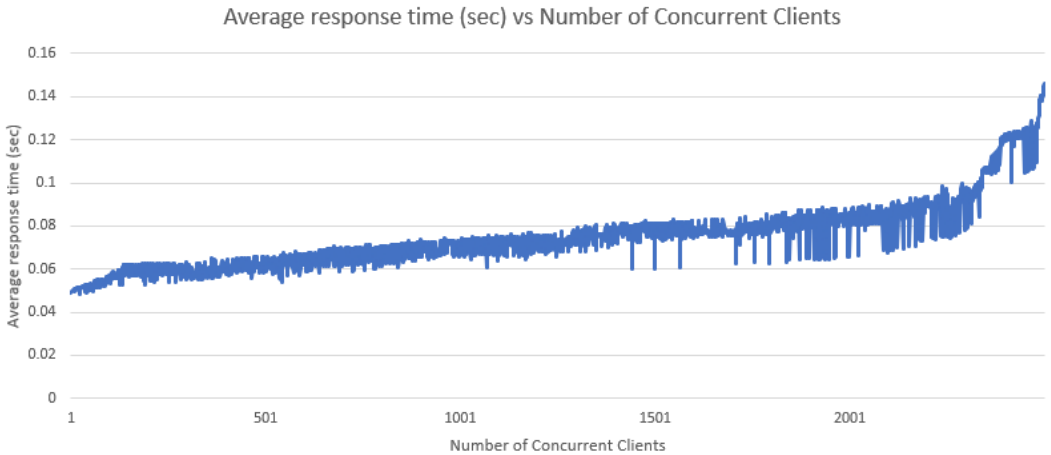
Case 6-11 Results

In these cases, we try to determine the effect of increasing the frequency of requests from each client to the server. We do this by decrease the delay between requests until we can no longer establish all client endpoints on the server and the performance on the server side drops.

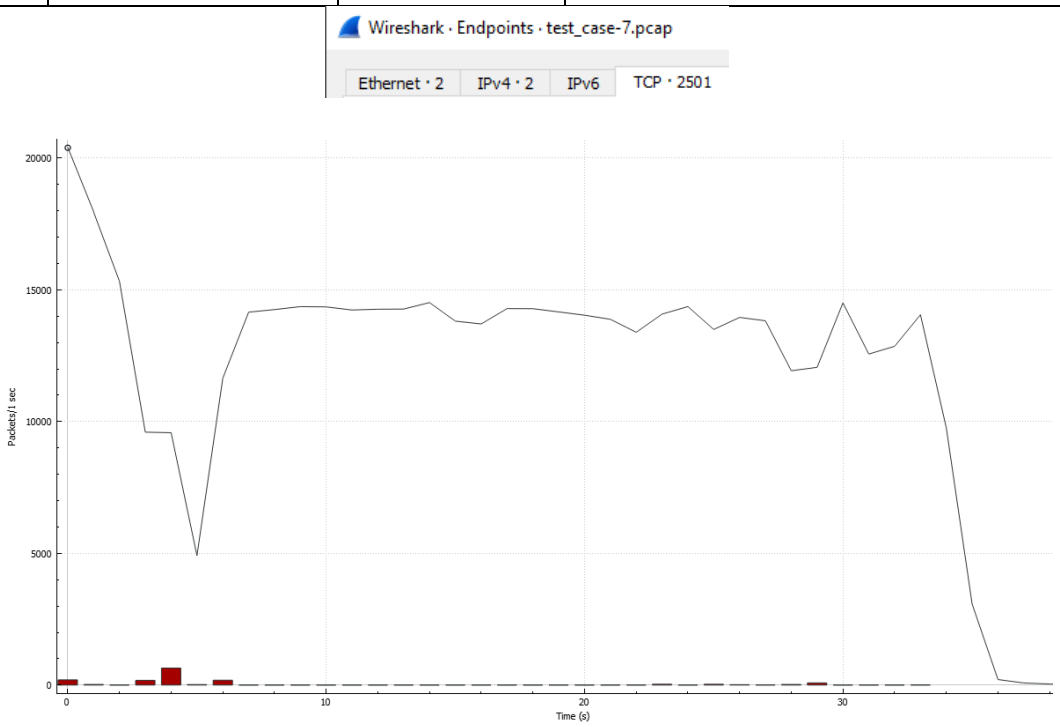
Test Case	Number of clients	Request/Client	Requests Delay (ms)
6	2500	60	500



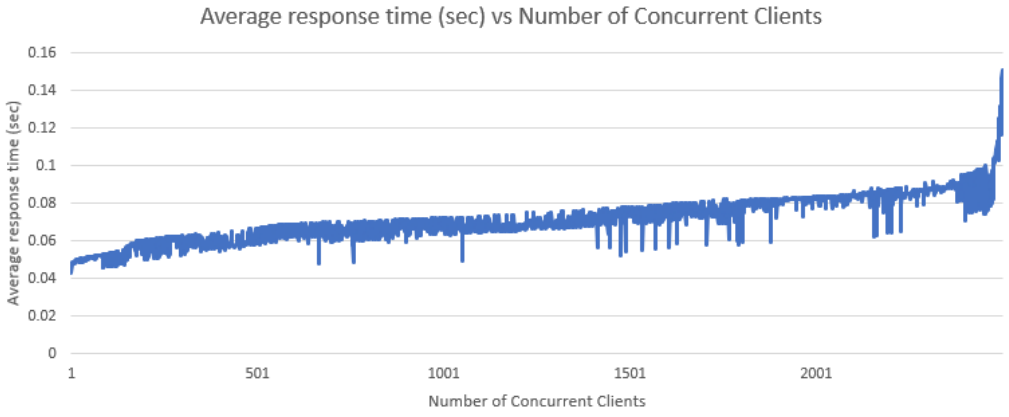
As seen in the previous test cases, the IO rate stabilizes around 15000 packets per second. However, it takes almost half the amount of time to service all requests as seen in test case 1 (60 seconds) with an IO rate of 7500 packets per second.



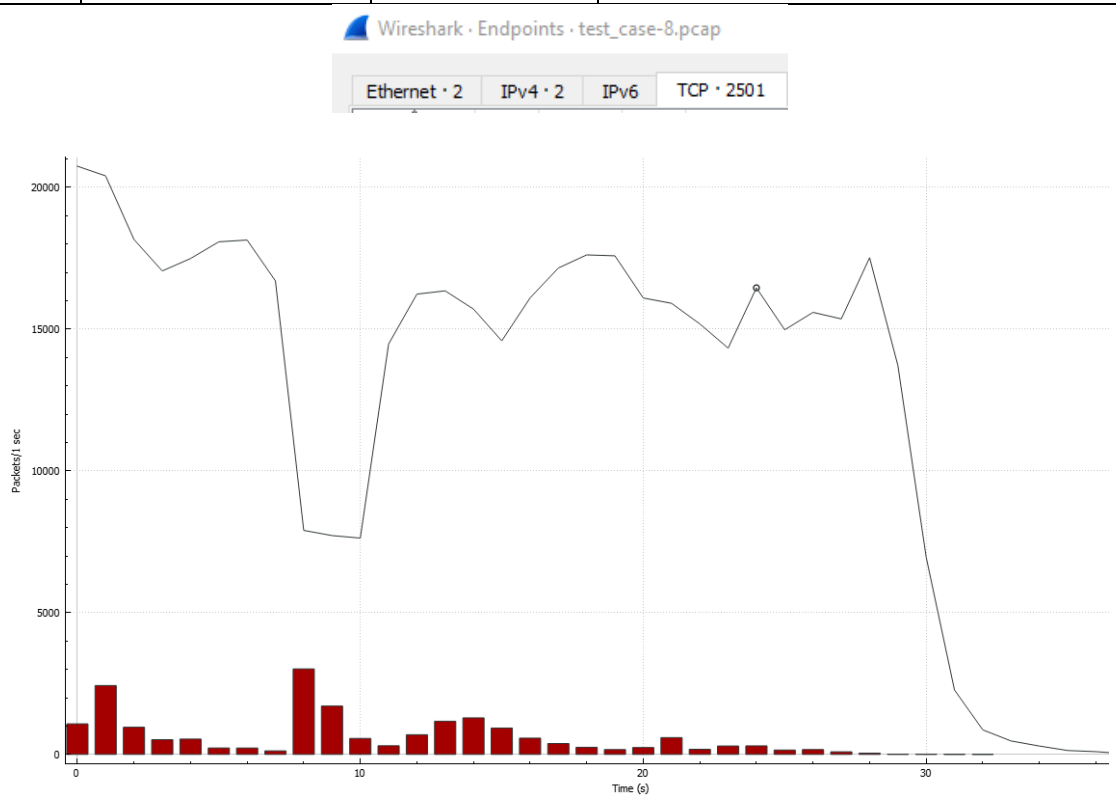
Test Case	Number of clients	Request/Client	Requests Delay (ms)
7	2500	60	250



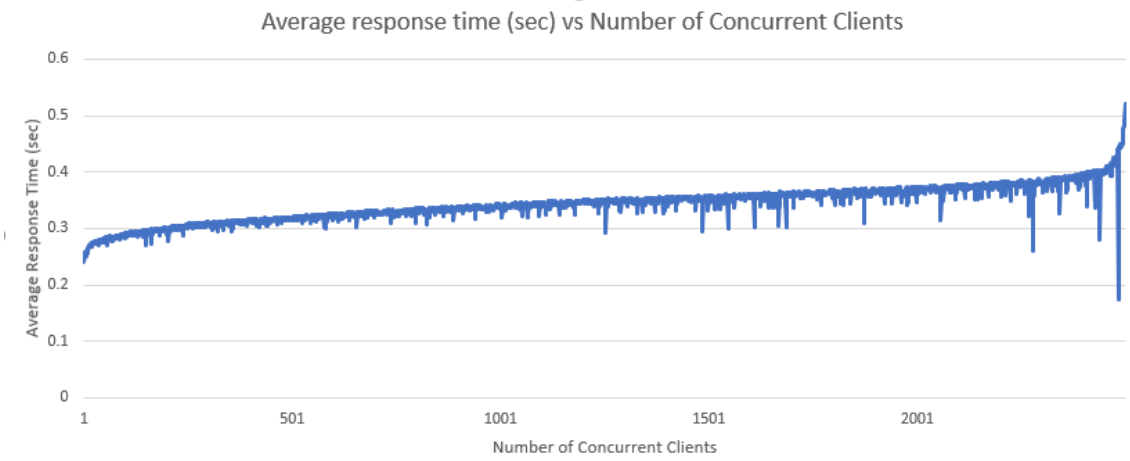
There is no significant performance difference between test case 6 and test case 7.



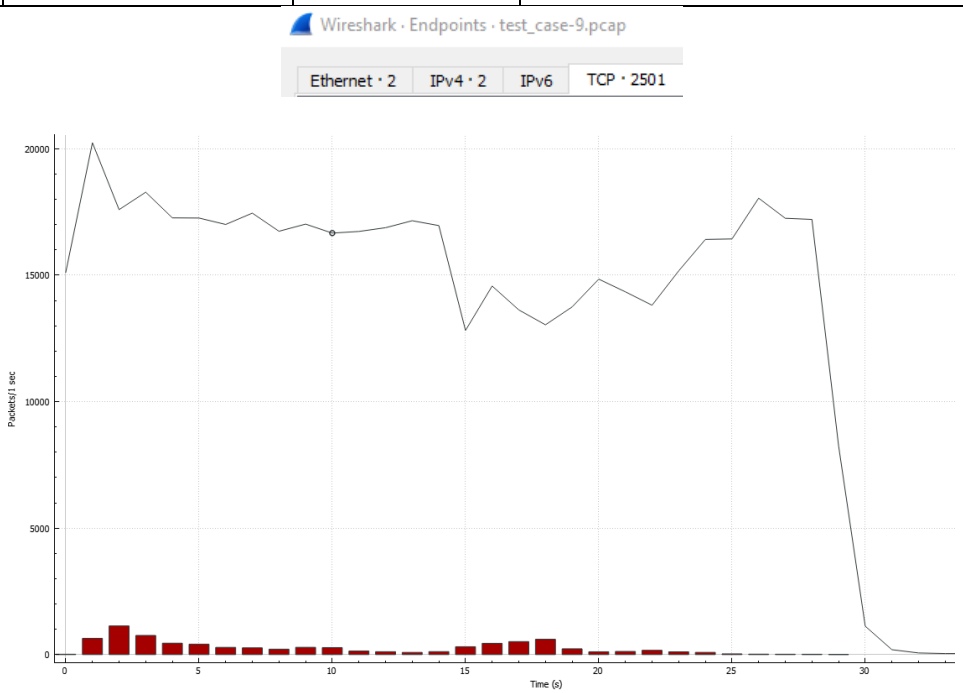
Test Case	Number of clients	Request/Client	Requests Delay (ms)
8	2500	60	125



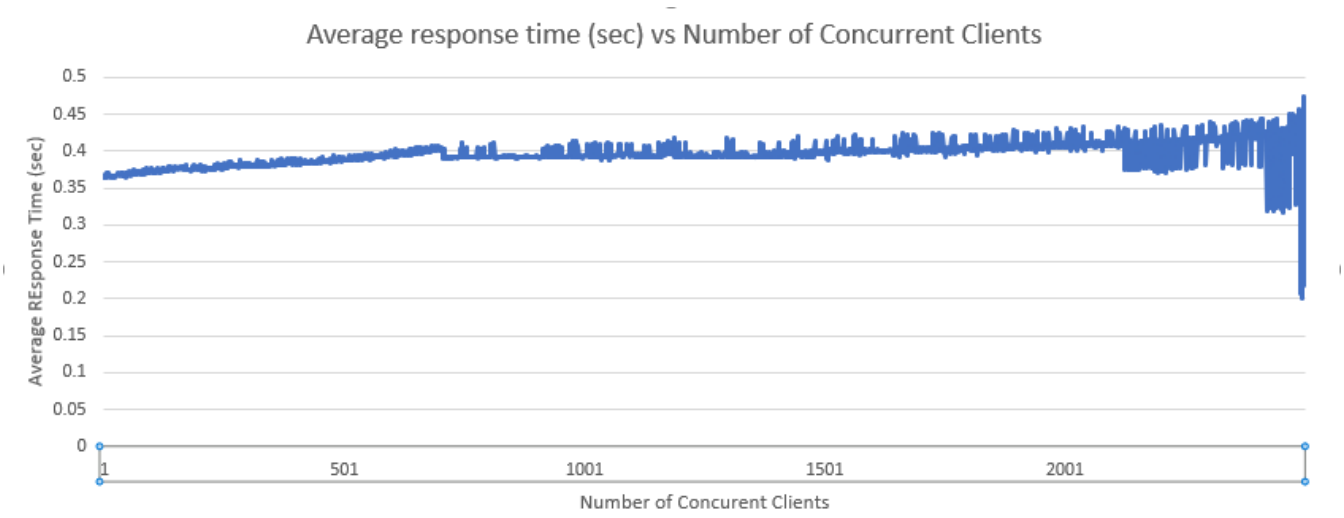
There is an observable increase in IO rate to greater than 15000 packets per second but with greater number of retransmissions. This is likely due to the server not being able to handle all the requests fast enough.



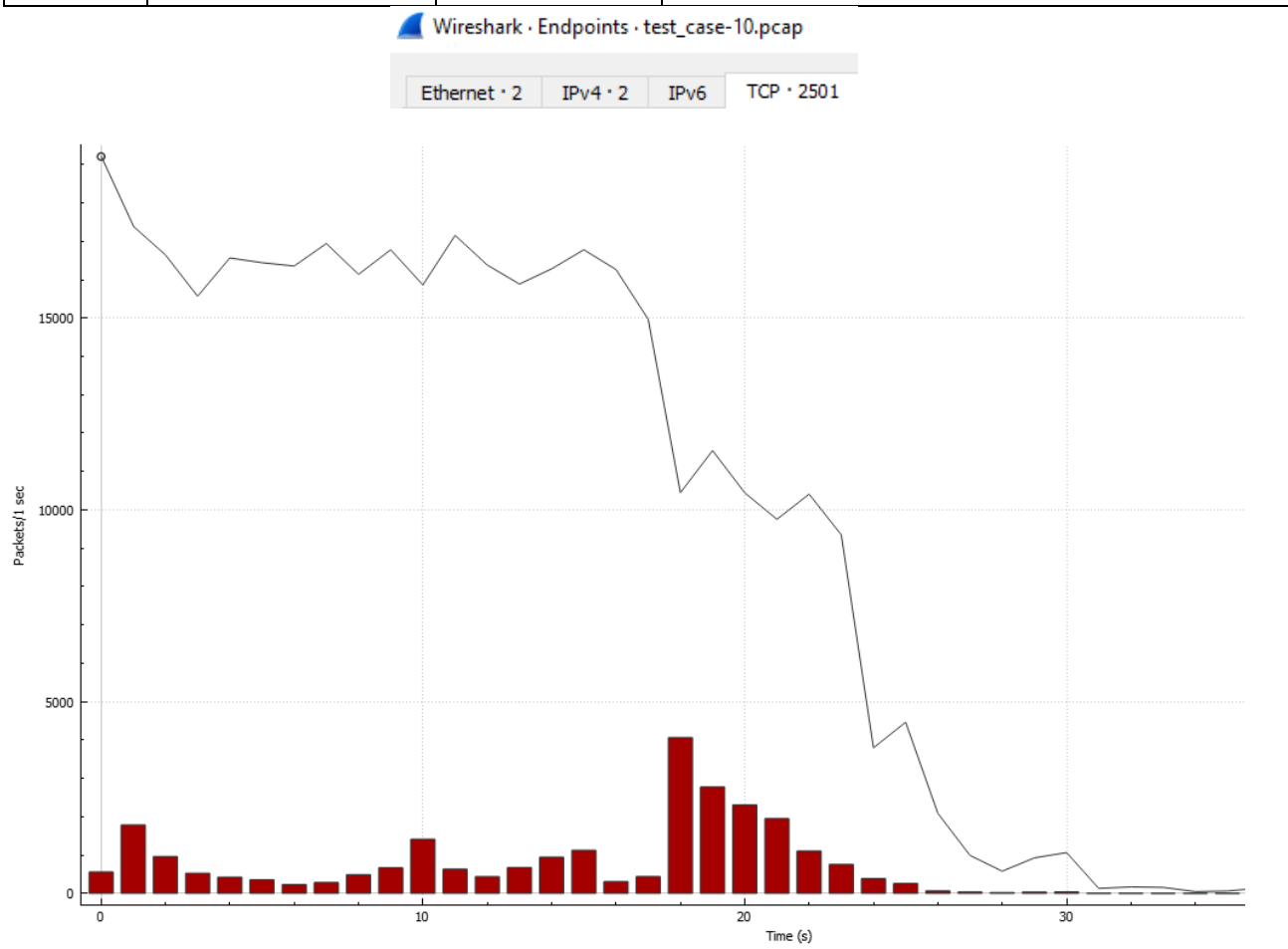
Test Case	Number of clients	Request/Client	Requests Delay (ms)
9	2500	60	50

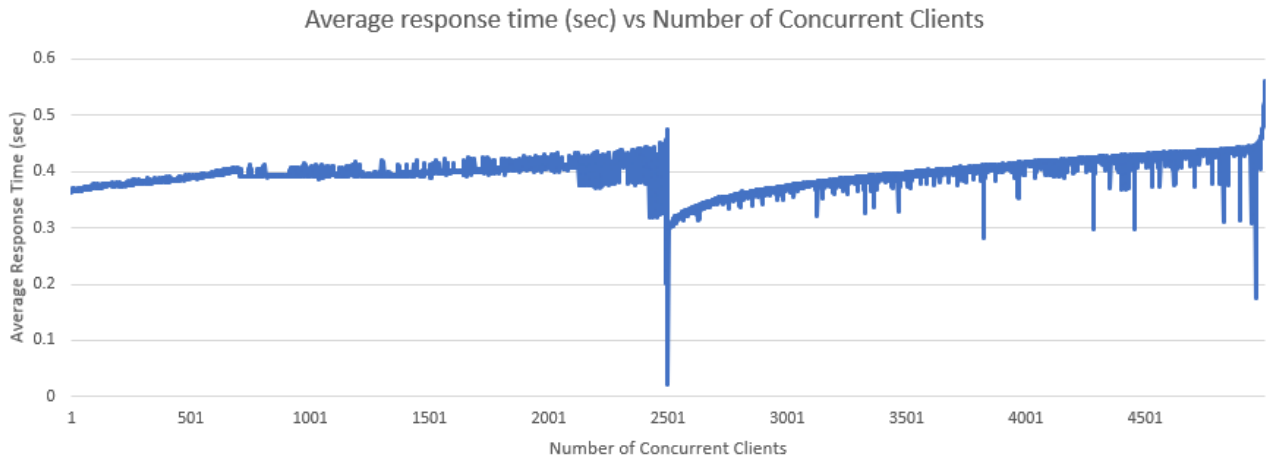


As with test case 8, we can see a slight observable improvement in performance which translates to an overall faster test.

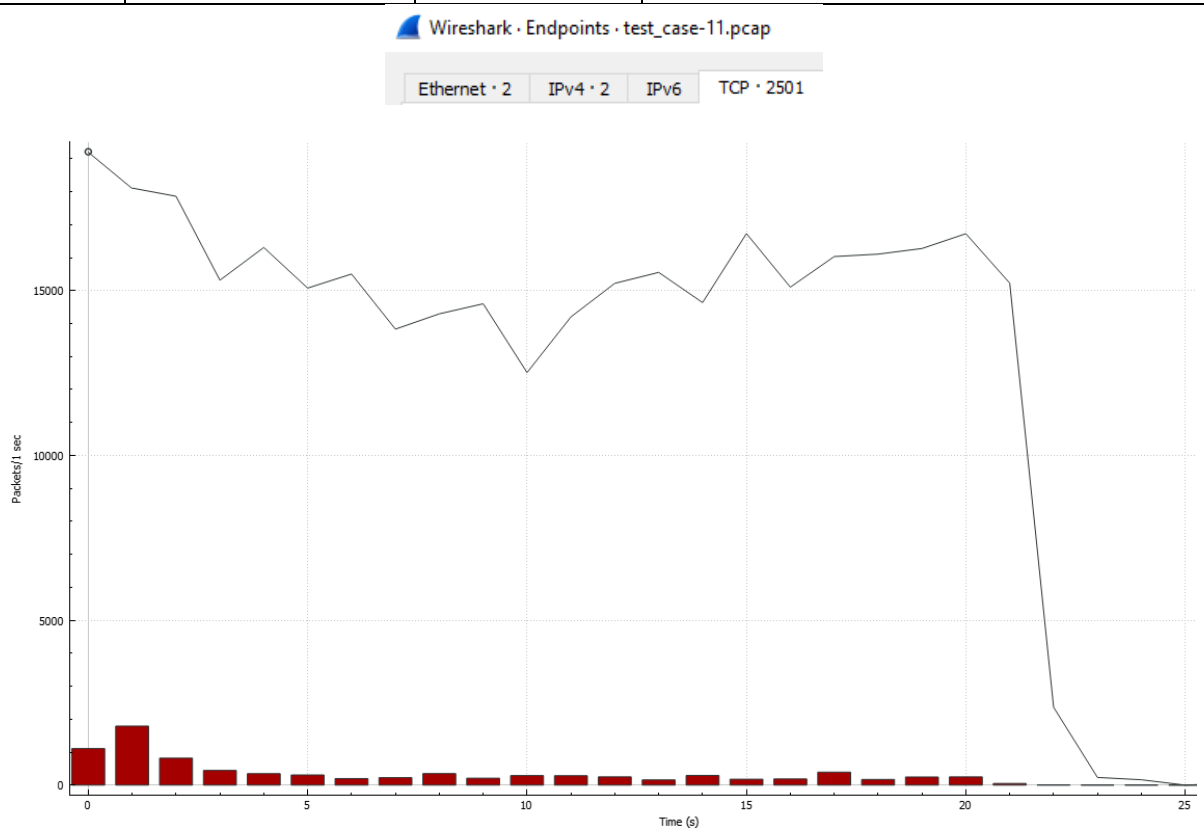


Test Case	Number of clients	Request/Client	Requests Delay (ms)
10	2500	60	10

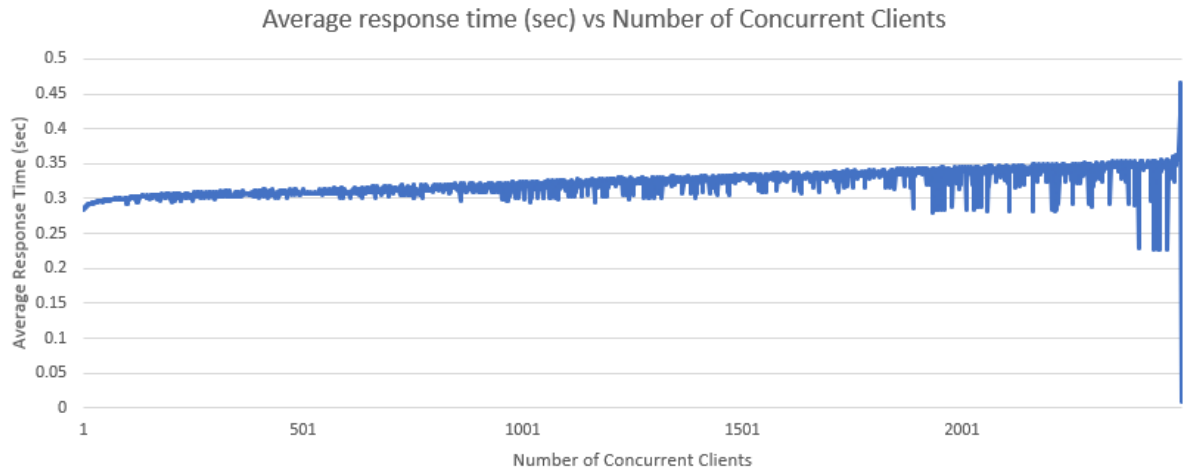




Test Case	Number of clients	Request/Client	Requests Delay (ms)
11	2500	60	1



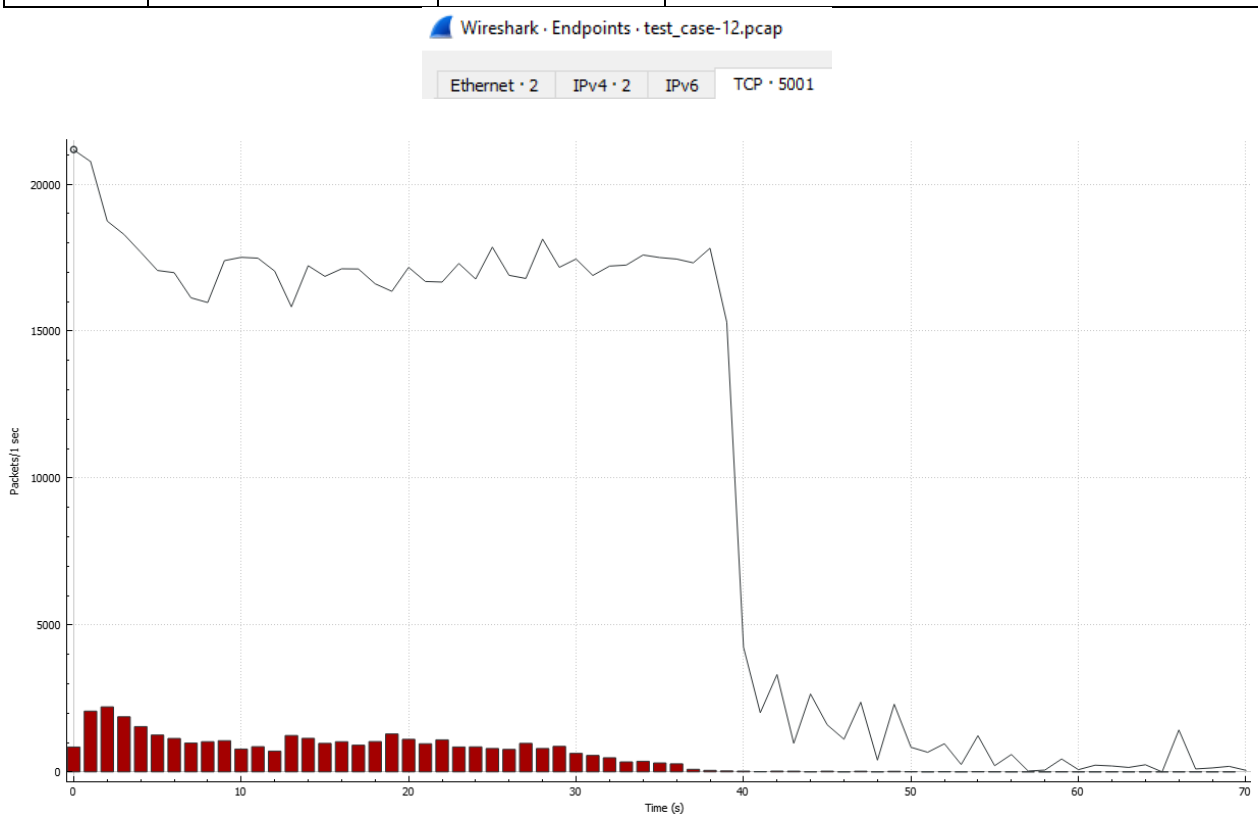
This test case shows an average of about 15000 packets being transferred per second but is the only case that completes the test within 30 seconds. This implies that a greater IO number is most likely due to retransmissions since the number of packets sent and received should be the same despite increasing the frequency of requests.

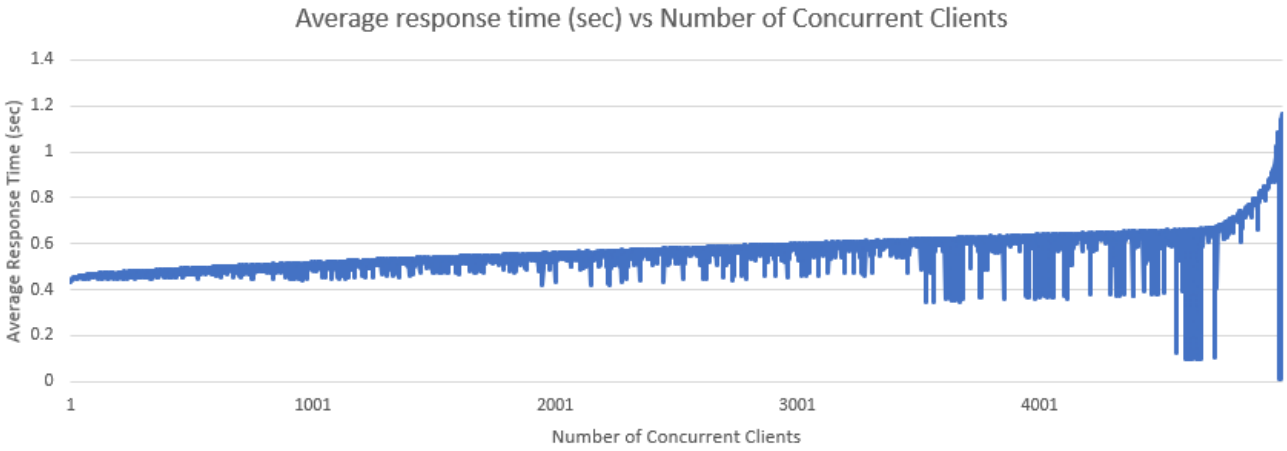


Case 12-14 Results

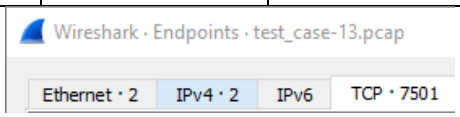
In these cases, we try to determine the effect of increasing the number of clients to lowest frequency of requests from each client to the server. We do this by increasing the number of clients until we can no longer establish all client endpoints on the server and the performance on the server side drops.

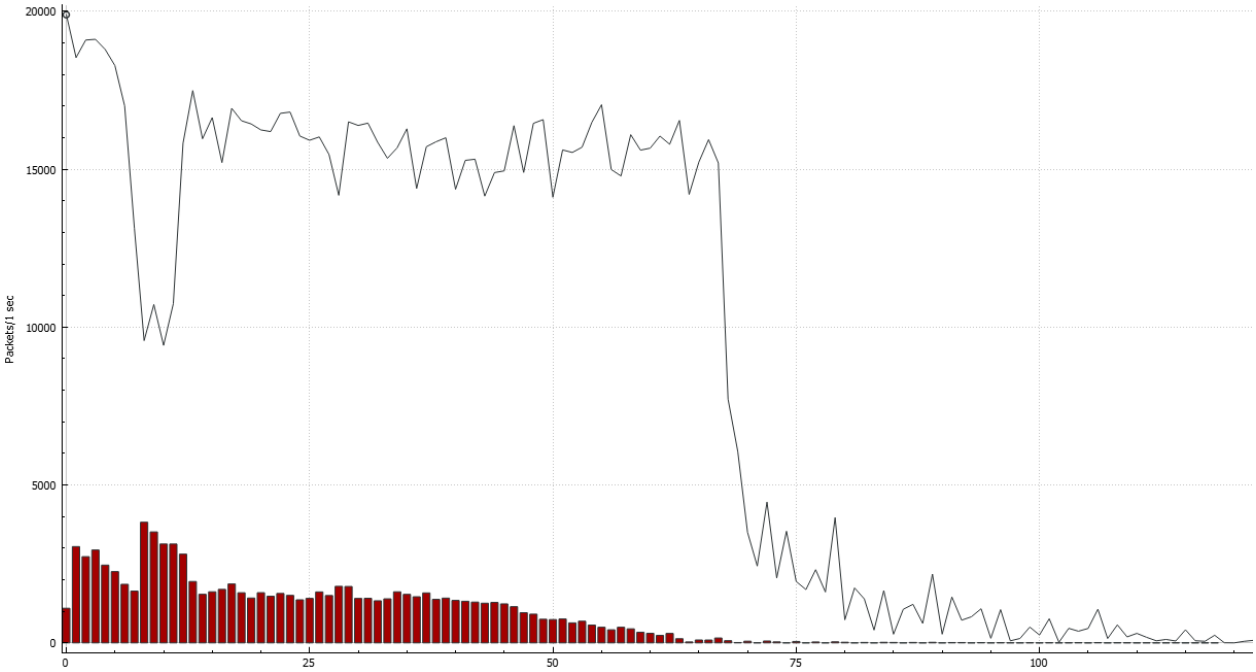
Test Case	Number of clients	Request/Client	Requests Delay (ms)
12	5000	60	1

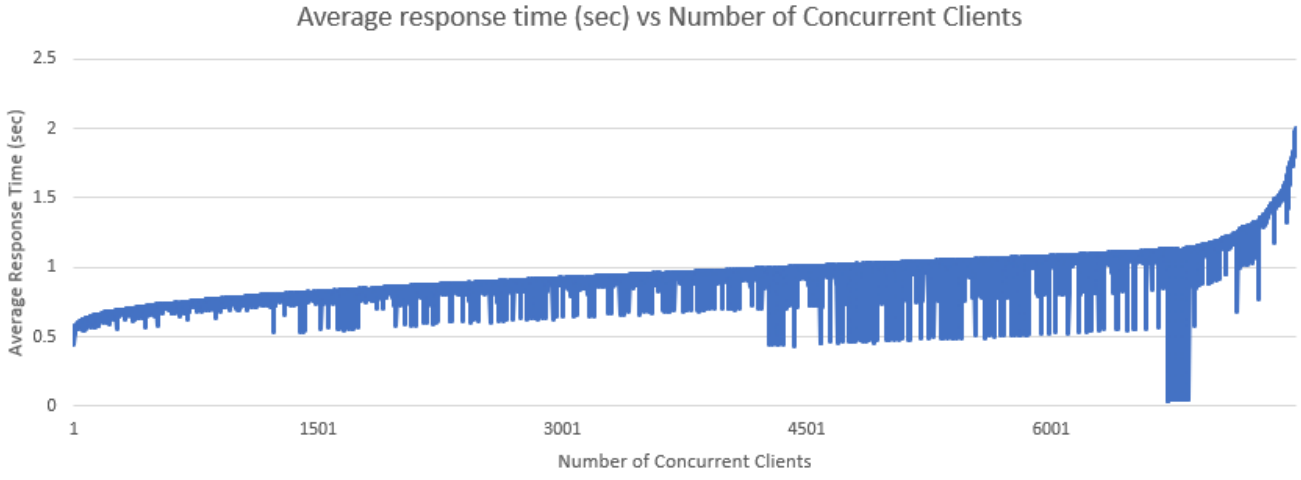




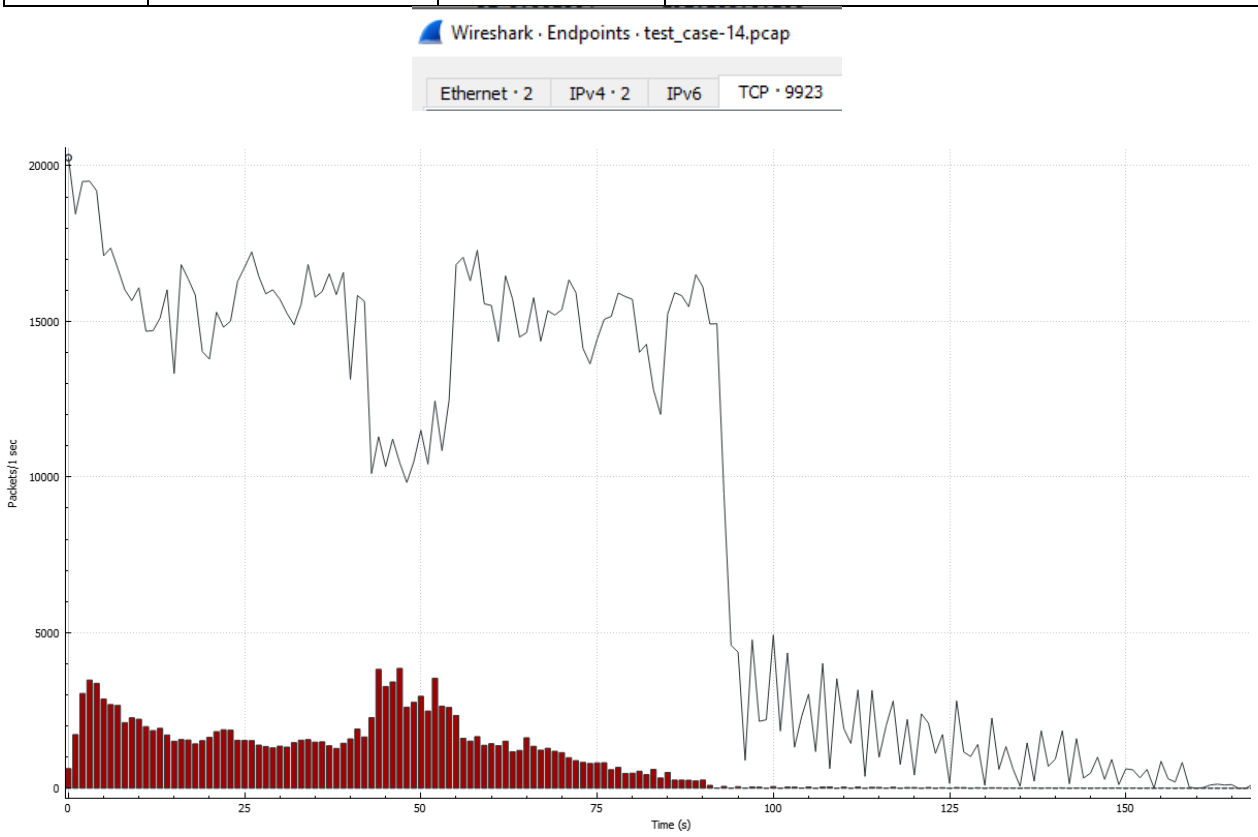
Test Case	Number of clients	Request/Client	Requests Delay (ms)
13	7500	60	1

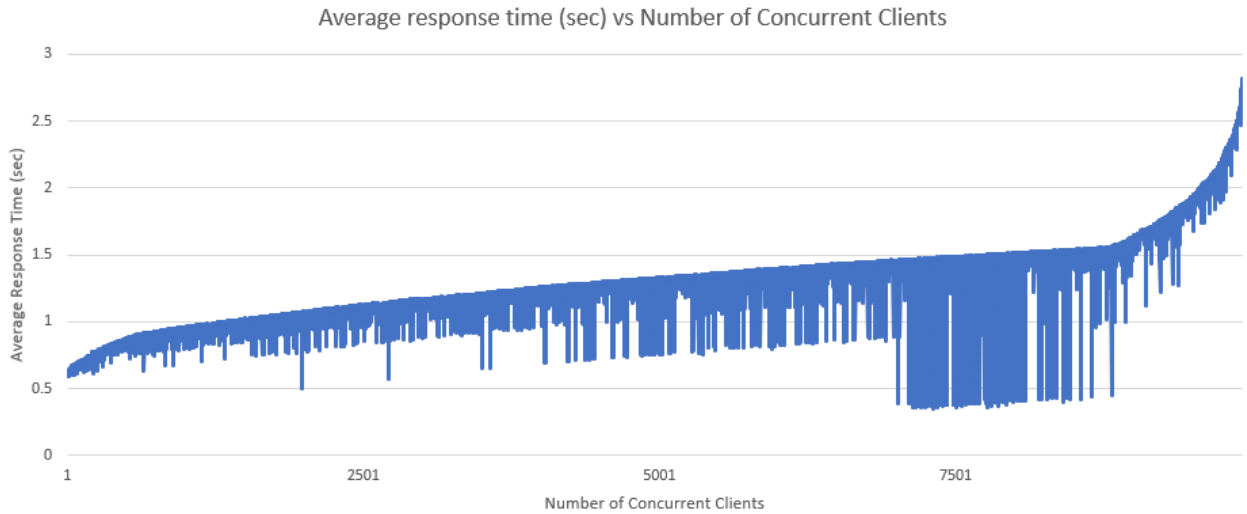






Test Case	Number of clients	Request/Client	Requests Delay (ms)
14	10000	60	1





Analysis

Case 1-5:

Our implementation of the epoll client-server achieved close to 20000 concurrent active socket connections in one session. We noticed a couple of patterns emerge as we begin to increase the number of connected clients. In terms of IO over the wireshark capture, there is a 15000 packets/s average limit for our server once the number of clients is at 5000. Moreover, the variability in IO begins to increase as well along with the response time. When plotting out the average response time (sec) versus the number of concurrent clients in one session for each test case we see a line that is best described as a linear model for most of the clients but exponentially increases as we get to the user specified client number. This means that as we add more clients in one session, the performance degradation is linear until we reach the final clients. There may be room for programmatic optimization that can be done here to remove the exponential response time towards the last few clients. Finally, between test cases, we see that as we double the number of concurrent clients, we approximately double the average response time for each client as well.

Case 6-11:

For these test cases, we observed the change in average server response time to the change in frequency of client requests. The fastest response times (~30ms) are seen when we have a request delay above 125 ms. As soon as we breach 125 ms and lower, the response times take about 10x longer (~300-400ms). This makes sense because as we approach the server threshold to service requests, any additional requests will take longer to attend to. Based on the IO graphs, we also note that there are greater amounts of TCP error flags due to retransmissions, which is likely due to the requests not being handled before timeout.

Case 11-14:

Case 11-14 mirrors case 1-4, but this time we set the requests delay to 1 ms. Here we are trying to see how scalable our server is when clients are sending frequent requests. For example, when we compare the number of client pair at 2500 clients with 1 ms and 1000 ms delay, we are completing the entire test in 25 seconds rather than 68 seconds but with more retransmissions and higher average request response time. When the server is loaded with active connections at 10000 clients, our test with 1 ms delay finishes the entire test just under 175 seconds rather than 220 seconds at 1000 ms delay.

From these tests, we concluded that if we wanted to optimize our server for the least amount of retransmissions to save on bandwidth, we want to be ideally allowing multiple requests from one client with a delay of at least 250 ms. However, if we want our requests to be serviced as fast as possible, it makes sense to send each request as fast as possible. Just note that the average time for request/response will be longer since there will be retransmissions.

Conclusion

Epoll uses edge-triggered readiness notification, where file descriptors are only returned and available for I/O after a change has happened. By implementing an epoll server with edge triggered notification, we observed the linear scalability in various tests that allows for our high-performant server to manage large amounts of concurrent connections. Thus, it is useful when implementing a server that deals with massive number of file descriptors, each needing I/O and data processing operations.