Implementation of Scalable Servers - Epoll versus Select versus Multithreading

David Tran - A00801942 | Cole Rees - A00741578

COMP 6D

COMP 8005 - Assignment 2

British Columbia Institute of Technology

Aman Abdulla

Monday, February 17 2014

# Table of Contents

# 

# Background

The purpose of this assignment is to compare three different approaches to implementing a scalable server. The objective is to create three distinct servers using multi-threading processes, epoll and select functionalities and then to provide them “work” using a TCP client.

For this assignment, “work” will be provided by scripting enough executable TCP clients on one host, and having many client terminals send multiple text lengths to a single server, which will sit on another host entirely (refer to **Design Work & Testing** document for further clarification). Our client, as a multi threaded application, will maintain a sustained connection with it’s multi threaded capabilities. In other words, the server will not terminate connection with the client until all of the clients’ threads are finished.

In our **Observations** section, we will highlight important differences between our three servers. The intention is to find the most efficient implementation of a server so that our goal of developing scalable servers can be achieved. Our goal of “scalable” and “efficient” includes time management, memory usage, and processing capabilities. Through theory and some initial analysis, our hypothesis is that multi threaded servers will be the least scalable, with Epoll servers being the most and followed by Select servers being second.

# Tools & Equipment

## Hardware

|  |  |  |
| --- | --- | --- |
| * 8GB RAM | * Intel i5 Quad Core | * 500GB HDD |
| * Client Host | * Server Host |  |

## Software

|  |  |  |
| --- | --- | --- |
| * Fedora Linux 19 64-bit | * C Programming | * htop |
| * Valgrind | * Wireshark |  |

# 

# 

# Testing Procedure

## Test Cases Table

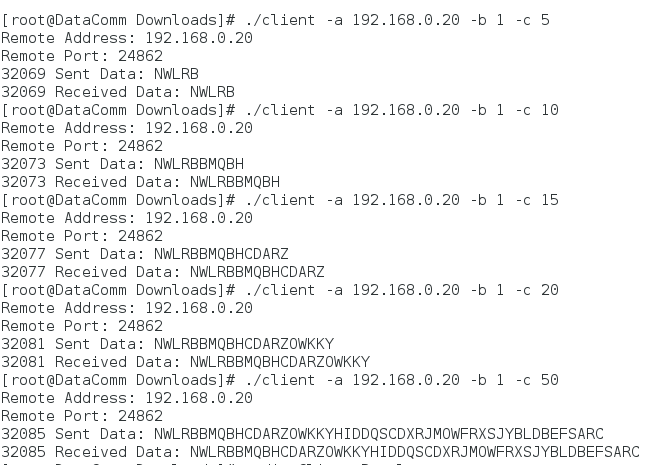
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Case #** | **Test Case** | **Tools Used** | **Expected Outcome** | **Results** |
| 1 | Multithreaded client can send varied string lengths up to the user’s input | Terminal,  csv file | It can send different string lengths | PASSED. See results. |
| 2 | Multithreaded server receives varied lengths of string | csv file | It can receive varied lengths of string | PASSED. See results. |
| 3 | Select server receives varied lengths of string | Wireshark | It can receive varied lengths of string | PASSED. See results. |
| 4 | Epoll server receives varied lengths of string | Wireshark | It can receive varied lengths of string | PASSED. See results. |
| 5 | Multithreaded Client is sending some sort of string | Wireshark, Terminal | It sends some sort of expected string | PASSED. See results. |
| 6 | When the client runs, Multithreaded server receives said string | Wireshark | It receives the same string being sent from client | PASSED. See results. |
| 7 | When the client runs, Select server receives said string | Wireshark | It receives the same string being sent from client | PASSED. See results. |
| 8 | When the client runs, Epoll server receives said string | Wireshark | It receives the same string being sent from client | PASSED. See results. |
| 9 | Multithreaded client is sending multiple sets of strings (via many requests) | Wireshark | We expect to see more than one packet of strings being sent | PASSED. See results. |
| 10 | When the client runs, Multithreaded server receives many strings equal to the number of requests | Wireshark | We expect to see an equal number of packets coming to the server | PASSED. See results. |
| 11 | When the client runs, Select server receives many strings equal to the number of requests | Terminal | We expect to see an equal number of packets coming to the server | PASSED. See results. |
| 12 | When the client runs, Epoll server receives many strings equal to the number of requests | Wireshark | We expect to see an equal number of packets coming to the server | PASSED. See results. |
| 13 | Client keeps track of requests made | GDBC | requests made are equal to requests | PASSED. See results. |
| 14 | Multithreaded server keeps track of number of requests received | GDBC | requests made are equal to the client’s requests | PASSED. See results. |
| 15 | Select server keeps track of number of requests received | GDBC | requests made are equal to the client’s requests | FAILED. No results. |
| 16 | Epoll server keeps track of number of requests received | GDBC | requests made are equal to the client’s requests | PASSED. See results. |
| 17 | Multithreaded server closes the sockets after each client finishes requests | GDBC | closes after requests are processed | PASSED. See results. |
| 18 | Select server closes the sockets after each client finishes requests | GDBC | closes after requests are processed | FAILED. No results. |
| 19 | Epoll server closes the sockets after each client finishes requests | GDBC | closes after requests are processed | PASSED. See results. |
| 20 | Client closes after number of threads are finished | GDBC | closes after requests are processed | PASSED. See results. |
| 21 | No memory leaks from multithreaded server | Valgrind | no memory leaks | Permitted. See results. |
| 22 | No memory leaks from select server | Valgrind | no memory leaks | FAILED. No results. |
| 23 | No memory leaks from epoll server | Valgrind | no memory leaks | Permitted. See results. |
| 24 | No memory leaks from client | Valgrind | no memory leaks | PASSED. See results. |

## 

## 

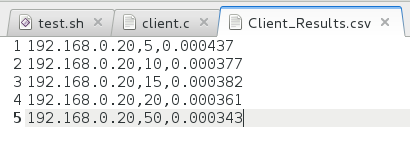
## Test Case Evidence & Details

1. Client can send varied lengths



In the following CSV file, these are the following columns:

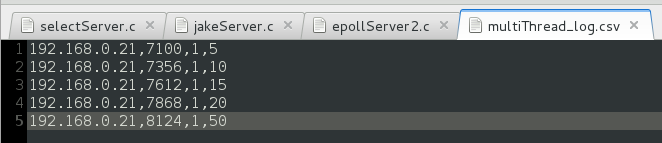
Server IP, String Length, Time to Process



1. Multi thread Server can receive varied lengths of string

In the following CSV file, these are the following columns:

Server IP, Socket, Number of Requests, String Length

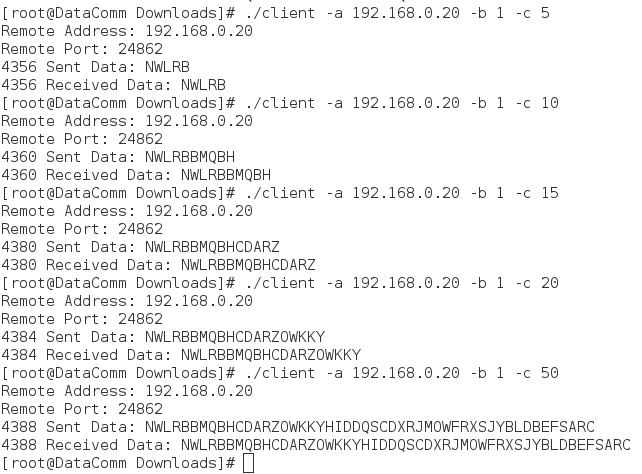


1. Select Server can receive varied lengths of string.

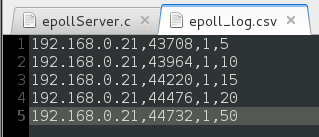


1. Epoll Server can receive varied lengths of string.

Here’s a screen capture of our Client’s requests of varied string:

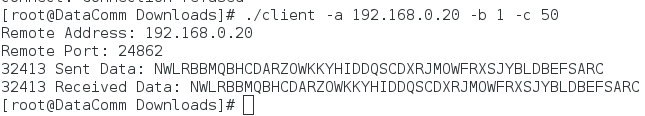


Here is the CSV dump from our Epoll Server. Note the last value, for it is the size of data per client request:

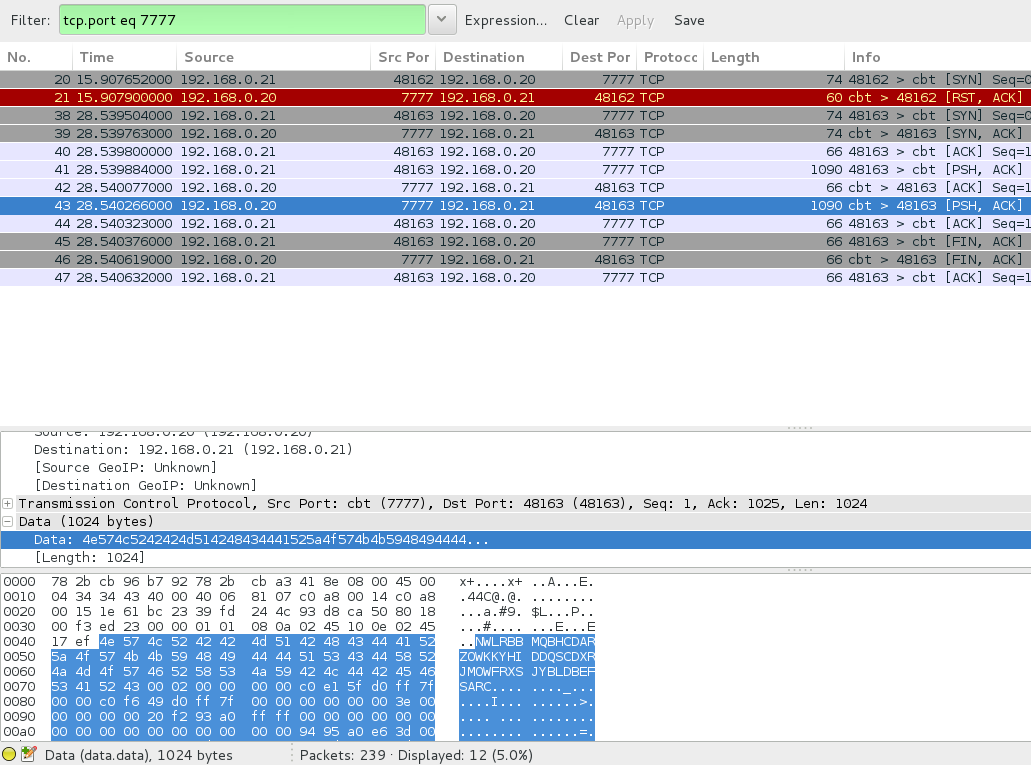


1. Multi threaded Client is sending some sort of string.

In Terminal, here is the expected sending and receiving of our text:

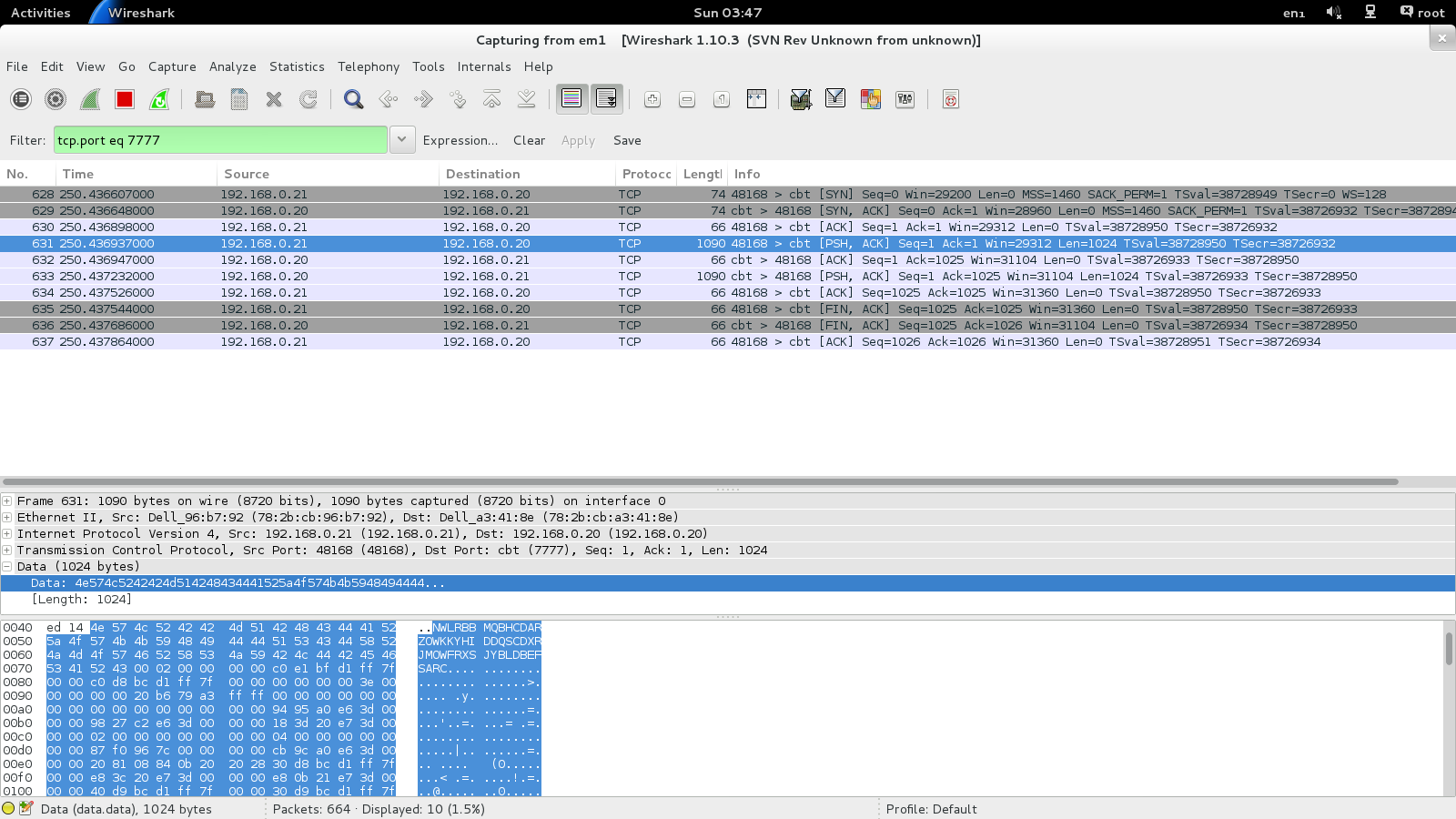


On Wireshark, here is the capture:



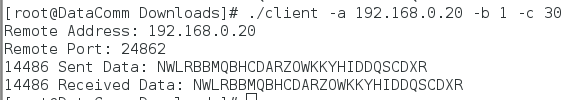
1. Multi threaded Server is receiving data from the Client.

Here’s the Wireshark capture on the Server Host from the previous message from the Client:

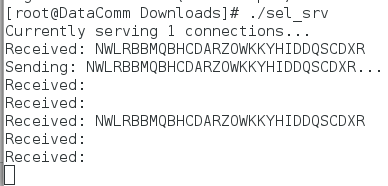


1. Select Server is receiving the same string from the Client.

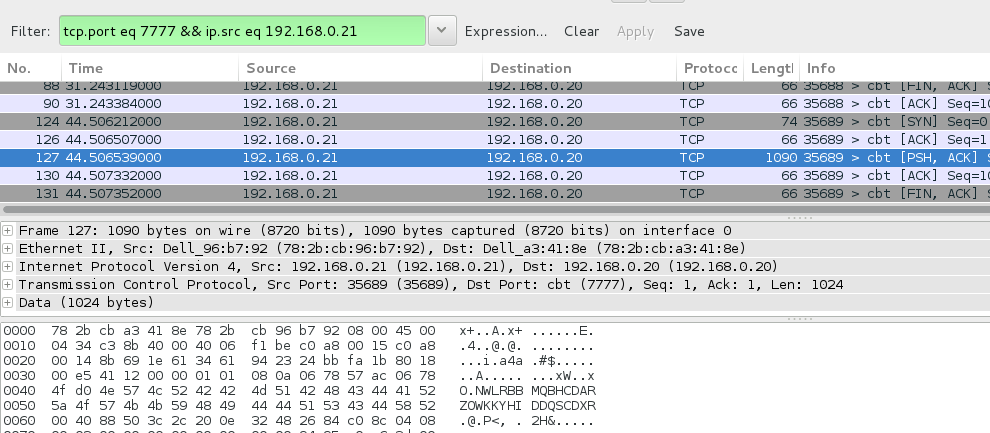
Screen capture from Client:



Screen Capture from Server:

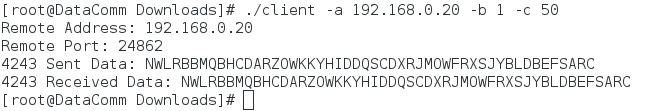


Wireshark Capture:

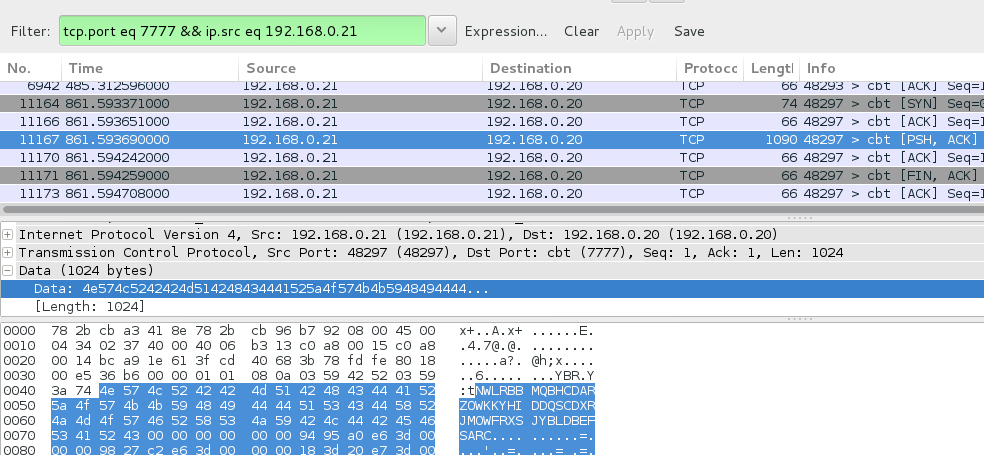


1. Epoll Server is receiving the correct string value from Client.

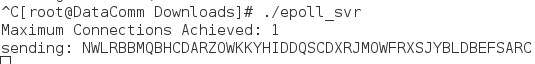
Here is a screen capture of the Client’s string from Terminal:



Here’s the Wireshark Capture:

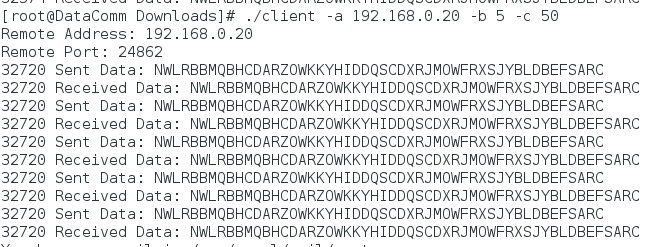


Finally, if the server received correctly, it will send back the exact same string. Here’s the Terminal screen shot of our Server echoing back:

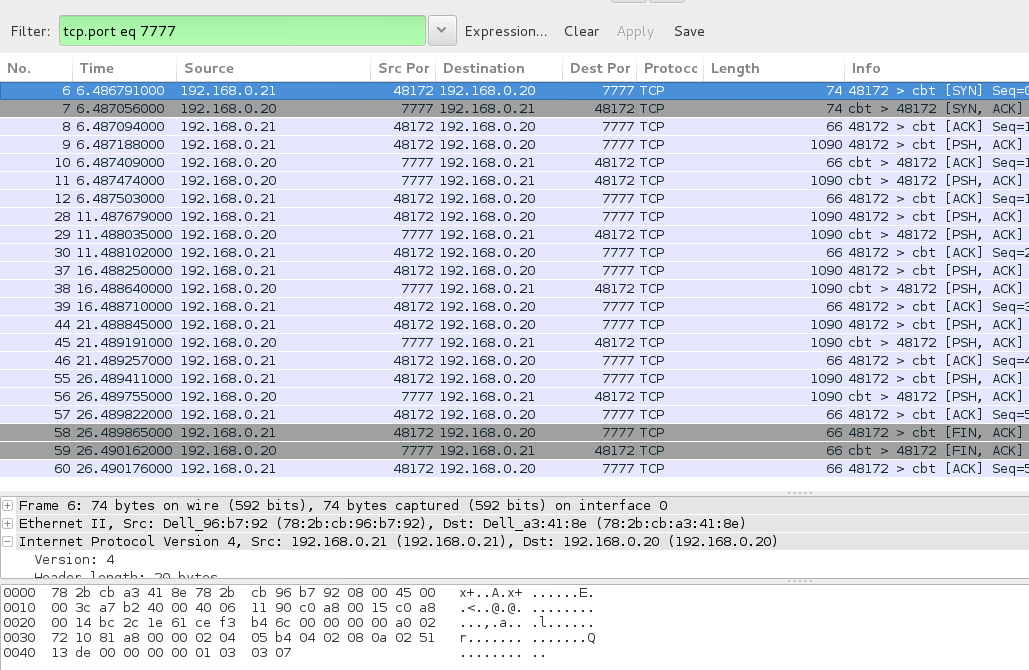


1. Client is sending multiple strings via multiple requests.

In our executable, we’ve specified to send 5 requests to the Server. In our Terminal output, we have 5 pairs of send and receive data:

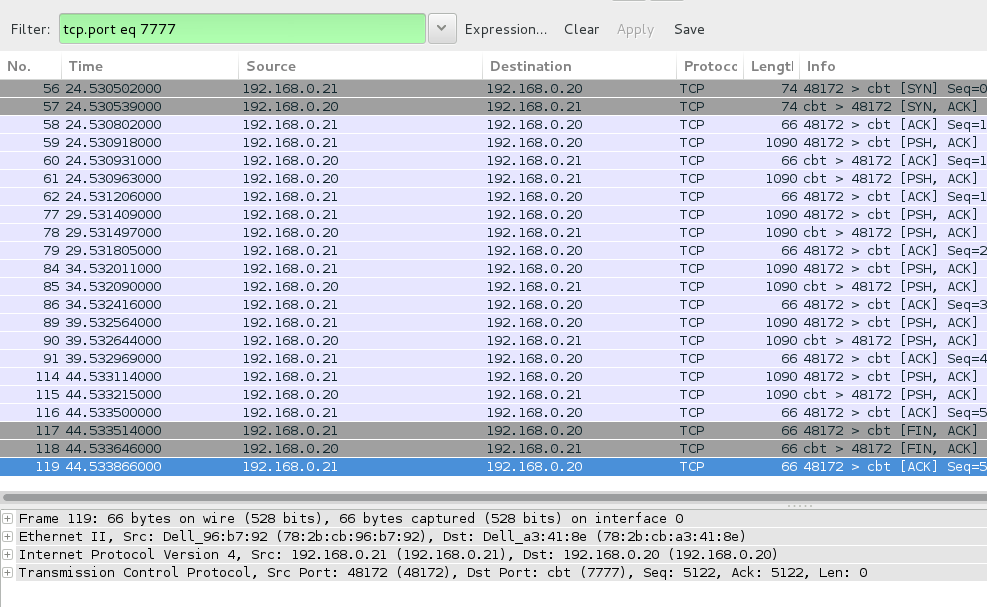


In our Wireshark, here’s our capture on the Client side:

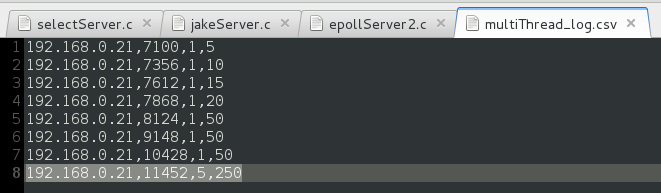


1. Server is receiving that many strings equal to the number of requests.

In our previous test case, we’ve specified the client to send 5 requests. Here is our Wireshark capture of the server:

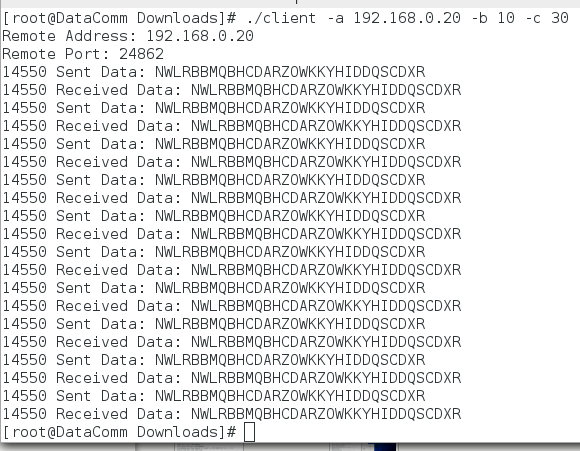


And here is our CSV dump. Note the highlighted line, and the last value of 250, which is 5 requests of 50 bytes each:



1. Select Server receives many strings equal to the number of requests:

Screen Capture of the Client. Note the 10 requests:



Screen Capture of the Server:

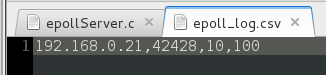


1. Epoll Server receives many strings equal to the number of requests from Client.

Here’s a screen capture of the Client’s requests:

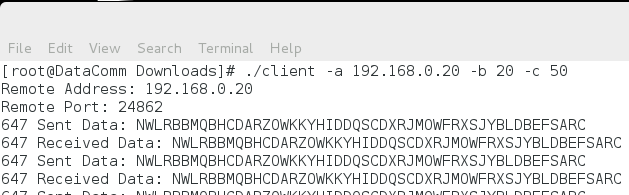


Here is the CSV dump from the Server Side. Note the last two values; 10 is the number of requests received and 100 is the total data processed:

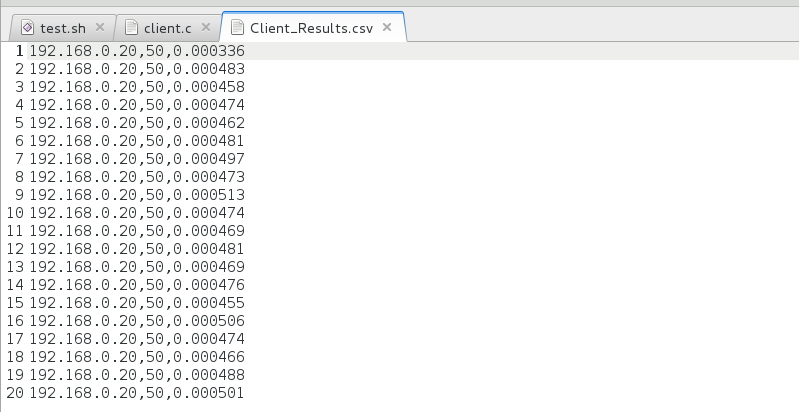


1. Clients are keeping track of the number of requests made.

In our Terminal, we’ve specified the Client to send a request 20 times. Here’s our Terminal output. Note, we cut short the screenshot:

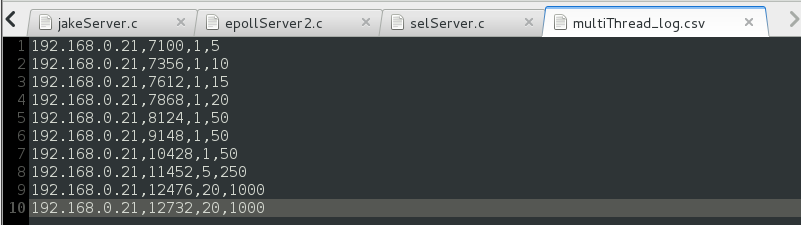


Here is our finished CSV dump:



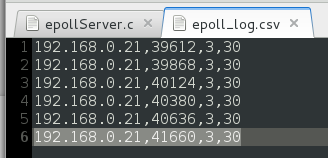
1. Multi threaded Server is keeping track of number of requests received.

In our previous Client, we’ve specified to run 20 requests to the server with 50 bytes of data each. Here is our CSV dump from the Server. Note the highlighted value and the last two values. The second last value is the number of requests and the other is the total bytes sent by that transaction:



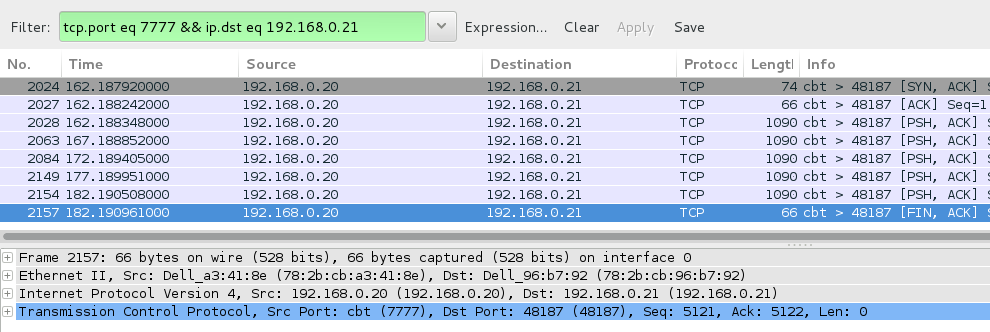
1. Select Server keeps track of number of requests received. FAILED.
2. Epoll Server keeps track of number of requests received.

We’ve specified our client to send 3 packets of 10 bytes each in a separate instance. Here is the screen capture of the CSV dump from our Server. Note the two last values. The second to last value indicates the number of packets received, and the last value indicates the total data in bytes received by the Server:



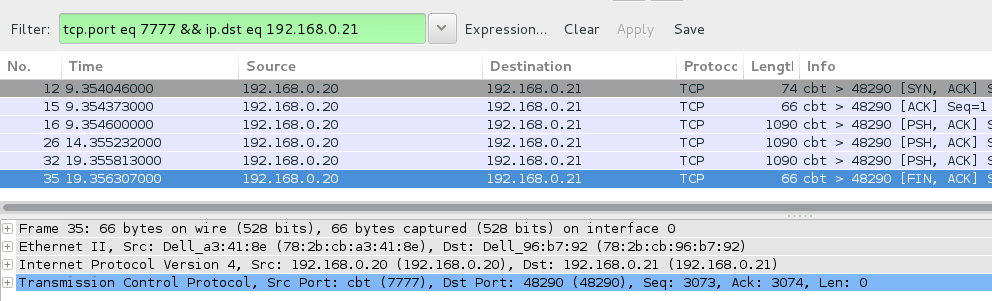
1. Multi thread Server closes the sockets after each client thread finishes requests.

Here is the Wireshark dump of our Server closing the socket:



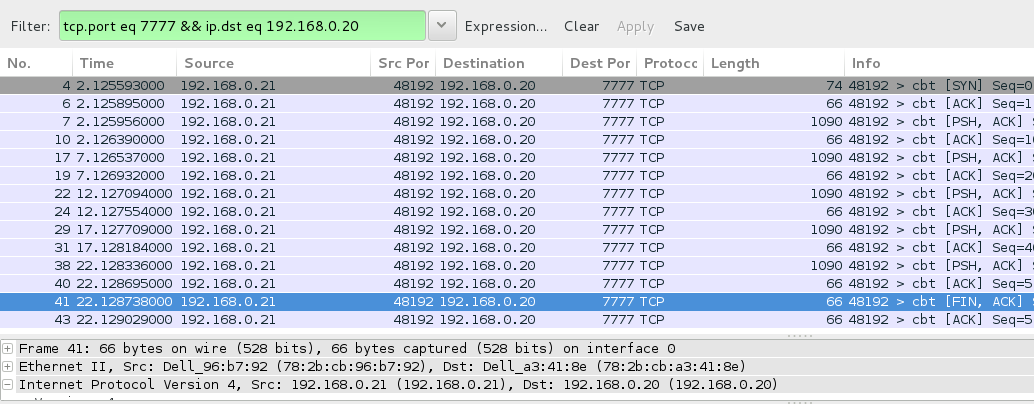
1. Select Server closes the sockets after each client finishes requests. FAILED.
2. Epoll Server closes the sockets after each client finishes requests.

Here is a screen capture of Epoll sending a FIN/ACK, and then closing the socket:



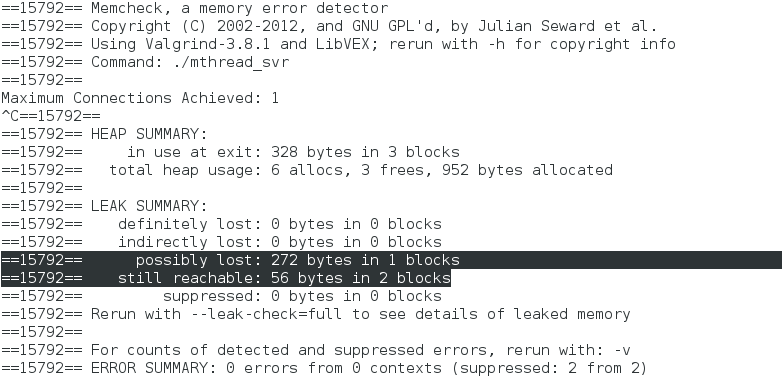
1. Client closes socket after finishing requests.

Here is our screen dump of our client:



1. No memory leaks from Multi threaded Server.

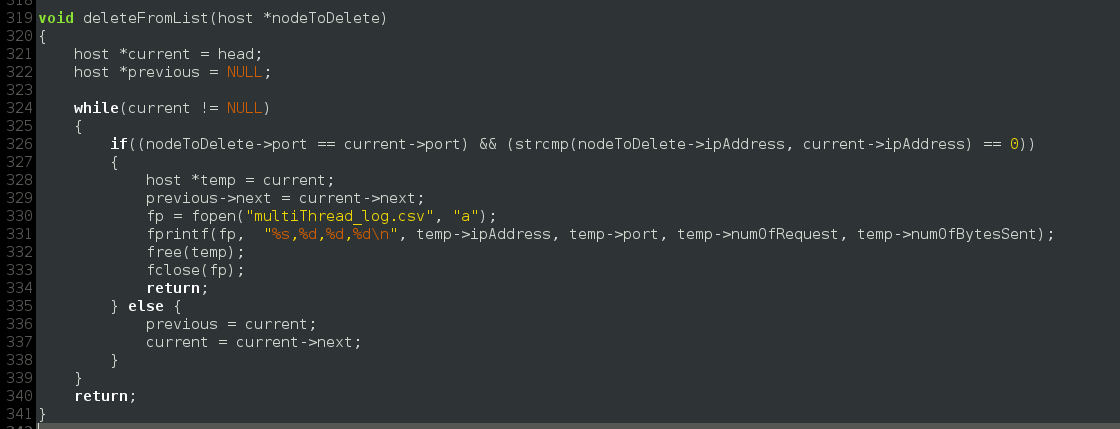
After running Valgrind on our multi threaded server, the screen capture below shows some erroneous values:



These values are revealed because of our code design. Please refer to the following two screenshots:



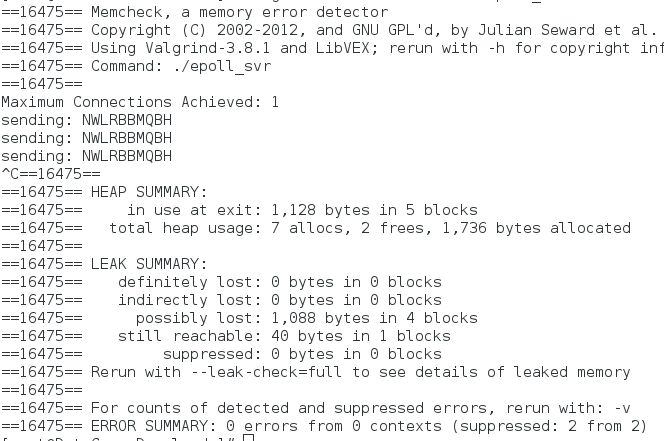
Note the two mallocs() from addToList(). Now, note the one free() in deleteFromList().



Because these calls are made within other methods, they resulted a mismatch. This then triggers a false positive for Valgrind because it’s embedded in other methods. We will allow this to be permissible since it’s not a definite or indirect loss of memory.

1. No memory leaks from Select Server. FAILED.
2. No memory leaks from epoll server.

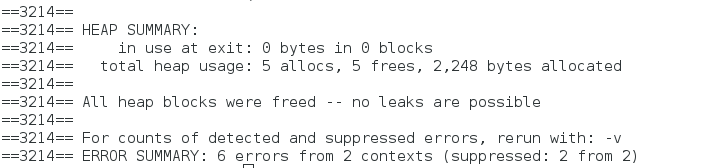
Again, same expected results that are similar with our Multi threaded Server. Here we see some erroneous values of memory leaks:



However, because these calls are made within other methods, they resulted a mismatch. This then triggers a false positive for Valgrind because it’s embedded in other methods. We will allow this to be permissible since it’s not a definite or indirect loss of memory.

1. No memory leaks from our Client.

After running Valgrind on our Client machine, the screen capture below shows no erroneous values:



# Observations

To begin our comparisons, we will start the three servers and fill in the following table:

|  |  |
| --- | --- |
| **Host** | **IP Address** |
| Servers (Multi Threaded, Select, Epoll) | 192.168.0.20 |
| Client (1) | 192.168.0.19 |
| Client (2) | 192.168.0.21 |
| Client (3) | 192.168.0.22 |
| Client (4) | 192.168.0.18 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Case / Server** | **Multi Thread Server** | **Select Implemented Server** | **Epoll Implemented Server** |
| Max Clients\* | 1782 | 574 | 19990+ |
| Max Requests\* | 13547 | \*\*2870000; 2775 | \*\*199 900 000; 321410 |
| Avg. Time to Process | 0.00036335 s | 0.0004602 s | 0.001196839 s |
| Memory Usage | 1.825 GB | 1.400 GB | 2.651 GB |
| Avg. Processing Power | 93.25% | 29.5% | 12.33% |

\* the maximum value before server or client crashes

\*\* theoretical value; actual value

## Multi Threaded Server

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Client 1** | **Client 2** | **Client 3** | **Client 4** |
| Max Clients | 3000 | 3000 | x | x |
| Max Req’s | 6870 | 6870 | x | x |
| Avg. Time | 0.0003648 s | 0.0003619 s | x | x |
| Mem. Usg. | 1067 | 1054 | x | x |
| Prcs. Power | 6.3% | 5.8% | x | x |

Clients 3 and 4 were omitted in this test because their inclusion would result in a quick crash of the server.

## Select Server

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Client 1** | **Client 2** | **Client 3** | **Client 4** |
| Max Clients | 1000 | 1000 | 1000 | 1000 |
| Max Req’s | 5000 | 5000 | 5000 | 5000 |
| Time | 0.0004602 | 0.000451607 | x | x |
| Mem. Usg. | 1.200 GB | 1.225 GB | x | x |
| Prcs. Power | 20% | 21.2% | x | x |

## Epoll Server

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Client 1** | **Client 2** | **Client 3** | **Client 4** |
| Max Clients | 5000 | 5000 | 5000 | 5000 |
| Max Req’s | 10 000 | 10 000 | 10 000 | 10 000 |
| Time | \*0.000535055 s | \*0.000566556 s | \*0.001023102 s | \*0.002662644 s |
| Mem. Usg. | 2.200 GB | 2.000 GB | 1.650 GB | 2.500 GB |
| Prcs. Power | 13.8% | 12.2% | 12.6% | 16.2% |

\*these stress tests were cut short to grab data earlier; values are actually greater than specified



Figure 1 - An Interpretation of Max Clients for the three Servers

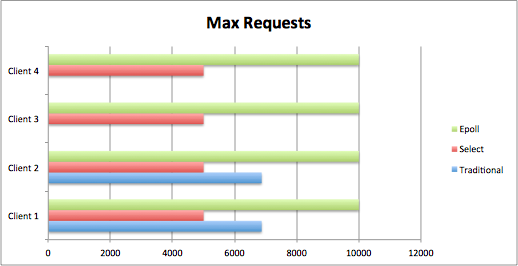


Figure 2 - An Interpretation of Max Requests for the three Servers

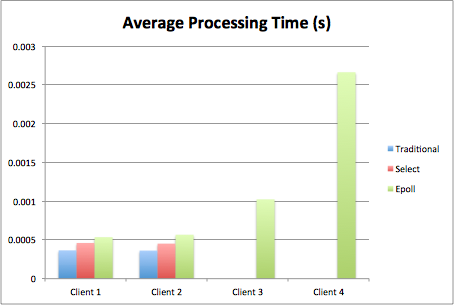


Figure 3 - An interpretation of Average Processing Power for the three Servers

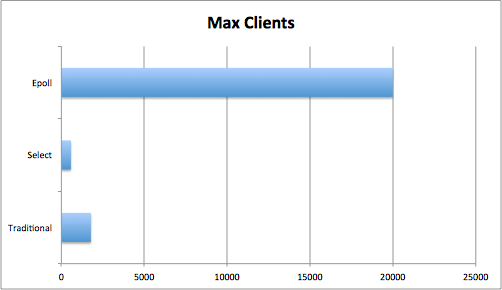


Figure 4 - The Maximum Clients per server at run time

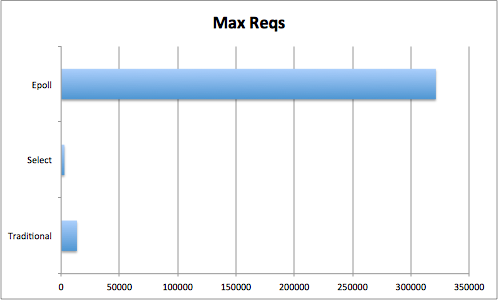


Figure 5 - The Maximum Requests per server at run time

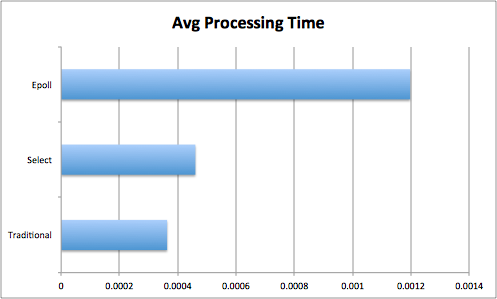


Figure 6 - The Average Processing Time per server at run time (s)

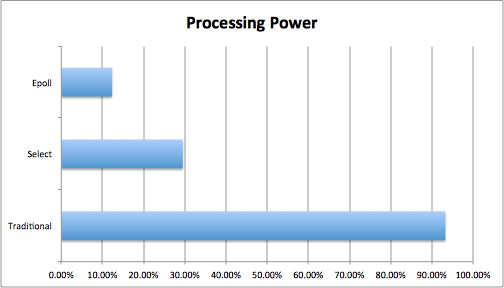


Figure 7 - The Average Processing Power over four cores per server at run time

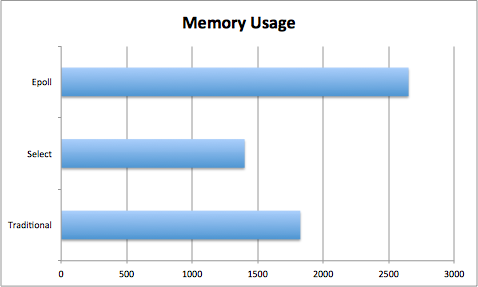


Figure 8 - Average Memory usage over 8 GB of RAM per server at run time

# 

# Limitations

Due to time constraints, we have failed to develop a working and functional Select server. The Select server provided in this assignment was designed to follow our initial ideas and thought process on how to approach the problem. However, the approach was flawed and it was too late for us to change the Server design for a more complete one.

Our Select server is able to sustain at least 1000+ connections. The logic within our code was at fault for the lack of a sufficient server. Our problem lies within the Server closing sockets and how we were storing file descriptors. We failed to implement proper logic to designate which file descriptors have been used. What resulted was, when a file descriptor was cleared it was still being used by the client.

To redeem ourselves, we focused more heavily early on with the traditional multithreaded server in the hopes we would get a more thorough understanding. Afterwards, we dedicated more research and time into the Epoll server, which is fully functional as intended. We feel that since we will be using Epoll later on, we had a heavier emphasis on it versus the Select server.

# Conclusion

From the observations in our experiment, we conclude that our hypothesis is true to an extent. Firstly, we see that our Epoll server implementation outperforms any of the other servers by far in terms of scalability and reliability. Unfortunately, the case where we assumed that Select was more functional than the traditional multi threaded server is proven to be a false case (see **Limitations** section for more details).

In Figure 1, we see the 4 Clients being serviced by the Servers. Obviously, it’s important to note that only Epoll was able to service all four clients without either crashing or being compromised. Edge: Epoll Server

In Figure 2, the maximum requests are proportionate to the number of Clients and their client spawns. It is important to note that here, we expected Select server to be more dominant than our multi threaded server. Instead, because our Select server was much more inefficiently designed, our multi threaded server outperforms. Still, it is important to see that Epoll is designed to handle more requests per customer.

Edge: Epoll Server

In Figure 3, analyzing Clients 1 and 2, we see that Epoll takes slightly longer to process the client’s requests versus Select and traditional multi threaded server. However, the slightness is minute, so we can neglect it.

Edge: Multi Threaded Server

In Figures 4 and 5, only the Epoll server was able to satisfy the maximum amount of requests and clients at run time versus the other two, which ended up crashing. The request are proportionate to the number of clients as well.

Edge: Epoll Server

In Figure 6, it is apparent that Epoll server takes a longer time to process these requests. Since these requests are small in size, it is consistent that multi threaded servers are servicing faster, but albeit less.

Edge: Multi Threaded Server

In Figure 7, it is important to see that, while the traditional server is quickly processing these requests, it is taking up a large amount of processing power over four cores. Instead, it is better to see that Epoll and Select servers are pacing themselves, only consuming up to 30%.

Edge: Epoll Server

In Figure 8, it is interesting to see that Select is consuming less memory than its counterparts.

Edge: Select Server

## Verdict

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Traditional Server** | **Select Server** | **Epoll Server** |
| **Figure 1** |  |  | x |
| **Figure 2** |  |  | x |
| **Figure 3** | x |  |  |
| **Figure 4** |  |  | x |
| **Figure 5** |  |  | x |
| **Figure 6** | x |  |  |
| **Figure 7** |  |  | x |
| **Figure 8** |  | x |  |
| **TOTAL** | 2 | 1 | 5 |

In conclusion, it is important to note that Epoll Server wins its case 5 out of 7 comparisons. It is wise to develop and design future projects using Epoll implementation.

# 

# Appendix

Located on disk are the following:

* Implementation of Scalable Servers - Epoll versus Select versus Multithreading (.pdf)
* Implementation of Scalable Servers - Design Work & Testing (.pdf)
* selServer.c (and selServer.exe)
* multiThreadServer.c (and multiThreadServer.c)
* epollServer.c (and epollServer.exe)
* test.sh
* Makefile
* Client.c
* README.txt