**Comparing and Evaluating multi-threaded, select and epoll Event Mechanisms In The Post “C10K” World.**

**Elton Sia, Thilina Ratnayake  
*British Columbia Institute of Technology***eltonsia@gmail.com,thilina.ratnayake1@gmail.com

**Abstract**

This paper uses a high performance event driven echo server to compare the performance of default multi-threaded response mechanism, as well as the select (level-triggered) and epoll (edge-triggered) event mechanisms. Using Louiay Gammo and team’s paper as a guide, our goal was to beat their results as well as the “C10K” challenge..

Throughout the experiment, we noticed that our server using the select mechanism, was not getting the performance that we wanted. To improve performance, we revised the server to use a multi-threaded approach to handling connections.

Upon completion of the experiment, we were able to beat the Waterloo team’s results, as well as the “C10K” challenge even with our multi-threaded server. This can be attributed to the improvement in hardware over the last decade. Our results also showed that performance in handling a large number of connections were in the order of Epoll, Select and multi-threaded. Interestingly enough, we found that while EPoll reigned supreme in its ability to handle a large amount of connections, it had the slowest response times.

**1 Introduction**

The Internet came, and has continued to expand in all aspects since its inception in the early 1980’s. Fueled by a number of factors but not limited to: Moore’s law, the increasing amount of wireless infrastructure and even socio-technological cultures such as the “Internet Of Things”, it’s expansion has been so furious that we are even at a point now where we are facing a shortage of addresses for the amount of devices.

In Loay Gammo et al’s paper in 2003, they spoke about how this expansion in use makes it “imperative to be able to support these changes with faster and more efficient HTTP servers”. This is especially true today, but with the added requirement that it is no longer just HTTP pr merely about being able to respond to a large number of requests with responses. While that may have been the case with the Internet as early as a decade ago, the nature of today’s internet is much more dynamic, placing an emphasis on real-time (stateful) and dynamic content which in turn creates a requirement for servers to establish and maintain connections with clients.

Each of the echo servers uses different event mechanisms to compare the performance amongst them. We use the traditional multithreaded server, a multithreaded select server, and Epoll server to compare our tests. All three servers only accept connections, echo back the data that was sent to it and disconnect the clients.

**2 Experimental Environment**

The experimental environment consists of one server and twelve clients. The server and clients are running identical hardware and software containing four Quad-core Intel Core i5-2400 CPU’s, with 8 GB – 1333 mHz of RAM. Each server only functions to listen and establish connections, as well as receive and “echo” back the data to each client.

**2.1 Workloads and experiment design.**

Using the Waterloo team’s experiment as a guideline, we decided to use two scenario’s to evaluate our servers with. Each test involves 12 clients sending messages to the server, starting with a base of establishing 5000 connections before beginning the transmission of data. It will then send 10 volleys of messages and upon successful receipt of the replies, will continue to increment the number of established connections by 1000. With each successful increment of clients, we will re-send 10 volleys for every client and wait for successful receipt of replies before continuing to increment more. The test concludes after the RTT’s for clients exceeds our time-out value, which is 60 seconds. The only variables for our experiments were the size of each message, for which we tested (1) 256 bytes and (2) 512 bytes.

For the purpose of this experiment, performance is defined by the following two criteria:

1. Observing how many concurrent connections can be be handled and;
2. The RTT’s experienced by each client.

**3 Experiment Results**

Upon completion of our experiment, our data seems to be in line with the results found in Waterloo in that the performance of *epoll* beat out *select* which was better than the performance of a purely *multi-threaded* server. In this section we will present the data in graph form, as well as pertinent data at-a-glance. We will attempt to explain the data in the following Discussion section.  
  
**3.1 Results At-A-Glance**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Multi-Threaded 256 bytes | Select 256 Bytes | Epoll 256 bytes | Multi-Threaded 512 bytes | Select 512 Bytes | Epoll 512 bytes |
| Max number of connections | 31884 | 71724 | 99612 | 31884 | 68736 | 93636 |
| Average RTT (of Average RTT’s) | 100.8377143 MS | 100.8377143  MS | 233.8739  MS | 146.2034286  MS | 168.0428462  MS | 273.988856  MS |
| Average Min RTT | 0.373464286 MS | 13.00093 MS | 14.11965 MS | 0.589821429 MS | 12.34966154  MS | 11.2308  MS |
| Average Max RTT | 8935.322393 | 2873.348  MS | 4352.063  MS | 10359.22407  MS | 2470.043138  MS | 4267.85743  MS |

**3.2** **Results In Graphs  
  
Figure 1: The overall picture of performance amongst all three servers with two different data sizes.**

**Figure 2: Performance of servers with 256 bytes of data.**

**Figure 3: 512 Byte data size – Performance of servers with 256 bytes of data.**

**Figure 5: 256 & 512 Byte Data Size – Average Minimum RTT’s vs Number Of Connections**

**Figure 6: 256 & 512 Byte Data Size –Average Maximum RTT’s vs Number Of Connections**

**4 Discussion**

Upon completion of our experiment, our data seems to be in line with the results found in Waterloo in that the performance of *epoll* beat out *select* which was better than the performance of a purely *multi-threaded* server.

**4.1 Multi-Threaded**

Our server using a multi-threaded mechanism was the least capable of the three servers, maxing out at only being able to handle 31884 connections at 256 bytes of data and 31884 connections at 512 bytes. This can be explained by how our server is written; wherein we create and maintain a thread for every connection. These threads must remain active and be constantly listening for activity which is very expensive in terms of memory and intensive on the system. The server fails when the context-switches are unable to keep up with activity in all the threads.

One interesting point to note was that even with both data sizes, the multi-threaded server “died” at the exact same point maxing out at handling 31884 connections. We believe that this indicates the bottleneck is in the amount of memory required to continuously listen to a client, and that the requirements of actually handling message activity is quite neglegable.

**4.2 Select**

Using the select mechanism garnered us a better level of performance, increasing our maximum number of connections to 71724 at 256 bytes of data and 68736 at 512 bytes of data.

The increase in “performance” in terms of being able to handle a larger number of connections, can be explained by how our server only utilizes multiple threads with one thread purely listening for connections and adding to the file descriptor list, and the other there for handling message activity. Only using two threads and blocking on a select call waiting for activity, we expense far less memory, especially when compared to when creating a new thread for every new connection.

While it did allow us to handle a larger number of connections, it is important to note that the the RTT’s increased as well. This “decrease” in performance can be explained by the nature of the select call, where upon each call it must check every file descriptor assigned to it before proceeding.

**4.3 Epoll**

The EPoll mechanism gave us the best results for both criteria of 99612 at 256 bytes of data and 93636 at 512 bytes of data. It proved to be the best server in both areas of performance, by also having the lowest average RTTs.

We believe that the reason this mechanism can handle so much more connections and respond to messages faster, is because of the truly asynchronous nature of the mechanism. While using the *select* call, the server is responsible for constantly listening for changes on every socket within the assigned list, and then responding accordingly which is a *quasi*-asynchronous and “pulling” approach; E*poll* allows the server to be strongly event-driven and truly asynchronous by clearly defining what to do in response to an event and only triggering once there is a defined action, which is similar/equivalent to that of a “pushing” action.

By declaring how to handle actions and allowing each socket to autonomously respond to events instead of blocking the whole program, checking all the sockets, then handling based on whatever the results of that check were; this allows our server to be single threaded, yet truly asynchronous and handle a LARGE number of connections.

4.4 Multi-threading Epoll To Increase Performance

Currently, our *epoll* server only utilizes a single threaded approach. We believe that multi-threading it would most probably increase performance at-least within the criteria of handling more connections. The basic idea would be having one thread to handle connections, and another/multiple threads to handle activity in sockets (similar to the way in which we wrote our *select* server).

**5 Conclusion**

Through conducting this experiment, we were able to prove the theories that were taught in lecture in that an event-handling architecture is a good technique for approaching network programming.

First and foremost, it is clear that event-based programming is the best way to handle large amounts of connections as having a single process or thread dedicated to each client is much too expensive as proved by the lack of performance in our multi-threaded server. We implemented this in a less than optimal fashion by using the *select* call, in which we blocked on a list of sockets and only acted when there was activity. While this improved our performance, we have concluded that the best way to approach this problem is by implementing event-handling at the lowest possible level. As demonstrated with the performance of our *epoll* server, performance is greatly increased when the lowest level (in our case sockets), are told how to handle events and allowed to handle them on their own (being edge triggered) instead of blocking the whole program, checking for events, and then handling them (being level triggered). Lastly, we noticed that even with our multi-threaded server, the improvement in hardware and software over the last decade has significantly increased performance to the point that the “10K” is easily surpassable even on standard machines.

**6 Acknowledgements**

We gratefully acknowledge Andrew Burian, fellow set-mate at BCIT BTech for providing us a distributed client network and application to facilitate testing.

**7. References**

1. Louay Gammo, Tim Brecht, Amol Shukla, David Pariag. Comparing and Evaluating epoll, select and poll Event Mechanisms. In *Proceedings of the Ottawa Linux Symposium.* 2004
2. Kegel, D. (2003, November 3). The C10K problem. Retrieved February 23, 2015, from http://www.kegel.com/c10k.html