Technical Report

Scalable Server

COMP 8005 BCIT 2016

February 13, 2016

Authored by: Marc Vouve, A00848381

TECHNICAL REPORT

Scalable Server

CONTENTS

SUMMARY

The Go programming language was developed by Google and is primarily intended for high volume web traffic. Within Go, it is primarily intended that regular connections use the net library; however, there also exists a system call library which provides a handle to various system calls including select and epoll which in Linux multiplex file descriptor I/O. Each approach to File IO has different strengths and weaknesses, the purpose of this report is to discuss the benefits of each.

Through testing, it was discovered that EPoll was the fastest way to handle a large number of connections to a server. There were mixed results though, EPoll often had much shorter RTTs than could be expected, and the reason for the anomaly has not been adequately solved.

Both select and the multi-threaded server produced reliable results, and it appeared as though each would be able to sustain a significant number of connections without failure. As far as implementation went, the simple multi-threaded server was by far the easiest, and it had the additional benefit of being cross platform compatible.

INTRODUCTION

This report discusses the strengths and weaknesses of EPoll, Select and a traditional multithreaded application approach to scalable networking using Go on Fedora 22 in a multi-core environment. They will be compared in terms of estimated RTTs over various server loads.

ENVIROMENT

This report was completed using 4 multi-core Linux computers running Fedora 22. The compiler used was a modified version of the Go 1.5 compiler. The modification made was to the FdSet which is a part of the syscall package. By default FdSet is defined as

```
FdSet {
Bits [16]int
}
```

In the test environment it was modified to be

```
FdSet {
Bits [2048]int
}
```

THEORY

In the modern Linux Operating System the primary ways to create a scalable server are using EPoll or Select multithreaded or multi-processed. A more traditional approach involves multithreading connections.

Go, a language developed by Google, provides a way to do all of these tasks; however, sometimes there are differences from the default Linux implementation which effect performance.

Historically epoll an answer to the C10K problem, a name for an issue first coined by Dan Kegel (Kegel, 2013). In the 2010s a new problem has emerged known as the C10M problem, in which computer Scientists attempt to have 10 million clients connect to the same host.

There are several differences between EPoll and Select, primarily:

- EPoll is edge triggered, which means that connections notify when first arriving at the server.
- Select is level triggered which means that it is notified when the data is at the host.
- EPoll uses an event queue, and its own form of file descriptor. Select uses fd_set, which is a structure made of bits, each bit is associated with a file descriptor.

The multi-threaded architecture is very different in that each file descriptor gets its own process/thread which is exclusive to it for the running time of the program. This is very costly as the overhead from threads can quickly add up. In Go Go routines are estimated to have about 2kb initial memory allocated to them (Sundarram, 2014). Which may lower the impact of this.

PROCEDURE

This section describes the procedure to carry out the experiment.

Compilation

SELECT

Compilation is fairly standard, however, with the select server, the Go compiler must be modified. The general instructions for installing the Go compiler from source can be found at https://golang.org/doc/install/source . For this test FdSet.Bits was changed from being an [16]int64 to an [2048]int16. While the original select call as implemented by go only handles 1024 connections the new implementation can handle. Once this has been completed select can monitor 131,072 files, well more than what is used in this test.

Once the above has been completed, the scalable server can be compiled with the following

```
$ cd $GOPATH/src/select_scalable_server
$ go install
```

EPOLL

Epoll is much simpler, if desired it may be compiled without modifying the go compiler.

```
$ cd $GOPATH/src/epoll_scalable_server
$ go install
```

MULTITHREADED

Multithreaded can be compiled similar to the above.

```
$ cd $GOPATH/src/multithreaded_scalable_server
$ go install
```

CLIENT

The client is also a similar compilation

```
$ cd $GOPATH/src/client_scalable_server
$ go install
```

Testing

In the documentation, go suggests setting $GOPATH/bin to part of your path. By doing this the programs can be run by doing the following

CLIENT

```
$ client_scalable_server –i 100 –r 100 –d 1000 –c 10000 x.x.x.x:PORT
```

SERVERS

```
$ multithreaded_scalable_server :5000
$ select_scalable_server 5000
$ epoll_scalable_server 5000
```

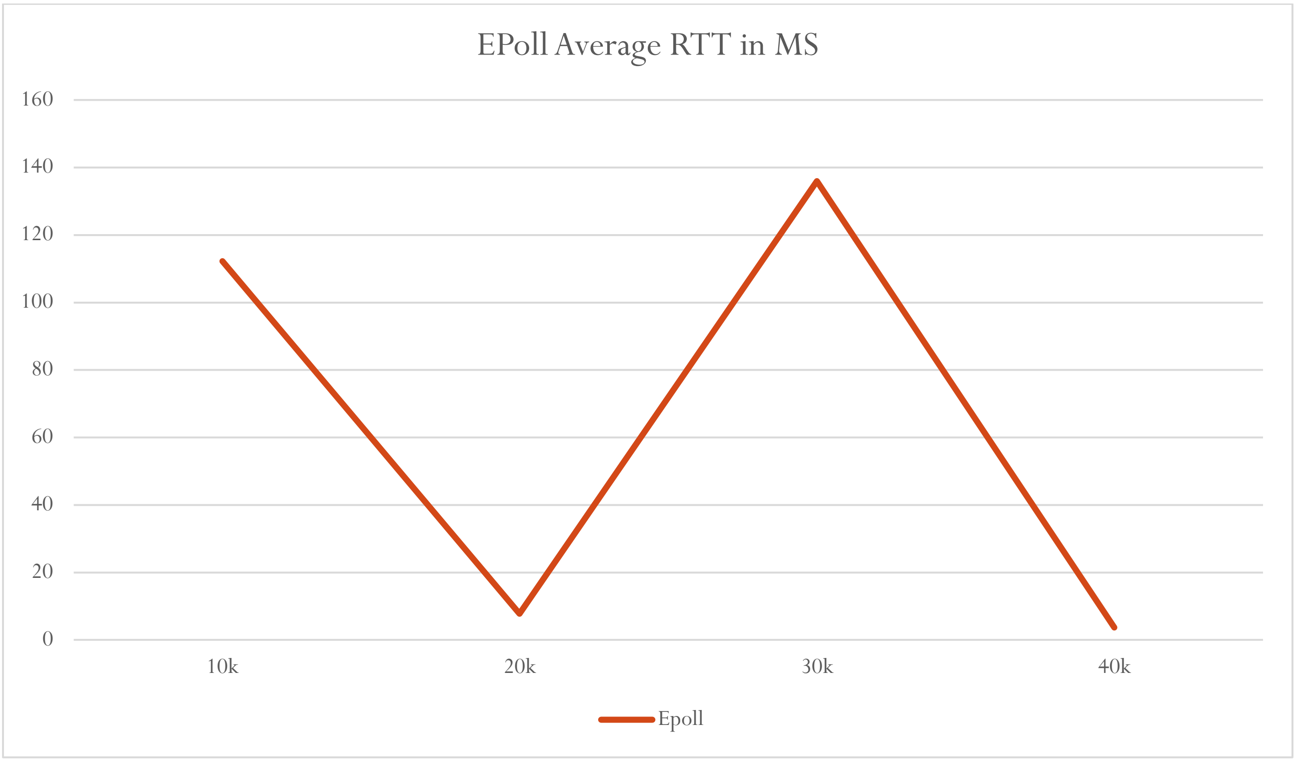
RESULTS & DISCUSSION

Test Parameters

- Server Port : 5000
- Epoll and select threads: 30
- Ammount of data echo'd: 1000bytes
- 1 client machine for each 10k connections

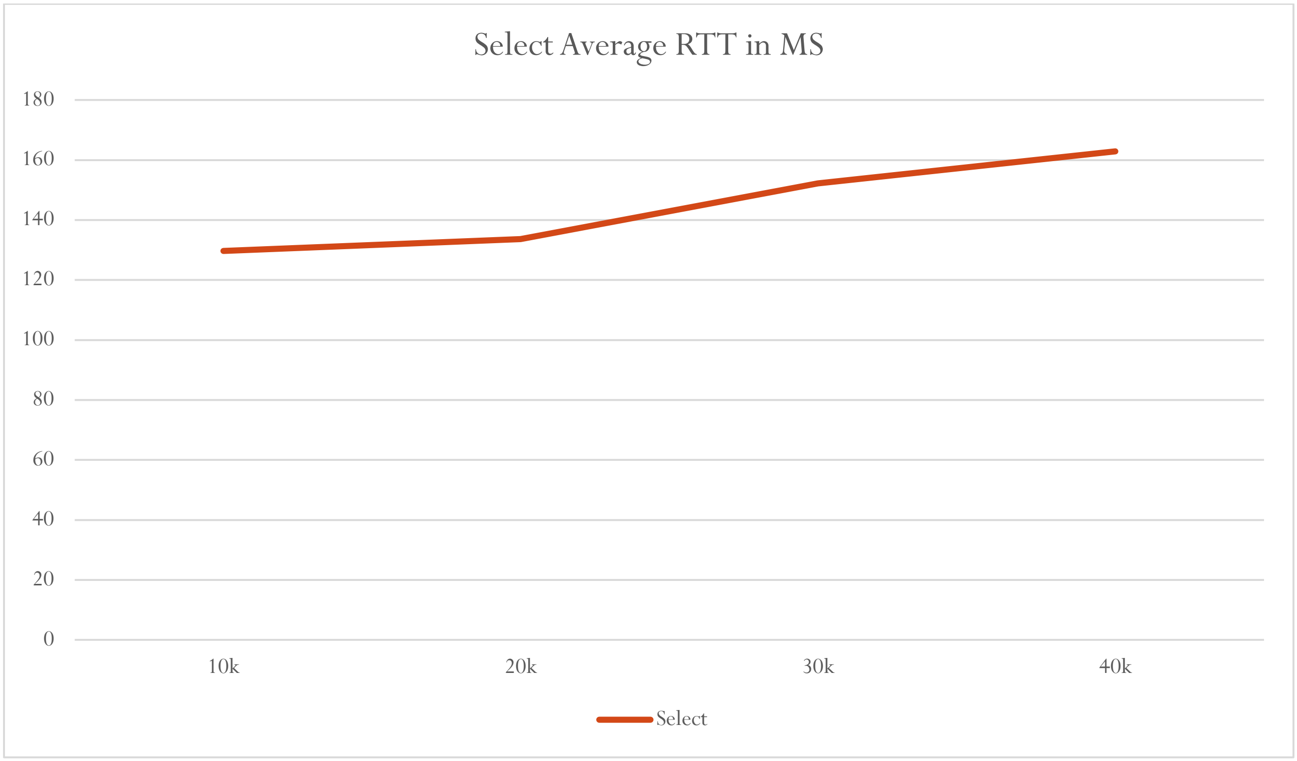
Additional machines were set up prior to starting the machine that results were gathered from. They were set up to allow for them to ramp up and weigh down the server. Once they had established their maximum connections the client was run. This ensured that the client from which statistics were drawn had the closest to the maximum connections from the beginning.

EPoll



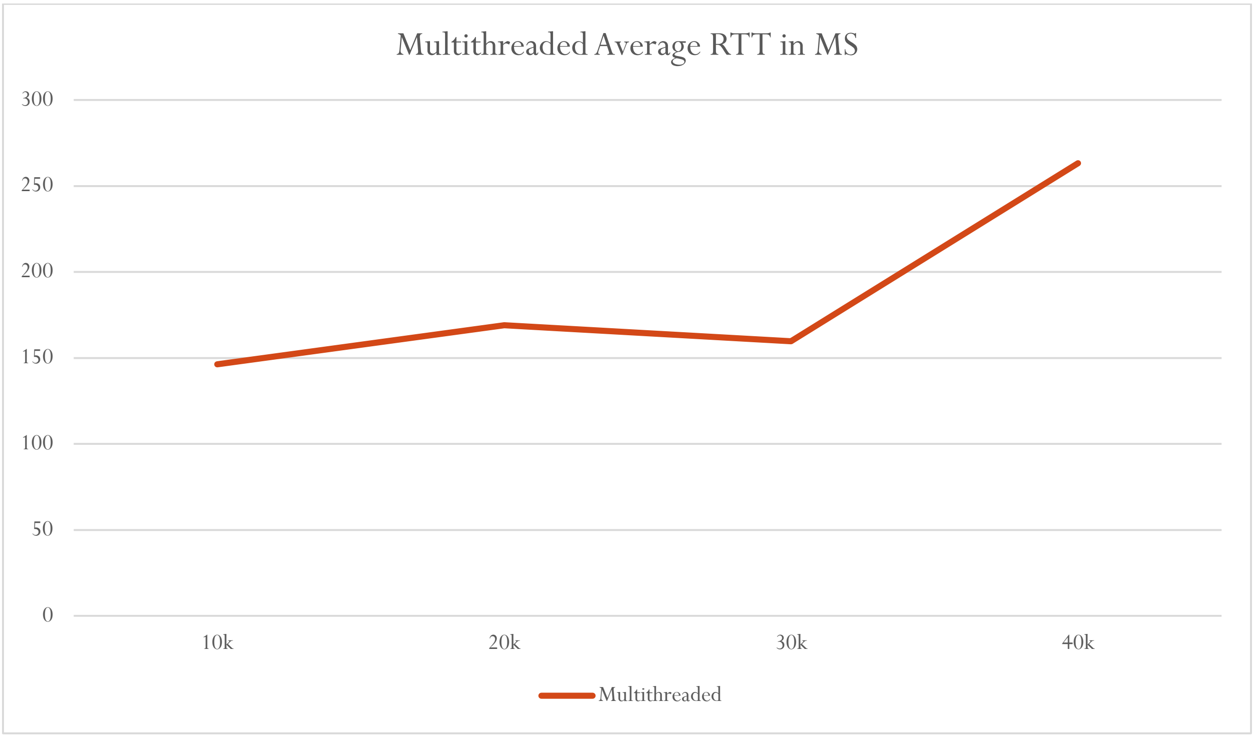
The EPoll Test provided confusing results, the number of incoming connections appeared to have very little effect on the amount of time it took to handle the connections. Assuming the client and the server were working as intended, this may suggest that there was other network traffic which caused the spikes, or that EPoll is very robust and could handle many more connections than the 40K that was used for testing.

Select

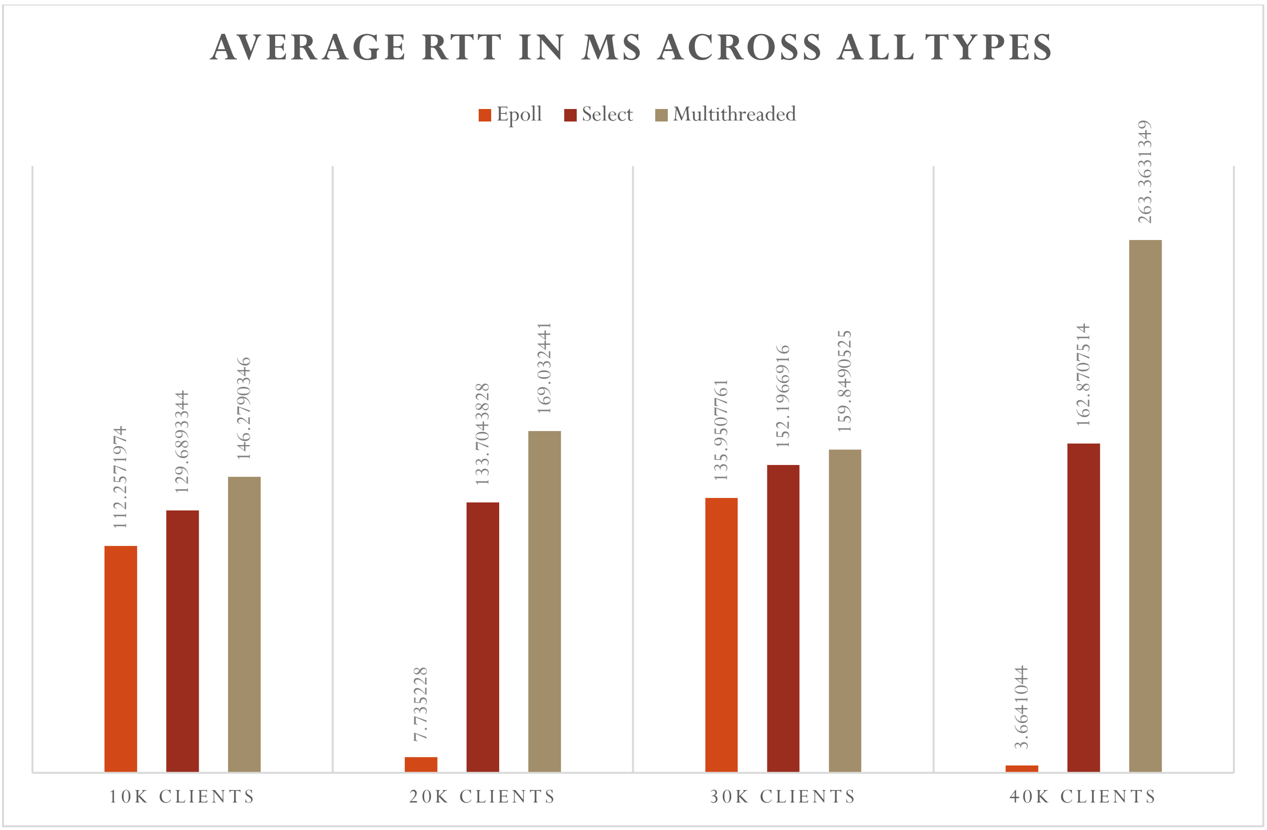


The select test results were more linear which is to be expected. When a low number of clients are used the server is better able to handle the connections it receives. The RTT is much more stable from select than it was for epoll, as these were both tested concurrently, this may rule out other network traffic as being the main cause of delay for epoll.

Multithreaded Server



The multithreaded RTT is very similar to select. This may be because go routines, goes method of concurrency are very light weight and take little time to initialize. Assuming the client and the server were working correctly it could also be due to network traffic. An interesting feature of the trend for the multi-threaded server is that the RTT increases rapidly after 30k. This could suggest that the overhead begins to overwhelm the server at this point.



Number of Connections	**Average RTT**		
	EPoll	Select	Multithreaded
10k	112.2571974ms	129.6893344ms	146.2790346ms
20k	7.7352280ms	133.7043828ms	169.0324410ms
30k	135.9507761ms	152.1966916ms	159.8490525ms
40k	3.6641044ms	162.8707514ms	263.3631349ms

When compared, the results gathered are more or less to be expected, with the exception of the EPoll RTTs. EPoll preforms better than select, which preforms better than the traditional multi-threaded server option. Other than some of the EPoll results which seem like anomalies, the difference seems fairly linear.

CONCLUSION

Based off the findings of this report, EPoll is the fastest to handle a large number of clients. Both select and the multi-threaded approach are also capable of dealing with a large number of clients, and are generally more stable in how they handle the additional clients. The tenacity of the multi-threaded approach is surprising to me. I did not expect them to be so comparable to select; however, the results for 40k would suggest that there does come a point where a multi-threaded approach becomes unusable.

The multi-threaded approach had the additional benefit of being cross platform compatible, and by far the easiest to implement. This suggests that although often thought of as being out of data, there are many use cases where it could perform adequately.

REFERENCES

Kegel, D. (2013, 07 18). *The C10K problem*. Retrieved from The C10K problem: http://www.webcitation.org/6ICibHuyd