COMP8005 - Assignment 2

Frederick Tsang, Clemens Lo
Set 6D
2016-02-15

Design and Performance

State Machine

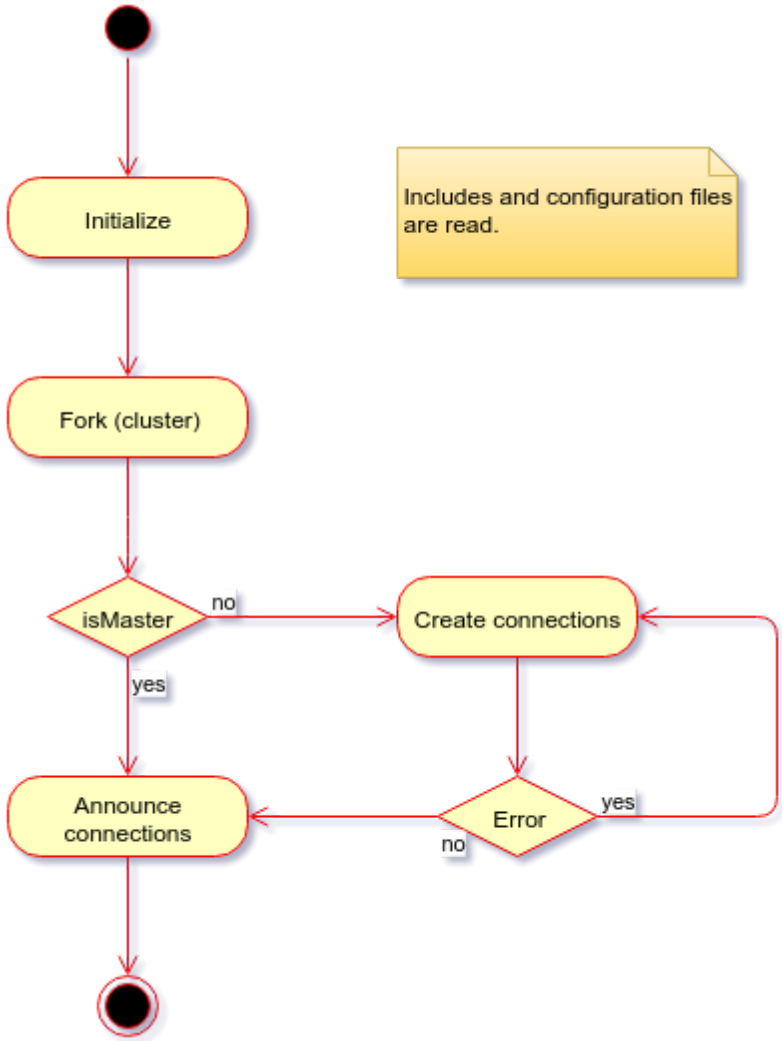
Client

The client will run all referenced modules before running the main code.

The ‘cluster’ module is in charge of forking processes.

The ‘master’ process will spawn ‘worker’ processes that creates connections to the server.

If a connection is prematurely terminated, a connect request will be resent at an exponential backoff rate. This will ensure that the number of connections active respect what is specified in the configuration file.

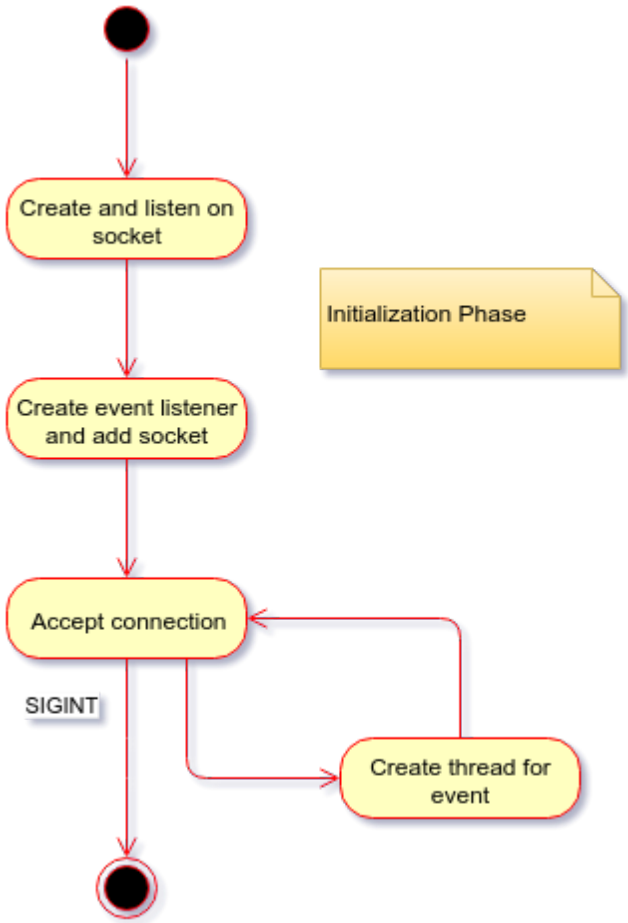


Every second, the ‘master’ process will print the number of active connections to the console.

Servers

Code for a simple echo server

```
require('net')
    .createServer()
    .on('connection', client =>
client.pipe(client))
    .listen({
        host: process.argv[2]
        port: Number(process.argv[3])
    });
```



Similar to the single-threaded echo server implementation above, two of the three servers use an event mechanism to be notified of any incoming connections.

Events

The code above echos the contents sent by the client when there is a 'connection' event. Likewise, the select server uses 'select' and the epoll server uses 'epoll' to notify that there is an event to be read. In the case of the traditional multithreaded server, the event is a blocking call instead.

Multi core Processors

To make use of the current multi core processors, a thread is created to handle the event.

Signals

Due to the server being in an infinite loop, the only way to exit is through signals. Ctrl+C or sending the SIGINT signal will cause the server to close.

Performance Comparison

Server Specifications

- 700 MHz single-core processor
- 256 MB memory
- 100 Mbit network adaptor

Generating data to send

> dd if=/dev/urandom bs=1M count=1 > 1MiB

File sizes being transferred

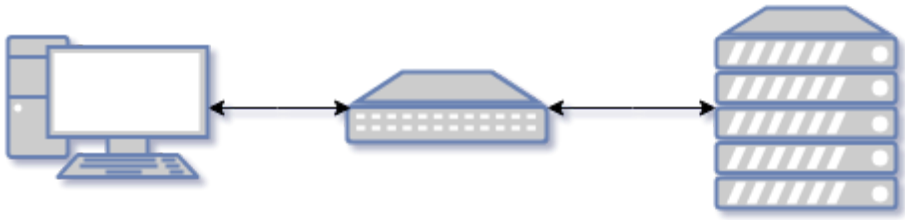
- 1 byte
- 1 kilobyte
- 1 megabyte

Over a 1 Gbit network.

Comparison Strategy

Three graphs for 1B, 1KiB, and 1MiB continuous transfers over 30 seconds will demonstrate the expected performance server 10,000 simultaneous connections. One more graph will be displaying connections that are added over time until a noticeable performance hit is shown.

Topology

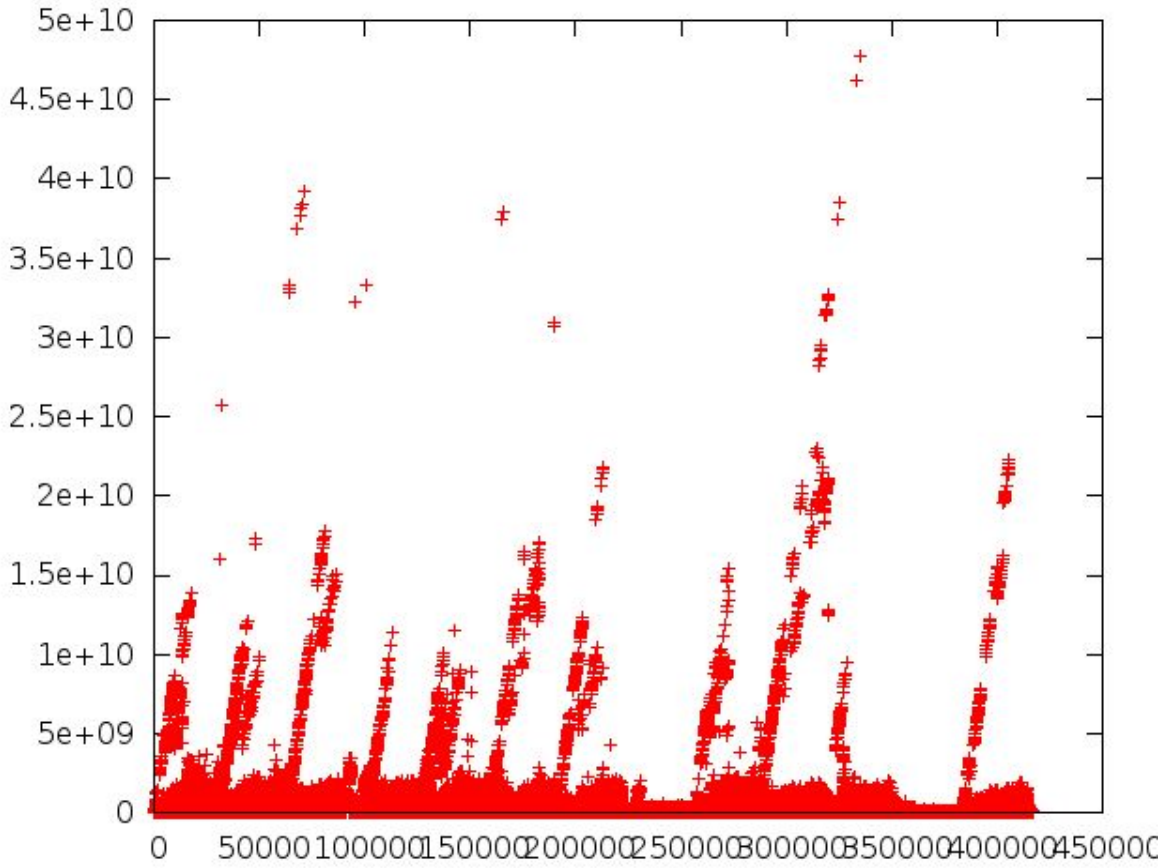


The devices interacting with the server are connected to a switch that connects to the server. While all the devices are connected using CAT6 Gigabit cables, the server only has a 100Base-T ethernet port.

Multi-threaded (Traditional)

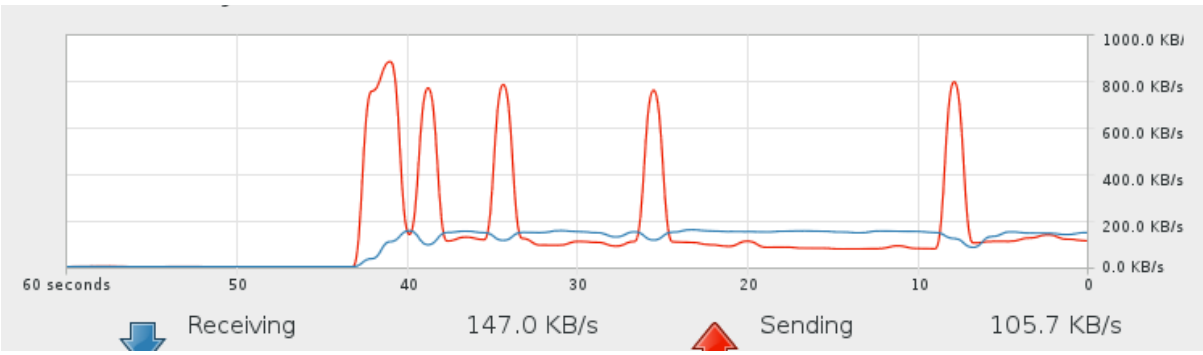
1 Byte Transfer

Log file: multi/1b10k.log



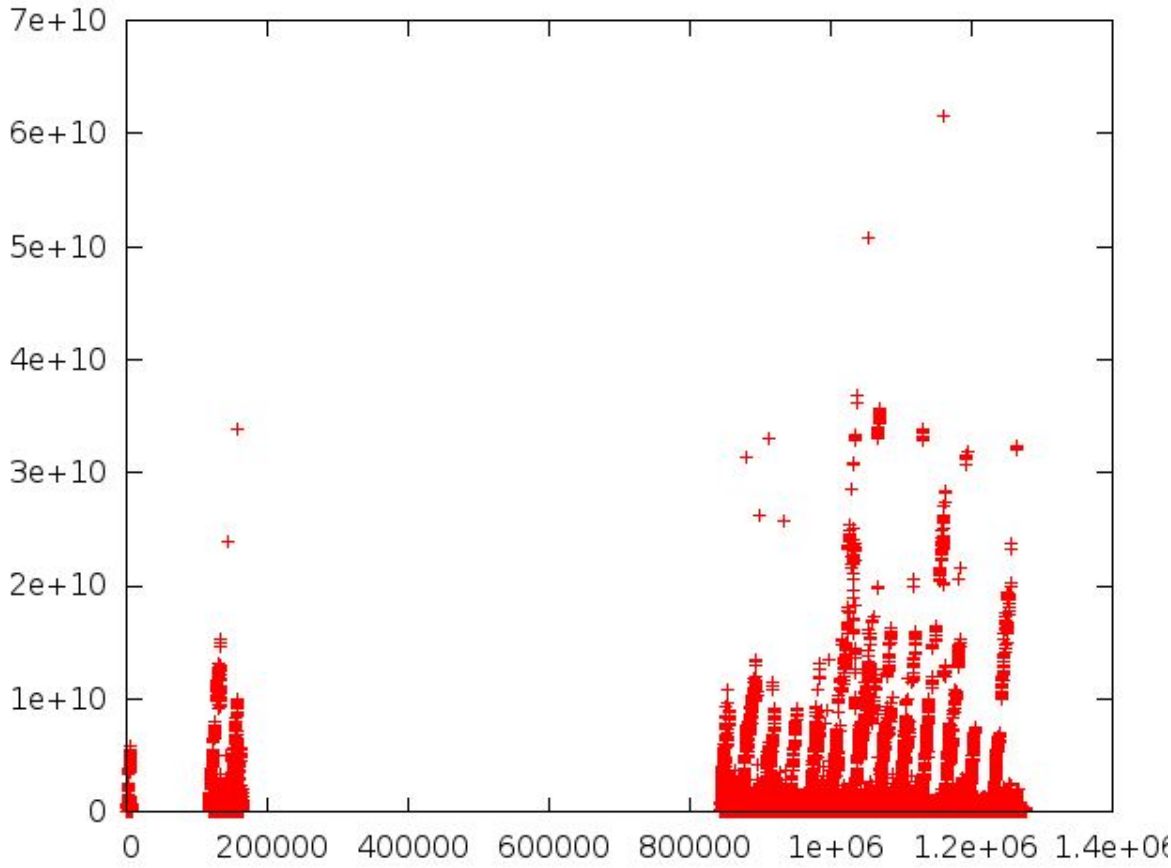
Response Time in nanoseconds over Time in milliseconds

Note: The server was manually terminated after half of the connections were closed.



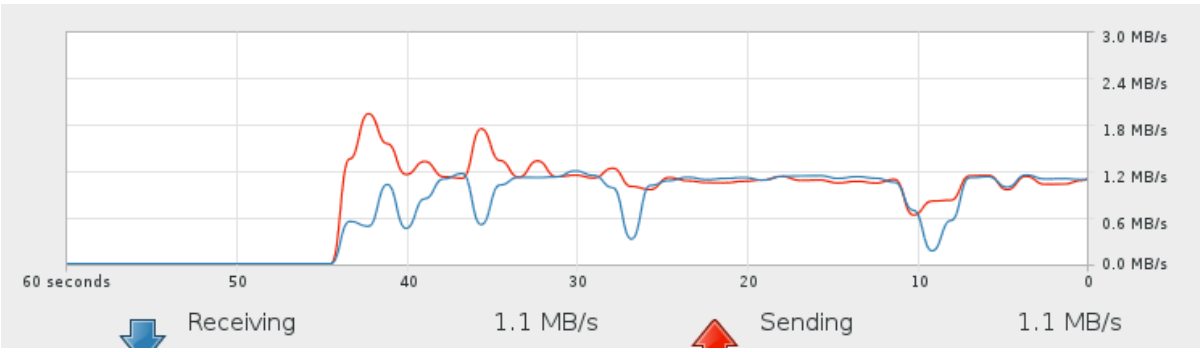
1 Kilobyte Transfer

Log file: multi/1kb10k.log



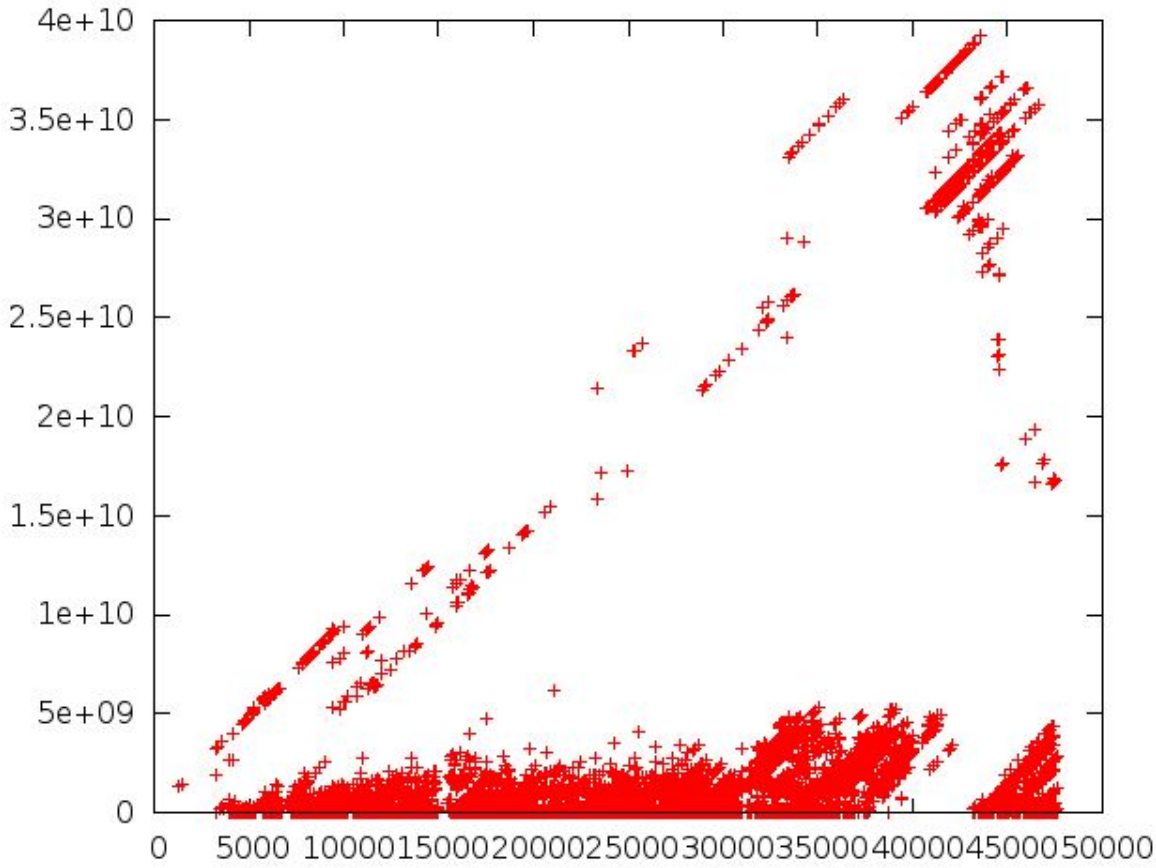
Response Time in nanoseconds over Time in milliseconds

Note: The server was manually terminated after half of the connections were closed.



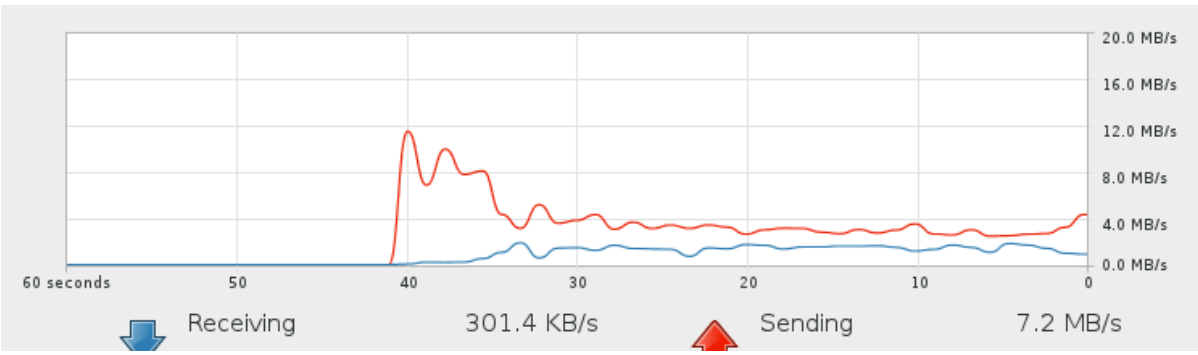
1 Megabyte Transfer

Log file: multi/1mb10k.log



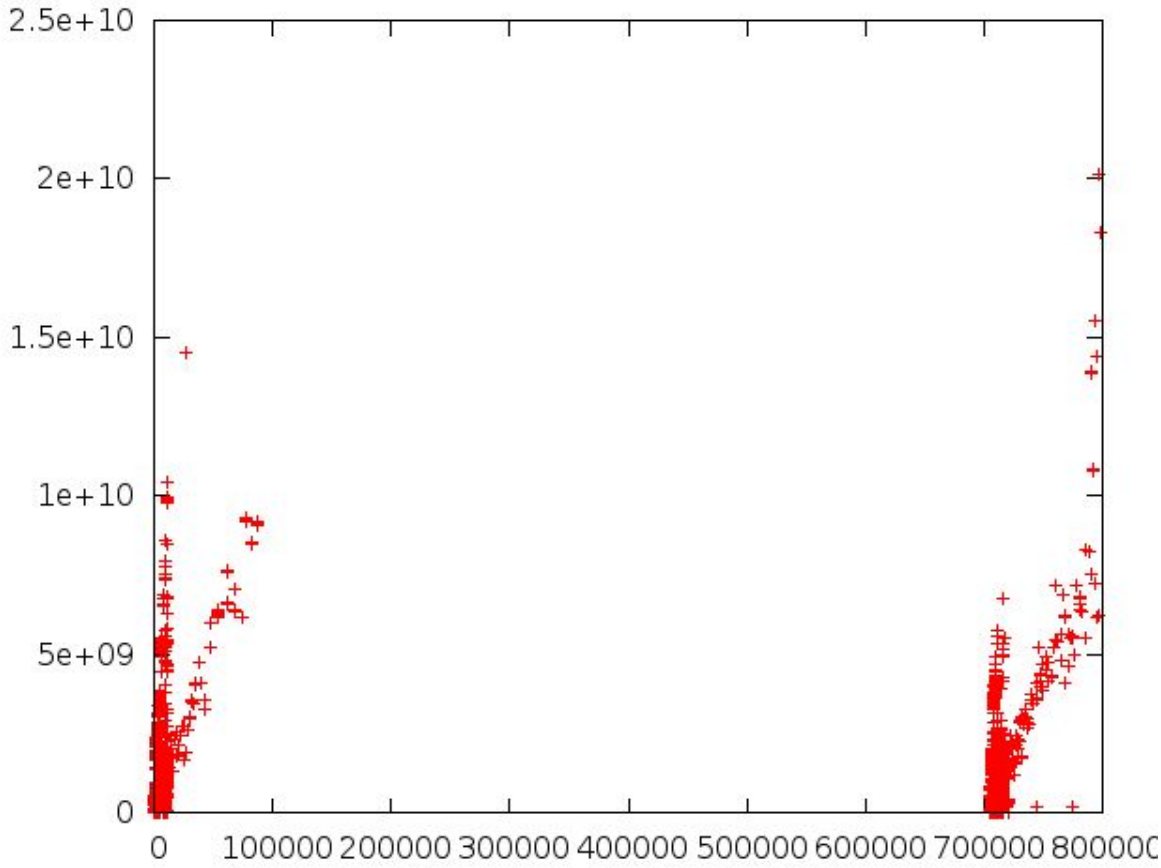
Response Time in nanoseconds over Time in milliseconds

Note: The server was automatically terminated by the system.



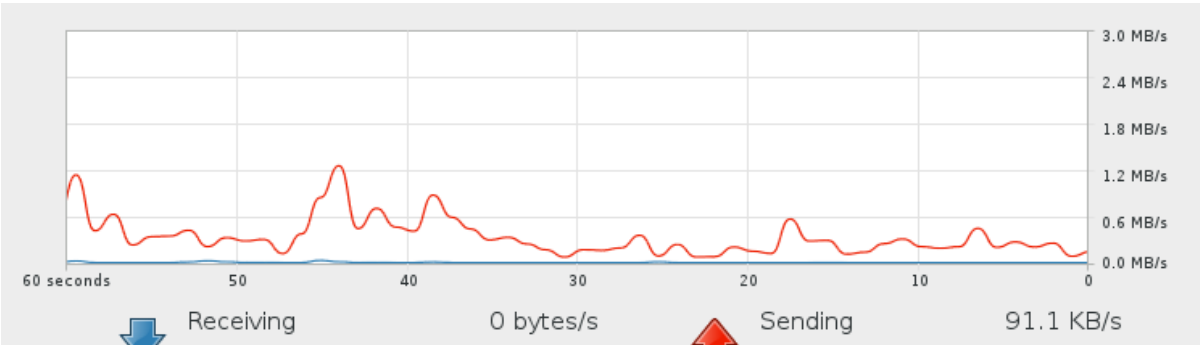
Continuous Transfer (1KB)

Log file: multi/1kbcont.log



Response Time in nanoseconds over Time in milliseconds

Note: The server was manually terminated after throughput collapsed (see below).

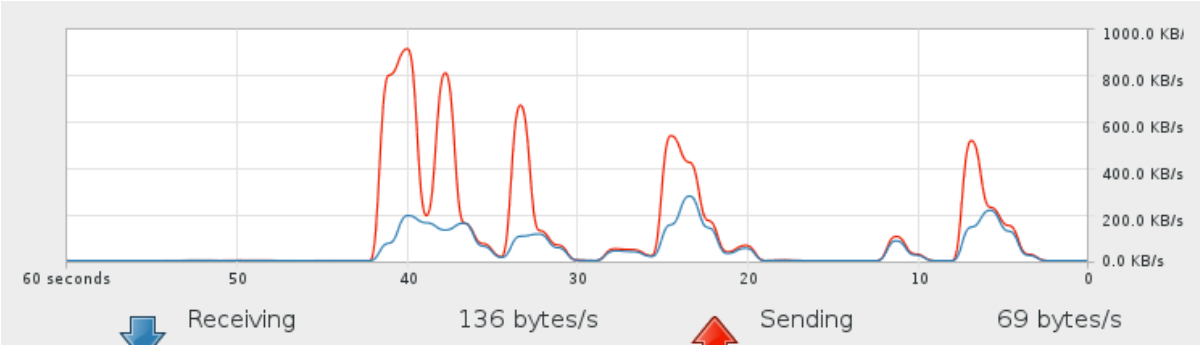


Select

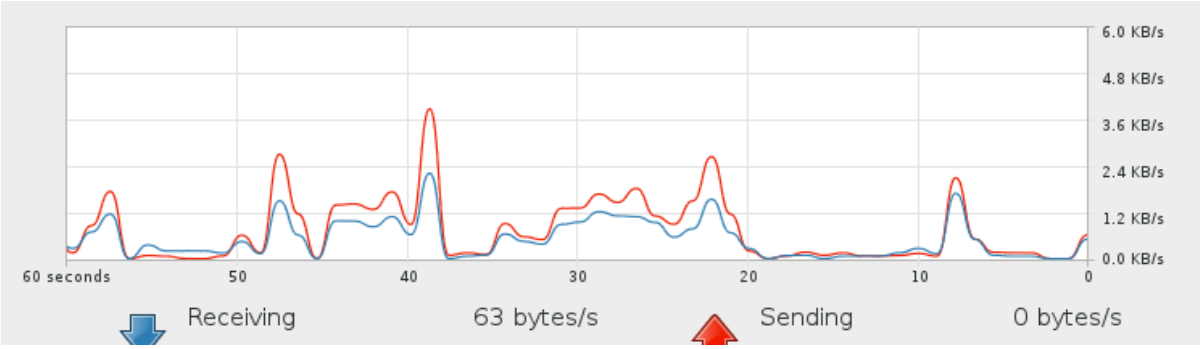
Assuming select does work as advertised, we were not able to write an implementation that could do it justice.

1 Byte Transfer

Note: Could not complete tests.



Initial burst is mediocre. Long pauses completely slow down the transfer.



After 10,000 connections have been made, the transfer speed slows to a crawl and is unable to complete within a reasonable amount of time.

1 Kilobyte Transfer

Note: Could not complete tests.

1 Megabyte Transfer

Note: Could not complete tests.

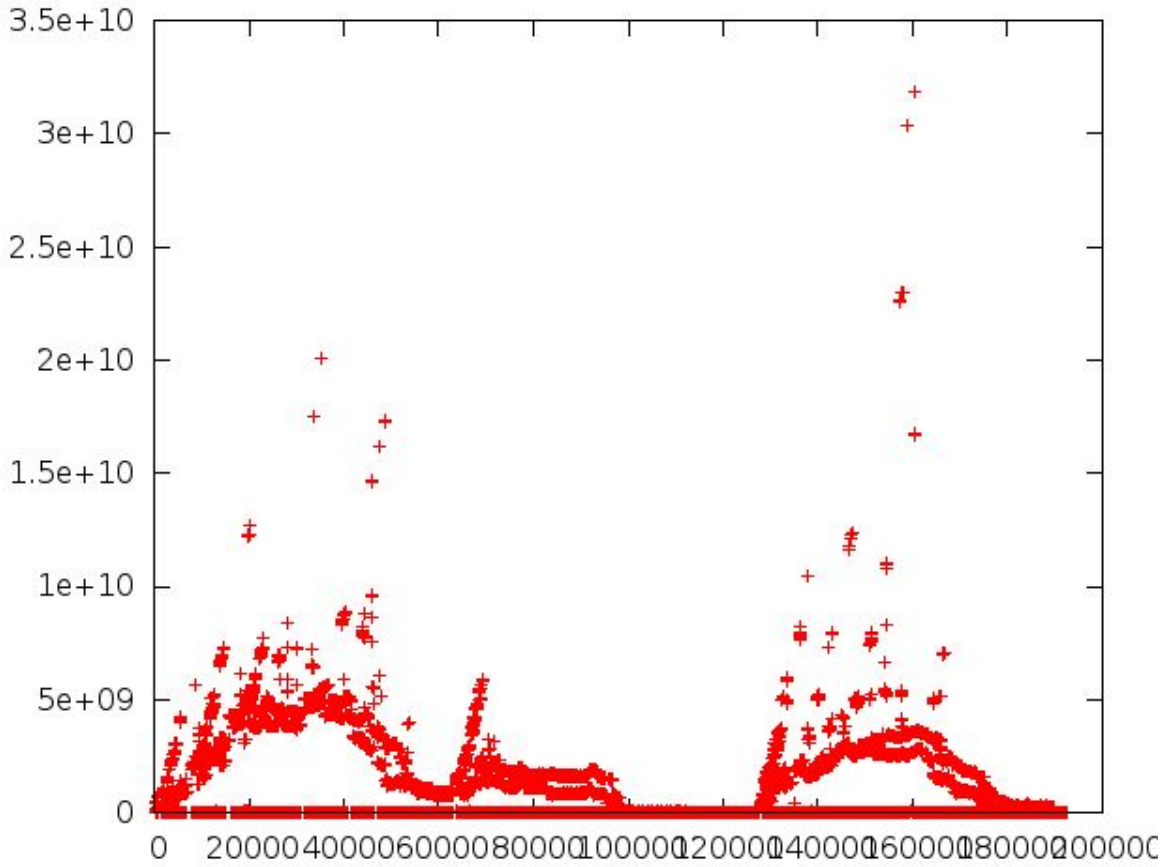
Continuous Transfer (1KB)

Note: Could not complete tests.

Epoll

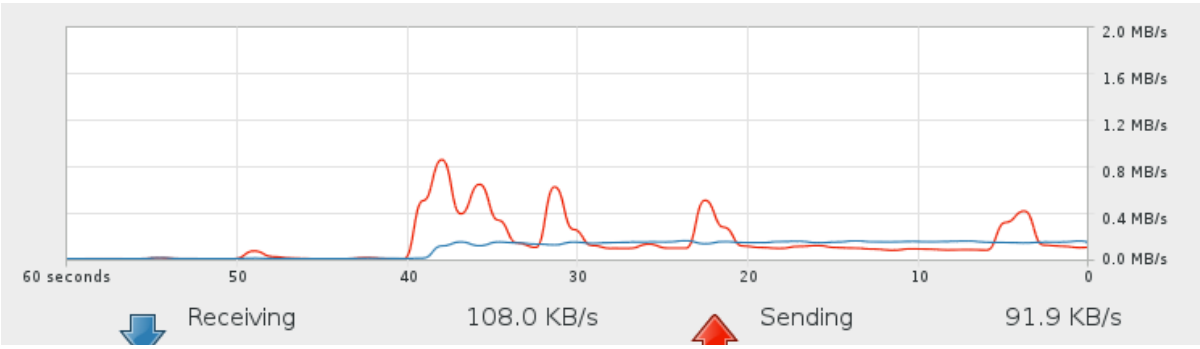
1 Byte Transfer

Log file: epoll/1b10k.log



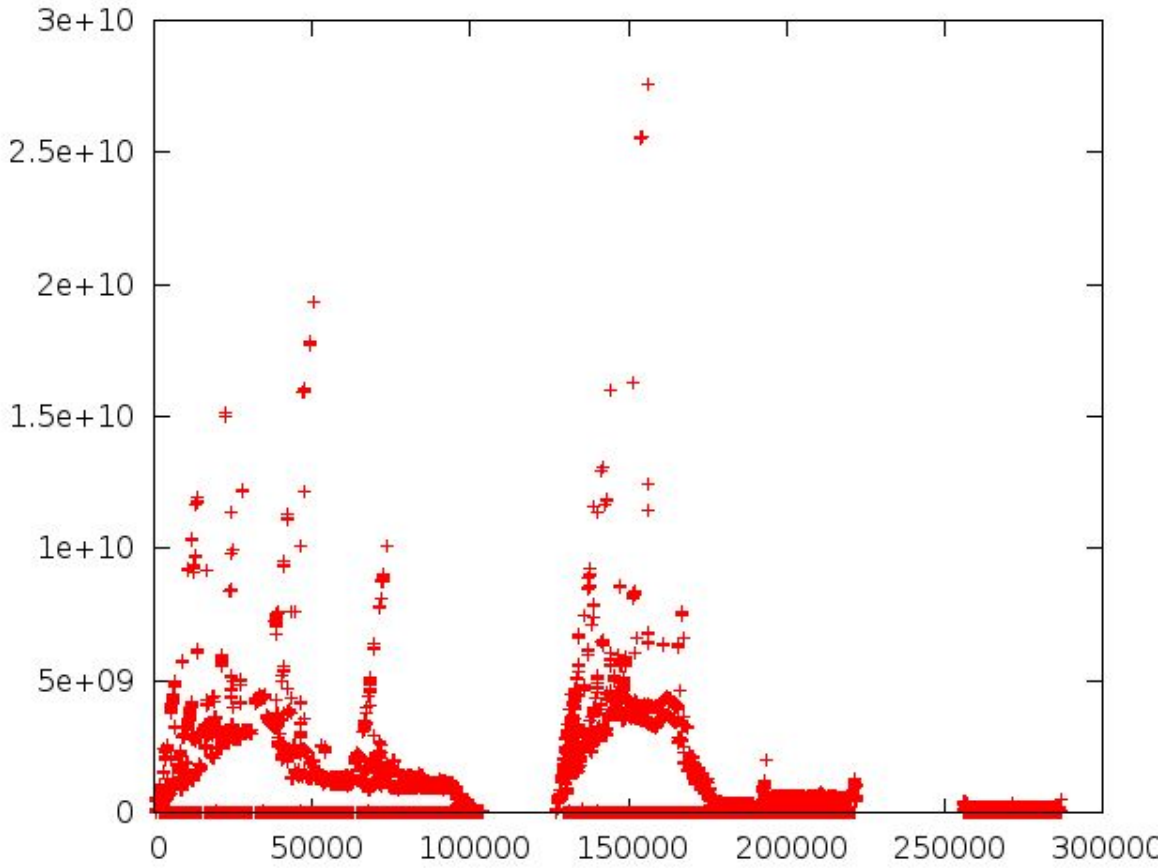
Response Time in nanoseconds over Time in milliseconds

It took over a minute before all 10,000 connections were responded to and closed. Spikes in response time is due to connections retrying after a backoff duration.



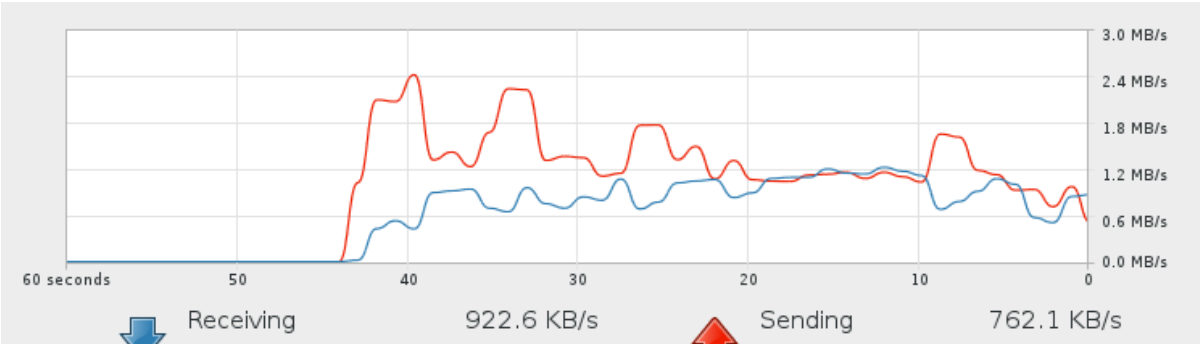
1 Kilobyte Transfer

Log file: epoll/1kb10k.log



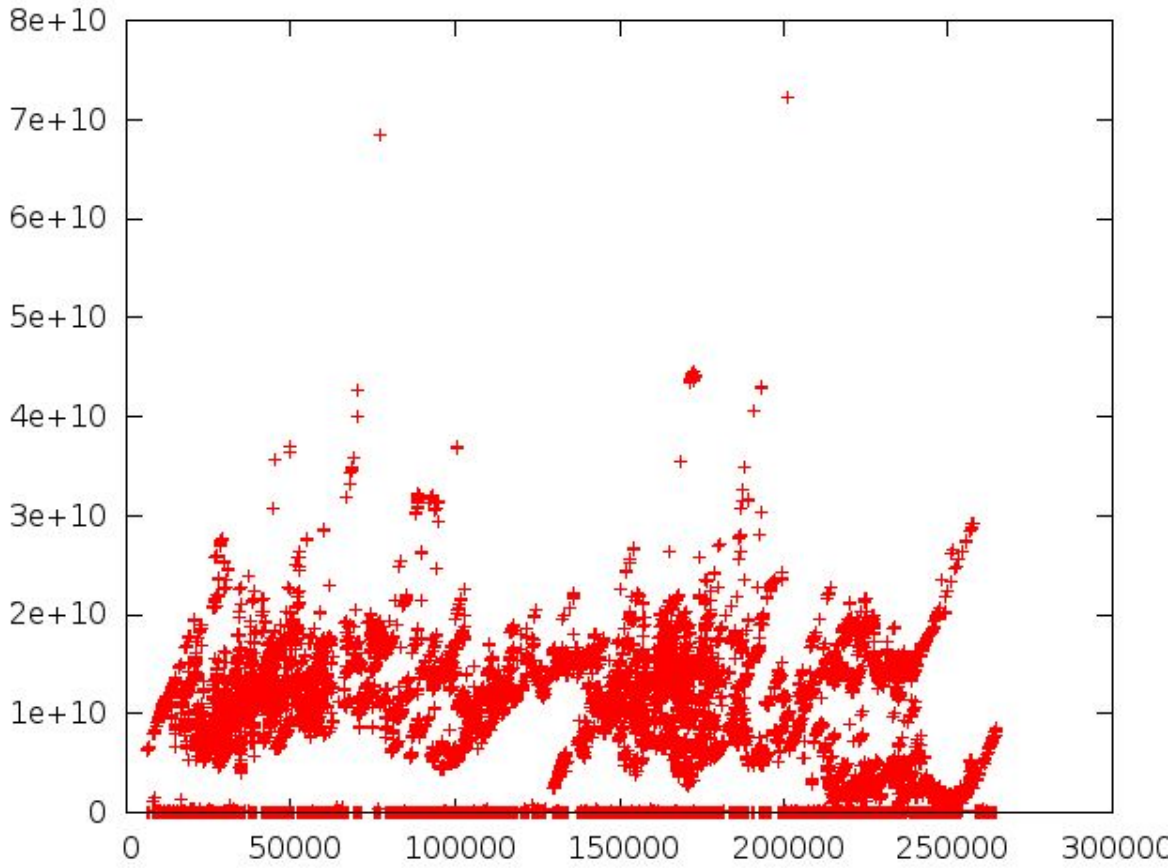
Response Time in nanoseconds over Time in milliseconds

Gaps are noticeable from the increase backoff timeout due to connections repeatedly being rejected.



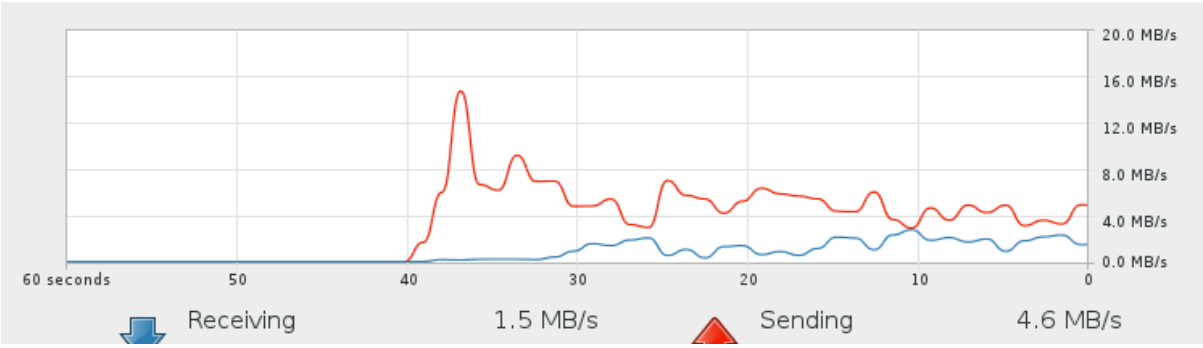
1 Megabyte Transfer

Log file: epoll/1mb10k.log



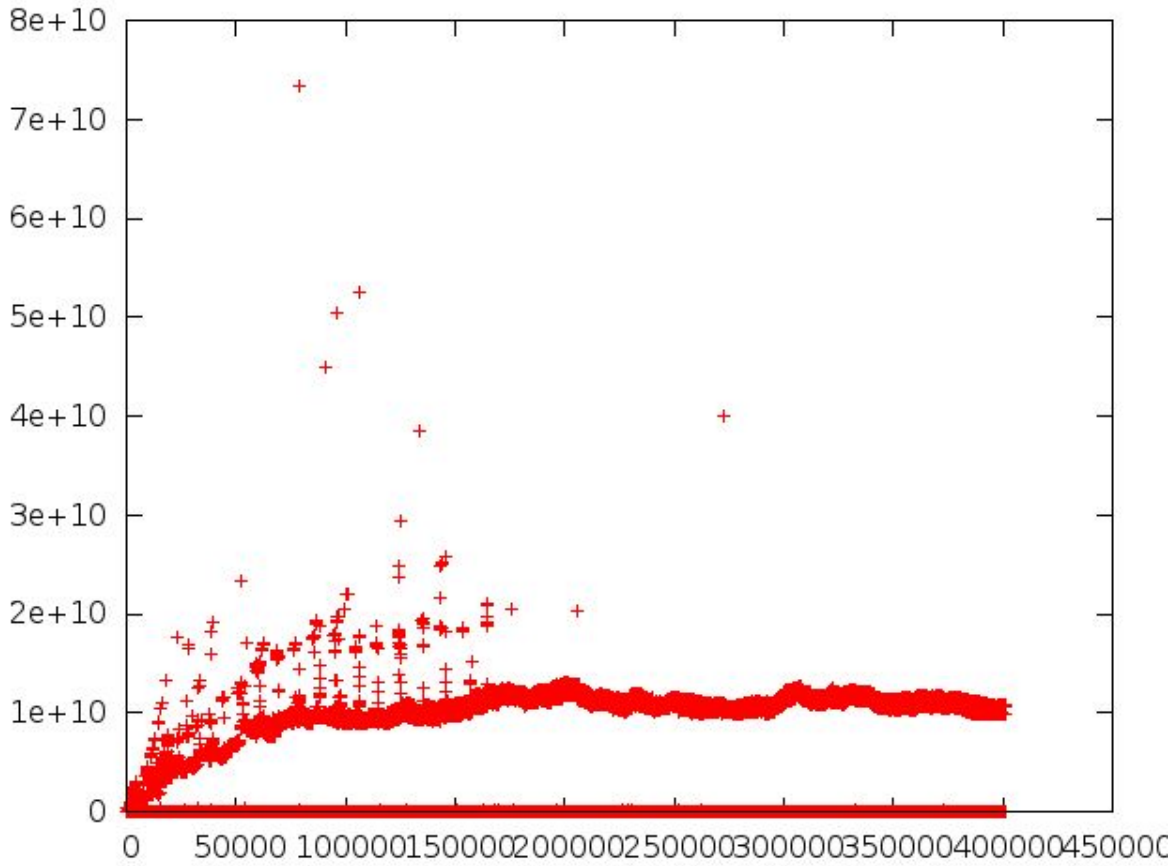
Response Time in nanoseconds over Time in milliseconds

Note: The server was not able to support this test. It starts to terminate connections near the 30 second mark.



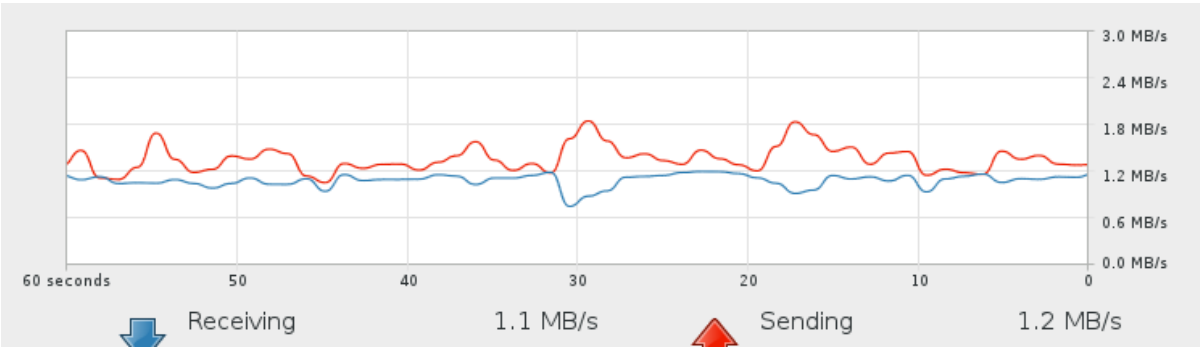
Continuous Transfer (1KB)

Log file: epoll/1kbcont.log



Response Time in nanoseconds over Time in milliseconds

Increase of 1000 connections every 5 seconds. The transfer was manually stopped at 80,000 connections.



The maximum amount of data the server could handle at once seems to be about 3.0 MB/s up and down-loading at the same time. Even with an increased number of connections, the speed is noticeably limited. While the server is able to accept a large number of connections, they're throttled to the point where there is no obvious decrease in response time.

Conclusion

Measuring the response time and rate of transfer is insufficient to provide detailed results. Based on the graphed results of the transfer rate, errors and rejected connections are alarmingly high and should have been included as a part of the comparison. A different approach is necessary. However, given the gap in performance, the traditional multithreaded server is obsolete compared to the newer event-driven epoll.