Technical Report

Version 1.0

Eric Tsang, A00841554, 6D

|  |
| --- |
| Table of Contents [Table of Contents 1](#_Toc443238618)  [1 Summary 2](#_Toc443238619)  [2 Introduction 2](#_Toc443238620)  [3 Theory 2](#_Toc443238621)  [Procedure 3](#_Toc443238622)  [3.1 Compiling 3](#_Toc443238623)  [3.2 Testing 3](#_Toc443238624)  [4 Results & Discussion 4](#_Toc443238625)  [4.1 Low client count, low duration tests 4](#_Toc443238626)  [4.2 High client count, high duration tests 5](#_Toc443238627)  [5 Conclusion 6](#_Toc443238628) |

# Summary

In Linux distributions, there are different ways to monitor and service a large number of file descriptors for an application: using threads, select, or epoll. Each of these methods operate differently. This report discusses the strengths and weaknesses of all three methods.

The performance of threads is more predictable and consistent than the other methods, however it takes up significantly more memory than the other methods. Select takes a set of file descriptors, and unblocks when it detects that a specific operation can be executed on any of the file descriptors in the set without blocking. Epoll is similar to select, but unblocks when a specified event occurs on any of the file descriptors in the specified file descriptor set.

In practice, epoll yields poor performance under a low load. select is able to serve clients at a much higher rate than epoll, but has a poor response time to events. threads are able to serve about the same amout of clients as select does, and has a response time similar to epoll's, which is good.

# Introduction

This report compares the strengths and weaknesses of different scalable server srchitectures on the Ubuntu 14.04 operating system in a multi-core environment. The three architectures being compared are the threaded server, select server and epoll server. They will be compared in terms of how many clients they can handle simultaneously, how many requests they can serve in a timeframe, and the time each takes to serve a request.

# Theory

In Linux operating systems, there are various methods used to monitor and service multiple file descriptors simultaneously. These methods include milti-threading, using select, or using epoll.

A pre-thread-spawning multi-threaded server is arguably better than a pre-process-spawning multi-process server:

* Threads generally initialize faster and switch faster than processes because each thread is usually smaller than a typical process.
* A multi-threaded server will be able to serve more clients simultaneously than a multi-process server because memory is a finite resource, and threads are smaller than processes.
* Thread performance is most significantly negatively impacted by thread synchronization; however, the only thing the threads share in this case is a semaphore and server socket. None of the threads need to allocate data on the heap, so the risk of a negative performance impact should be minimal.

Multi-threaded server:

* Has a main thread that makes sure there is always n threads waiting to accept a new connection. If less than n threads are waiting to accept a new connection, it spawns new worker threads until the condition is met.
* Each worker thread accepts a connection from the server socket, then serves the single accepted connection until it is closed and the thread terminates.
* Poor memory efficiency when compared to epoll and select because a new thread is created to serve each client which is much less memory efficient than simply adding a file descriptor to a data structure for monitoring as is in epoll and select.

Select & epoll severs:

* Select is level-triggered, while epoll is edge triggered; in the case of select, a file descriptor is marked if operations can be performed on it without blocking where as in epoll, a file descriptor is marked when new data arrives.
* Socket descriptors are put into a file descriptor set which are monitored by the select or epoll functions, which blocks until activity is detected in any of the file descriptors.
* Once activity is detected on any of the file descriptors, the program loops through all file descriptors, and services them as necessary.
* In these implementations of the epoll and select servers, the server socket is shared among n processes, each with their own set of file descriptors to poll in order to take advantage of the multiple cores of the system.

# Procedure

This section describes the procedures followed to carry out the various experiments used to measure the different performance aspects between multi-threaded servers, select servers and epoll servers.

## Compiling

This procedure is used to compile the modules for this project:

1. Open a Terminal. Navigate to ./source/ (relative to the project's root directory)
2. Make a clean compile of all modules of the program.

$ make clean

$ make epoll\_clnt

$ make epoll\_svr

$ make select\_svr

$ make thread\_svr

## Testing

This procedure is for running the programs, and testing how well they can handle a large number of clients (procedure 4.1 should be completed on all machines involved with the test prior to performing this procedure):

1. Open a Terminal on the server computer. Navigate to ./source/ (relative to the project's root directory)
2. Execute any server (epoll\_svr.out, select\_svr.out, thread\_svr.out) on any port, with as many processes (or idle threads) as needed.

$ ./[server] -p [port] -n [process / thread count]

1. Open a Terminal on any number of client computers. For each client computer, execute the client program:

$ ./epoll\_clnt.out -h [server ip] -p [server port] -n [num processes] -c [num clients] -d [data to send] -r [requests to make per connection] -t [test duration in milliseconds]

# Results & Discussion

All tests were performed in the following environment:

* 1 server computer, 4 logical processors
* 1 client computer, 4 logical processors
* client and server computers are directly connected via a dedicated ethernet connection.

More test specific details are provided in the appropriate sub-section.

The following are results of various tests.

## Low client count, low duration tests

The test parameters are as follows:

|  |  |
| --- | --- |
| * server port: 7000 * epoll & select server processes: 4 * thread idle threads: 300 * client processes: 4 | * data echoed: "hello" * requests per connection: 1 * number of clients: 500 - 4000 * test duration: 1000 milliseconds |

The following graph plots the average duration of each connection:

As shown above, both the epoll and threaded server suffer a steady increase in service time, while the select server's service time dramatically increases after 3.5 thousand connections are established concurrently.

The following graph illustrates the number of requests served for the duration of the test:

As shown above, the select and threaded servers serve a consistent number of clients in the given test duration despite the increasing number of simultaneously connected connections. in contrast, epoll oddly improves when more clients are connected.

From these tests, one may conclude that the thread server is overall the most desirable because of its predictable service times, and consistency in serving requests. the select server seems to match with the thread server in terms of clients served, but it has an anomaly in average service time, which is undesirable. epoll's performance under the test's conditions are simply unacceptable.

## High client count, high duration tests

The test parameters are as follows:

|  |  |
| --- | --- |
| * server port: 7000 * epoll & select server processes: 4 * thread idle threads: 300 * client processes: 4 | * data echoed: "hello" * requests per connection: 1 * number of clients: 4000 - 28000 * test duration: 10000 milliseconds |

The following graph plots the average duration of each connection:

As shown above, the service times for all servers are similar when they are under heavy loads.

The following graph illustrates the number of requests served for the duration of the test:

As shown in the above graph, the more clients that are simultaneously connected, the less overall requests are served.

from the abode test results, one can draw to the conclusion that under very heavy loads, all server architectures struggle just the same, and provide generally equal service.

# Conclusion

Based on tests 5.1 and 5.2, thread servers are the most predictable overall, but take up more memory than the other kinds of servers at heavy loads. The select server is also capable of serving many requests despite the number of concurrent connections. The epoll server oddly is only able to serve very few clients compared to the other servers under low load, but improves as the load increases.