UNIVERSITAT DE BARCELONA

Treball Final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Benchmarking Input/Output Multiplexing Facilities Of The Linux Kernel

---

Autor: Francesc Bruguera i Moriscot

Director: Dr. Lluís Garrido
Realitzat a: Departament de Matemàtiques i Informàtica
Barcelona, 4 Febrer 2019

Acknowledgements

This thesis would not have been possible with the help and support of many individuals.

I own my deepest gratitude to my supervisor, Dr. Lluís Garrido, for helping and encouraging me during all the journey.

I was introduced in late 2016 to the world of performance optimisation and scaling issues at work. My thanks to my former colleagues, Antonio and Gabri, for what we learned and for the work we did.

Finally, of course, I would like to thank my family for their support at all the times, always letting me chase my dreams.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.
– Alan Turing

Abstract

In the last few decades, concurrent connection processing needs have increased, and will continue to do so – both server-side and client-side. There has been a big push in the industry towards solutions that improve the efficiency of all the pieces involved in this task.

This Bachelor's Thesis focuses in a foundational feature for many programs: how to handle more than one connection at the same time. It is an apparently simple task –whether you ask a computer engineer or a computer user. However, it can be done in different ways, each of which has different efficiency consequences.

This document will explore the main ways to be able to handle concurrent connections in Linux computer systems. In addition, we will be experimentally analysing the efficiency of the main methods for accomplishing multiple concurrent connections: using different threads or using one of the three I/O multiplexing tools (`select`, `poll` and `epoll`) provided by the Linux kernel.

Resum

Al llarg de les últimes dècades, han anat augmentant les necessitats de processar connexions concurrents, tant en servidor com en client. L'indústria ha destinat recursos i esforços en solucions que milloren la eficiència de totes les parts involucrades en la tasca.

Aquest Treball Final de Grau es centra en una qüestió molt fonamental dels programes d'avui en dia: com aconseguir fer més d'una connexió al mateix moment. Una tasca aparentment senzilla –tant si preguntes a un enginyer informàtic, com a un usuari– però que es pot fer de diferents maneres i amb diferents conseqüències en termes d'eficiència.

Aquest document explora les principals maneres de fer-ho, en sistemes Linux. A més, analitzarem experimentalment l'eficiència dels principals mètodes per a aconseguir més d'una connexió a la vegada: utilitzant diferents fils, o utilitzant una de les tres eines de *I/O multiplexing* (`select`, `poll` i `epoll`) disponibles en Linux.

Resumen

En las ultimas décadas, han aumentado las necesidades de procesamiento de conexión concurrentes – tanto en los servidores como en los clientes. La industria ha destinado recursos y esfuerzos en soluciones que mejoran la eficiencia de todas las partes involucradas en la tarea.

Este Trabajo Final de Grado se centra en una cuestión fundamental de los programas de software actuales: como conseguir hacer una más de una conexión simultánea. Es una tarea aparentemente sencilla –tanto si preguntas a un ingeniero de software como a un usuario–, pero que se puede hacer de maneras distintas y con diferentes consecuencias.

Este trabajo gira en torno al análisis experimental de la eficiencia de los cuatro métodos principales para conseguir más de una conexión simultánea en Linux: usando distintos hilos o usando alguna de las herramientas de I/O multiplexing (`select`, `poll` y `epoll`) disponibles en Linux.

Contents

1 Introduction

1.1 Context and Motivation

In the last few decades, there has been a significant increase in Internet usage – with all its consequences: there are 4 billion connected users world wide, use of Internet bandwidth is increasing and more daily activities are moving to the digital world. This translates to online services that have to handle more concurrent users than ever. It is expected that this trend will continue, with even more connected devices in each household.

This has motivated a quest for performance engineering in the field. Big technological companies have spent significant resources into improving the efficiency of their systems for scalability and cost optimisation reasons. They contributed back to the Computer Science community by sharing their findings and solutions, both theoretically – by presenting the solutions in papers, reports and conferences, and sometimes, even with open source contributions.

Back in the beginning of the century, once the Y2K[1] bug had been mostly mitigated and forgotten, it started appearing a similar concept, C10K, across tech spheres. It is the problem of having a server handle 10 thousand concurrent connections.

With the advent of Web 2.0, there was a lot of emphasis in building products, technologies, frameworks and libraries that were *web scale*-ready. This basically meant having products that were able to scale horizontally *infinitely*.

In addition, during the past few years, a variety of new programming languages have become popular. They pretty much have in common that they are focused in performance, safety and developer productivity. This has been one of the results of the quest for performance engineering mentioned above. Go, Swift, Kotlin, and Rust are good examples of that. Some of them have gone

[1] For context, the Y2K bug was related to storing years with two digits. For example, year 1999 was represented as `99` but, year 2000 as `00`. This caused problems in lots of software once it overflowed: such as bad orderings, crashes due to not accepting new dates, etc.

as far as offering new concurrency primitives, such as co-routines or *green threads* to support building highly-concurrent services.

If this new concurrency primitives are displacing threads, at least user-facing and in high-level programming languages, how I/O multiplexing is being handled?

Answering this question is one of the reasons for this Bachelor's thesis project.

This project focuses on the underlying implementation of I/O multiplexing in the Linux kernel. It is a foundational part of today's programs, and something that every developer (and user) takes for granted. Turns out that there are different ways to achieve the same result: requesting and receiving data over the network concurrently.

These different ways were not designed at the same time. The oldest one was designed in the mid 80s. However, there are new approaches to this problem appearing. Both in user space and in kernel space.

As an example of new approaches in user space, there is a research group, called Project Loom[1], in the OpenJDK community investigating new concurrency primitives for Java, fibers in this case. The effort is to be able to support running millions of fibers on a few kernel threads. They are going as far as trying to work around calls that would block the underlying kernel-thread (such as an I/O operation) so that it blocks a fiber but not the whole thread. As an example of a new approach kernel side, it has been as recently as this past year, by combining I/O multiplexing with asynchronous I/O[2].

Non-blocking I/O (NIO) is deeply related to I/O multiplexing and it has been touted as *the way* to get past C10K.

It turns out that it is used under a lot of software, frameworks and tools both used to build scalable systems and for casual usages. It is the type of technology that lots of programmers use without knowing it – and even sometimes, even without understanding it because it is hidden under numerous abstraction layers. This project also explores this concepts, its usages and its perfor-

[2] Asynchronous I/O is outside the scope of this project. It is based on dispatching `read` or `write` calls that return immediately and the program is actively notified when data has been read or written.

mance characteristics.

1.2 Objectives

The main aim of this Bachelor's thesis is exploring the underpinnings of Input/Output multiplexing in Linux. As I/O is a large component of kernels and, particularly, of the Linux kernel, we will be focusing on network I/O across the project.

The main theoretical objectives of this project are understanding:

- How Linux facilitates I/O multiplexing
- The similarities and differences those different implementations have
- Which benefits NIO offers when compared to blocking I/O

As it is covered in the theoretical part (section 2), there are multiple ways to achieve I/O in multiple connections at the same time. We will focus on the `threaded` approach – based in blocking I/O. We will also study `select`, `poll` and `epoll` for non-blocking I/O. In this thesis, all these different methods will be described and compared in terms of its features, limitations and performance characteristics.

To understand the performance characteristics we will be benchmarking an implementation for each approach and we will compare the results.

We will be focusing mainly on benchmarking the throughput it can be achieved using each method. We will also consider the CPU usage implications of each one.

So, there are two main experimental objectives for this project:

- Building a benchmarking suite for each different I/O multiplexing method
- Being able to experimentally determine performance characteristics of each method and compare it with the theoretical performance characteristics

1.3 Organisation

This thesis is divided in two parts:

- A theoretical part (section 2) that describes general Linux Input/Output concepts, describes I/O multiplexing techniques and compares its features and limitations
- An experimental part (section 3) that benchmarks the performance of the described I/O multiplexing techniques and reports the results

1.4 Project Timeline

The timeline of the project has been defined by setting a monthly goal and it has been the following:

- **Month 1 (Sept)**: Define general project structure and goals.
- **Month 2 (Oct)**: Research the state of the art in Linux I/O, gather theoretical knowledge to continue the project to deepen the general project definition from Month 1.
- **Month 3-4 (Nov-Dec)**: Define and implement experimental tests; gather result data.
- **Month 4-5 (Dec-Jan)**: Extract conclusions from the results and write thesis.

2 Technical Background

2.1 Input/Ouput

Input/Output (I/O) is the communication between a computer system and other devices. For example: reading/writing a file from disk, sending packets through the network, etc.

It is one of the foundations of operating systems. There are multiple I/O subsystems in modern operating systems: storage, keyboards, network, pipes, etcetera.

I/O in Linux is performed using the standard POSIX system calls for it: `read` and `write`.

2.2 Linux Networking

"Everything is a file" is one of UNIX principles. And this also happens on the networking side.

Like other UNIX-based kernels, Linux provides networking support through Berkeley (BSD) Sockets. A socket is an abstract representation for the local endpoint of the connection.

Sockets are used through the usual system calls to perform I/O operations `read()`, `write()`, and some special ones that handle socket specific aspects, such as `socket()`, to create one, or `connect()` to connect a local socket to a remote socket, thus establishing a *pipe* where data can be read from and written to.

BSD Sockets were first released in 1993 in the 4.2BSD Unix.

2.3 Blocking and Non-blocking Input/Output

When using the standard system calls described above, operations block. This means, for example, a call to `read()` waits until there is data to be read from

the socket to continue the execution of the program.

Although this simplifies the structure of the program, this comes at a cost. In this day and age programs usually make more than one concurrent connection to a web service. How it is handled, if `read()`ing blocks until there is data?

There two different possibilities to handle concurrent connections, which will explore into detail in the following sections:

- Threaded Input/Output
- Non-Blocking Input/Output

2.3.1 Threaded Input/Output

Threaded I/O still leverages blocking input/output operations, however, in a way that allows multiple simultaneous connections.

The gist of it is using different threads, performing each blocking operation in a different thread. Then, when an operation blocks, it just blocks the thread that is executing the network connection, while the other threads continue having a runnable state.

In addition, it is quite an efficient method, because most operating system schedulers don't consider IO-blocked threads for scheduling until the operation can be completed successfully. This means that no CPU time would be wasted trying to execute the blocked thread.

However, one of the drawbacks of using a thread-pool for handling I/O operations is the scheduling and memory overhead that each thread incurs.

Each created thread needs to allocate a stack space (around 2MB in Linux). Even though the memory is not committed (used), virtual memory space is consumed – and this can be a problem in 32-bit systems [2].

This paradigm is frequently used in different programming languages and server components. For example, Erlang VM handles all disk I/O operations in a pool of threads, so it does not block the rest of program execution. Apache

HTTP server also uses a multi-thread model to be able to handle multiple concurrent connections.

2.4 Non-Blocking Input/Output

Non-Blocking Input/Output (NIO) changes the standard flow of software: it only allows issuing standard `read()` and `write()` operations when the kernel is certain that there is data to be read or the file descriptor is in a status that supports writing and it would not block.

In Linux, non-blocking I/O is supported for I/O operations in terminals, pipes, FIFOs and sockets. It is not supported on files, because the kernel has an I/O cache that guarantees that it will not block.

To use non-blocking sockets in Linux, the `O_NONBLOCK` flag must be set on the socket's file descriptor status flag when creating the connection. Then, a `read()` or `write()` operation will only succeed if the kernel is certain that it does not need to block to perform I/O. If, instead, the operation should block, the kernel will not perform it[3] and will return a `EAGAIN` or `EWOULDBLOCK`, which actually, are equivalent error codes.

There are three main system calls in Linux that help when using non-blocking Input/Output, by providing a way to do I/O multiplexing: `select`, `poll` and `epoll`.

They do not perform I/O directly, but they are a pillar of efficient non-blocking I/O, because they allow the program to poll for the status of their file descriptors. It replaces the need to retry every I/O operation infinitely when a `EAGAIN` error occurs.

It is worth noting that out of the three system calls, the first two are standard in the POSIX specification. This means that every POSIX-compliant operating system has the said system calls available for developers.

[3] This can happen, mainly, in two instances. In a `read()` call, when the received data buffer is empty and no new data has been received. When doing a `write()`, it could happen if attempting to write an amount of data that would fill the sending buffer.

On the other hand, `epoll` is a Linux-specific system call. It is similar in design to `kqueue`, which is available on BSD-based operating systems, such as FreeBSD or macOS.

It is said that Non-Blocking Input/Output is one of the ways to handle higher concurrency numbers, because no system resources are spent waiting for I/O. Instead of waiting, time is spent processing other data that it is ready.

2.4.1 `select`

The `select()` function arrived with the implementation of Berkeley sockets in BSD4.2 in 1983. `select` is a system call that provides I/O multiplexing facilities to programmers. This system call was available on the 1.0 version of the Linux Kernel.

According to its documentation, `select` "allows a program to monitor multiple file descriptors, waiting until one or more of the file descriptors become "ready" for some class of I/O operation" [3].

It is defined in the following way:

```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *errorfds,
           struct timeval *timeout
);
```

There are three types of states that can be monitored with `select`:

- A file descriptor is readable
- A file descriptor is writeable
- A file descriptor has an exceptional condition: out-of-band data is re-

ceived[4] or "a state change occurs on a pseudoterminal slave connected to a master that is in packet mode" [5]

When `select()` is used in a non-blocking way, the function returns the total number of file descriptors that are ready to be read, written and checked for exceptions. This means that `select()` can be used for busy-waiting on such file descriptors.

`select()` is not limited to handling non-blocking I/O, as it can be configured to block.[5] However, it allows programmers to use the function in a non-blocking manner (passing a value of 0 for the `timeout` argument).

`select` has one big limitation: in a single call, it can just keep track of a fixed number of file descriptors. This number is a constant defined at kernel compile time – it is *not possible*[6] to change this value in runtime. In Linux it is hardcoded to 1024.

2.4.2 `poll`

`poll()` is used in a similar way as `select()`. It is also used to perform I/O multiplexing. The function first appeared in the 2.2 version of the Linux Kernel, released in early 1996.

It is defined in the following way:

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

[4] Out-of-band data is a feature of streaming sockets that allow notification that there is high-priority data without having to read all the data received before[4].

[5] The developer can decide whether the `select()` call blocks or not. If it is configured to return immediately, it will not block, but the program needs to be busy-waiting. This has the advantage of optimising for lower latency at the expense of higher CPU usage. Instead, if the programmer decides to use `select()` in a blocking way, it will be slightly more inefficient (there is an overhead with blocking and resuming threads at the kernel level, due to context switches) but it will not waste CPU time performing the busy-waiting in user-space.

[6] More than impossible, it is designed to be difficult to workaround and change that number. Should it be changed, it would need a re-definition inside `glibc`. However, it is probably better to use `poll` or `epoll` instead of trying to *hack* `select` into accepting more values.

If we compare the definition of `poll` with the `select` one, we can see that it is very similar.

In this case, instead of separating file descriptors into sets based on the operation state we want to handle, just a set of `pollfd` is needed. This data structure contains the file descriptor and the events we want to "acknowledge".

Compared to `select`, `poll` does not limit the number of file descriptors that it can track and is able to track more fine-grained events, including:

- Data can be read[7] (Events: `POLLIN` and `POLLRDNORM`)
- Out-of-band data can be read (Events: `POLLRDBAND` and `POLLPRI`)
- Peer has shutdown their socket (Event: `POLLRDHUP`)
- Data can be written[8] (Events: `POLLOUT` and `POLLWRNORM`)
- Out-of-band data can be written (Event: `POLLWRBAND`)
- An error has occurred (Events: `POLLERR`)
- A hangup[9] has occurred (Event: `POLLHUP`)
- File descriptor is not open (Events: `POLLNVAL`)

`poll()` can also be used in a blocking or non-blocking way. The latter, also allows the programmer to perform busy-waiting for the specified events for the given file descriptors.

2.4.3 `epoll`

`epoll` is a collection of system calls (`epoll_create`, `epoll_ctl`, and `epoll_wait`) appeared in the version 2.6 of the Linux kernel, released in late 2003.

[7] Not including out-of-band data.

[8] Not including out-of-band data.

[9] A hangup is an state where there the file descriptor is valid but "a device has been disconnected, or a pipe or FIFO has been closed by the last process that had it open for writing."[6].

`epoll` provides a more modern foundation that provides I/O multiplexing facilities. It was designed to be more scalable, particularly, in terms of file descriptors to watch.

It has two main differences when compared to the previous two I/O multiplexing methods:

- It provides a more efficient way to track lots of file descriptors
- It provides edge-triggered or level-triggered notifications.

This API introduces two new concepts. File descriptor events can be edge-triggered or level-triggered. An event is edge-triggered when it does not keep status, once the event has been emitted.

Let's look at an example. With level-triggering, even if no new data is received, the file descriptor will be returned as being *readable* until data is fully consumed. Instead, when we set up `epoll` with edge-triggering semantics to monitor that a file descriptor can be read, there is a different behaviour. Once `epoll_wait()` has returned once that the file descriptor can be read, it will not return the same status until new data comes in.

`select` and `poll` are only level-triggered.

Edge-triggering requires more care when programming (and some kind of state machine), as it can produce a *deadlock* when one side misses one event or there is more data that can be written or read with a single system call. Then, one side would be waiting for more data to arrive (instead of sending data back) and the other side would be waiting for data from the other side.

There is a limitation on the maximum number of file descriptors that a single instance of `epoll` can track. This limitation, however, is not static and hard-coded as `select` is. It basically depends on the available system memory, but it is much larger than `select`. There is also a maximum number of `epoll` instances that can be created system-wide.

When it is just used in an level-triggered mode (like `select` or `poll`), the manual describes it as: "`epoll` is simply a faster `poll`, and can be used wherever the latter is used since it shares the same semantics."[7].

We can see all the `epoll`-related system call definitions below:

```
int epoll_create(int size);

int epoll_ctl(
    int epfd,
    int op,
    int fd,
    struct epoll_event *event
);

int epoll_wait(
    int epfd,
    struct epoll_event *events,
    int maxevents,
    int timeout
);
```

When using `epoll` we notice that we first need to use `epoll_create()` to create a file descriptor.

This file descriptor represents the *poller* that will be used to perform further calls. Once it is created, we need to register the file descriptors that we want to track with the file descriptor provided by `epoll_create()`. We register the file descriptor we want to watch using `epoll_ctl()`, passing the events we want to keep track of. Finally, to check if any file descriptor has triggered any event, we use the `epoll_wait()` in a similar way to how `select` and `poll` behave.

`epoll` supports reporting a subset of the events that `poll` report.

- Data can be read[10] (Event: `EPOLLIN`)
- Out-of-band data can be read (Event: `EPOLLPRI`)
- Peer has shutdown their socket (Event: `EPOLLRDHUP`)

[10] Not including out-of-band data.

- Data can be written[11] (Event: `EPOLLOUT`)
- An error has occurred (Events: `EPOLLERR`)
- A hangup has occurred (Event: `EPOLLHUP`)

It is worth noting that when using `epoll_wait()`, we do not need to pass the list of file descriptors to the kernel each time. This is due to the stateful nature of `epoll`. It is the main conceptual difference in usage from the other two.

We can see from the definition above that `epoll` is completely different from `select()` and `poll()`. The others are stateless on the kernel side, where the program needs to pass the list of file descriptors each time it wants to *poll* for new events.

[11] Not including out-of-band data.

3 Benchmarking I/O Multiplexing Implementations

This Bachelor's thesis focuses on comparing the throughput of four different ways of achieving I/O multiplexing: using a threaded approach, using `select`, `poll` and `epoll`. As it has been mentioned previously in section 1.2, we will be only focusing in networking I/O.

3.1 Benchmark Components & Architecture

The tests have two main components:

- A simple server
- A client

The server is the simplest component in the test suite. It is implemented using the Go language. It is a simple non-blocking TCP server that uses language-specific paradigms so it is performant and scalable.

The point of this project is not to test the performance of the server implementation and we want a server that is performant so it does not become a bottleneck on our tests.

The server immediately accepts a connection from the client and replies the same response to every request, after waiting for a specified time.

The client is where the benchmark is performed. It is written in C++ and it contains a benchmark implementation of every method we are testing. It logs information[12] regarding when a request has been made, a response has been

[12] The logging operation has been designed in such a way that imposes the least overhead possible in the application. It uses a circular buffer to temporally store tracing information and it is persisted to disk periodically, asynchronously, using a background thread. This should not have any impact in network I/O. However, there is a small impact on raw throughput, as some CPU cycles are being spent storing and writing tracing information instead of processing requests. It is worth noting, also, that this overhead should be more or less constant with the concurrent connection value, irrespectively of the other variables.

received and the CPU time consumed for each test run. All this data is aggregated and analysed later.

3.1.1 Client

As it has been mentioned above, the client has been implemented in C++, because it allows abstracting the differences in every test while allowing code reuse and because it is easy to access the system calls we are using in the tests without much performance hit. As we can see in fig. 1, an abstraction for the test runner has been designed, and every benchmark has the specific implementation for its multiplexing method.

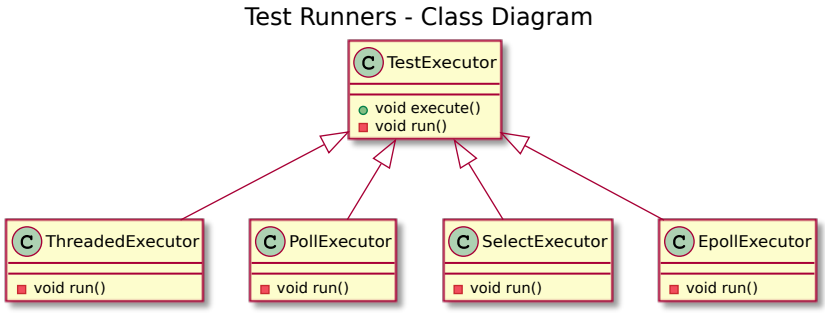


Figure 1: Class Diagram of the implementation of the test runners

In addition, there are numerous operations that are done in each test. For example, for creating a new socket and connecting it to the remote server, sending a request to the server, reading a response back or emitting tracing data.

Each operation has been implemented so that it can be used independently of the test it is running. This means that every test is executing the same code for the operations above. Because every test executed the same shared code, we have results that can be better compared between them. If there were a small inefficiency in an operation, it should have the same effects in all the tests, thus, it would still allow relative comparison between all.

As it has been mentioned previously, a log is created for every test. Each log is stored in a CSV format and contains a record of every socket that has been

opened and closed, every request issued, every response received, every connection failure[13] and used user and kernel CPU time during the test run[14].

The information contained in a trace is aggregated with all the other information for further analysis.

Test runs that use `select`, `poll` and `epoll` run using a single thread and perform busy-polling. `threaded` tests, instead, use blocking I/O and a thread per connection.

3.1.2 Protocol

We defined a simple protocol for our tests. It is request-response text-based protocol where each message ends with the new line delimiter (`\n`).

Each request contains information regarding the test and the expected behaviour of the server. Namely, it contains the number of seconds we want the server to wait until returning the response. In the example below, we can see how a request for a delay=`500ms` would be represented in the wire.

```
500\n
```

The server always returns the same fixed-size response, to all clients and all requests. In our case, it is a 12 byte response with a fixed content:

```
Lorem Ipsum\n
```

[13] There were no connection failures in the tests, however the implementation accounted for it. In our tests, a connection failure is considered when `connect()` fails and when there is an error in `write()` or `read()` operations.

[14] This CPU time information is obtained directly from the kernel using `getrusage(RUSAGE_SELF)`. It is a system call that allows the calling program to get usage information from itself. It supports retrieving information such as CPU times, memory usage, page faults, IPC messages, etc.

3.2 Methodology

We focus on analysing the request/response throughput of every I/O multiplexing method we are testing.

To do so, the tests have been modelled with three different variables: number of concurrent connections, response delay and connection reuse.

We assume the request/response throughput will increase linearly with the number of concurrent connections, until the server cannot keep up with the load, the network is saturated or the underlying I/O implementation is pressured.

A response delay is injected in every test so that we can compare what happens when there is a delay. We will be using `5ms`, `50ms` and `500ms` as delay values.

We also compare what happens with a connection-per-request model[15] versus a persistent connection model[16]. In both scenarios, the client does not send a second request until the response from the previous request has been received.

In the connection reuse scenario, a test will look like:

1. A socket is created and a connection opened
2. A request is sent
3. A response is received
4. Repeat 1-3 until test is finished
5. Close socket

Instead, when not reusing connections, the test will look similarly to:

[15] This reproduces the HTTP/1.0 connection model. Each time a request is sent, a connection to the server is opened. After the response is received, the connection is closed.

[16] This reproduces HTTP/1.1 and newer versions (also used in multiple protocols). A connection is opened for sending a request and is not closed until the session finished. This means that the same connection is reused for multiple requests which results in increased throughput and lower latency [8]. It is usually known as persistent connections or *keep alive*.

1. A socket is created and a connection opened
2. A request is sent
3. A response is received
4. The socket in 1 is closed
5. Repeat 1-4 until test is finished

Each test run takes a minute to complete. Tests have been run multiple times on a 3 vCPU[17] servers[18] using an internal network between the machine that hosted the server and the one that hosted the test.

3.3 Results

More than 11GB of tracing information has been recorded for the analysis in section 3.3 across more than 1.700 test runs for this Bachelor's thesis.

Results of the analysis will be presented in three sections:

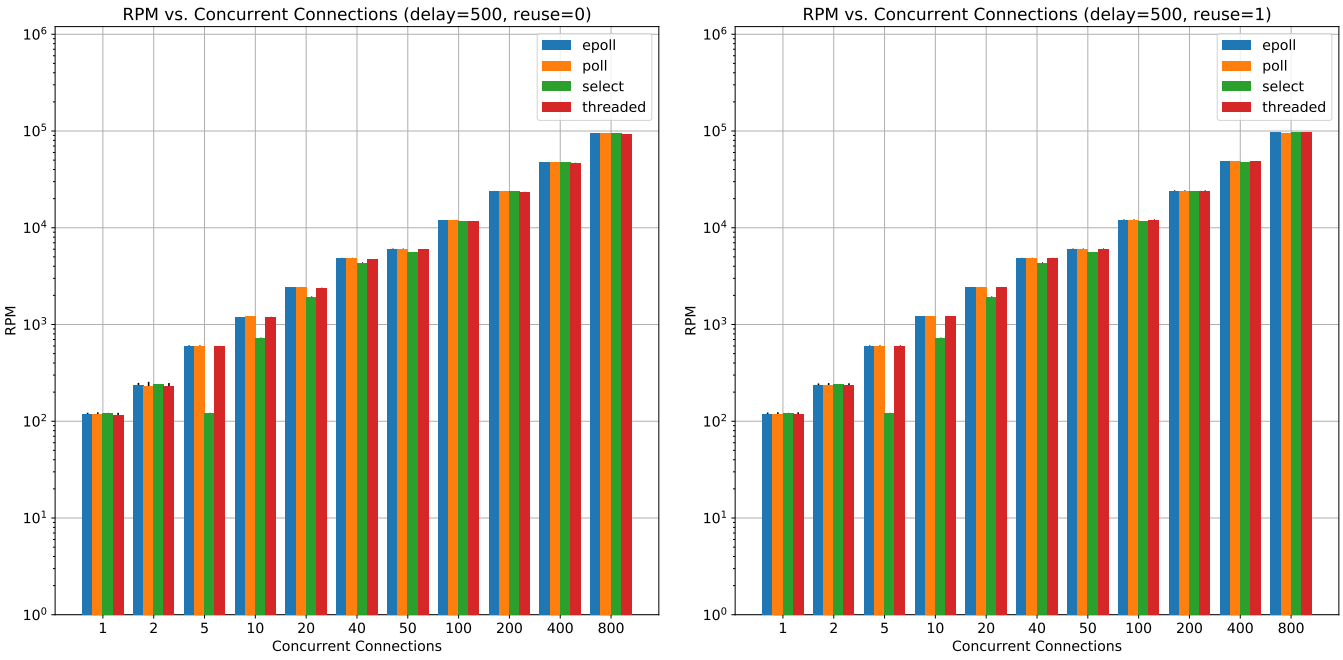
- **Throughput**: Analysing the impact each method has in throughput. We measure it through the total amount of successful Requests per Minute (RPM).
- **Connection Reuse**: We will be analysing the difference between reusing connections and using a new connection for each request in terms of throughput
- **CPU Usage**: Analysing the CPU usage in seconds of each method.

[17] vCPU is a share of the underlying physical CPU from cloud servers. Usually, a vCPU is a core or a thread of a core.

[18] Servers were hosted in DigitalOcean.

3.3.1 Throughput

We will be analysing throughput results separately by delay (`5ms`, `50ms` and `500ms`).



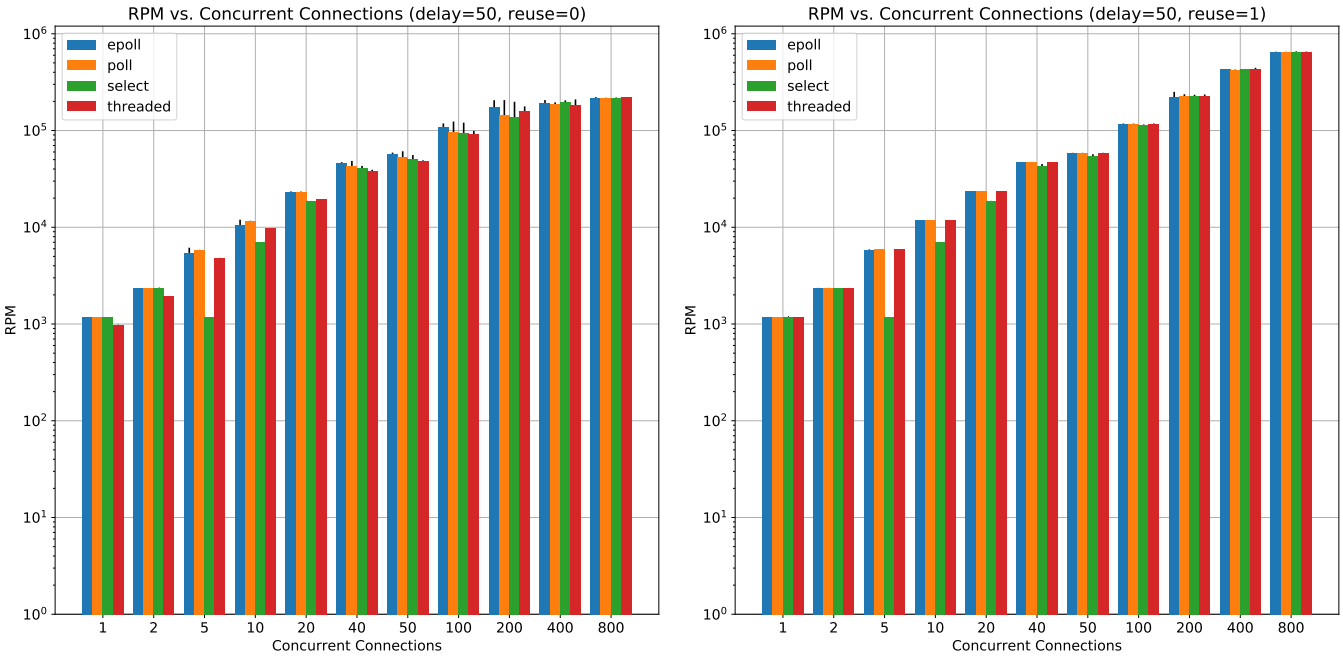
(a) RPM Throughput of tests with delay=`500ms` and no connection reuse

(b) RPM Throughput of tests with delay=`500ms` and connection reuse

Figure 2: Comparing RPM throughput observed with delay=`500ms`, with logarithmic scale for RPM

When we look at the request throughput at 500ms delay in fig. 2, we can see that RPM numbers are increasing linearly. And each method does so – the R^2 value for the regression line of every method is almost 1.

It is interesting to notice here that all methods perform similarly – both when not reusing connections and also when reusing them. This has an easy explanation: the delay is so large that small inefficiencies do not show.



(a) RPM Throughput of tests with delay=50ms and no connection reuse

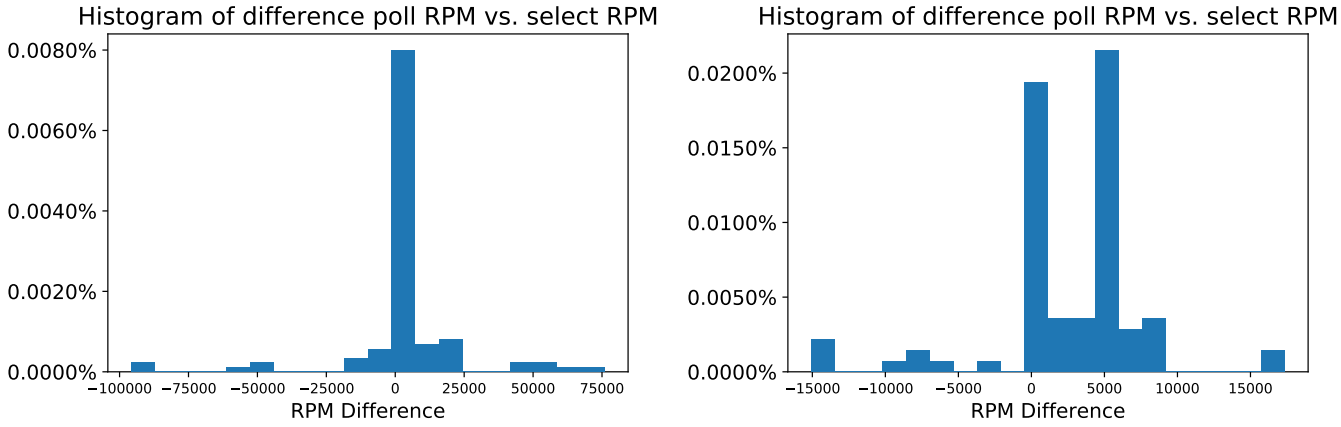
(b) RPM Throughput of tests with delay=50ms and connection reuse

Figure 3: Comparing RPM throughput observed with delay=50ms, with logarithmic scale for RPM

When looking at delay=50ms data in fig. 3, we start seeing some differences between each method both when using connection reuse and not.

We start seeing more absolute throughput with connection reuse compared to when not reusing connections. We will analyse in detail this effect in section 3.3.2.

As we can see in fig. 4, it is interesting to notice how frequently `poll` offers more throughput than `select`, despite not showing a significant throughput advantage.



(a) Histogram of distribution between `poll` and `select`.

(b) Histogram of distribution between `poll` and `select`, zooming on (-20000, 20000).

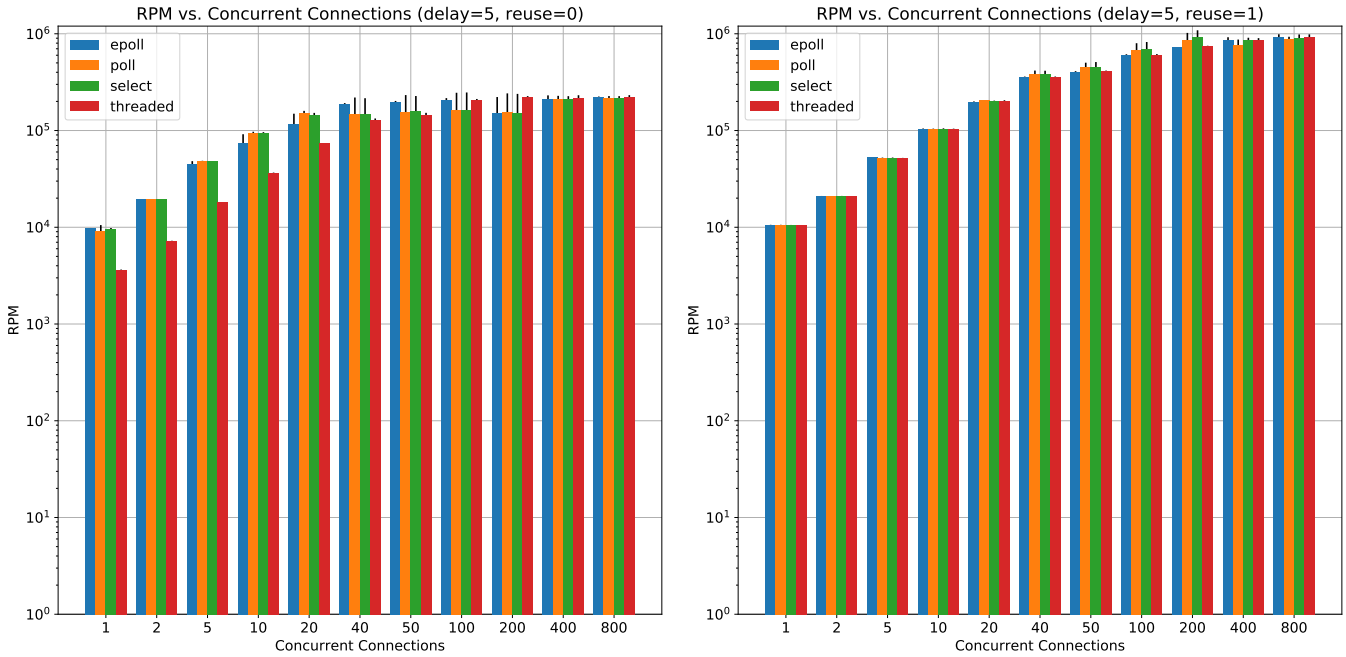
Figure 4: Histogram of distribution of the difference between `poll` RPM and `select` RPM in all runs with delay=`50ms`. A negative value indicates that `poll` was slower than `select`. A positive one, that `poll` was faster.

We can see that in almost every benchmark, the `poll` implementation had more throughput than `select`.

It is worth noting that `select` and `poll` have a similar implementation in the Linux kernel [5]. That explains why there is not a significant variance of the throughput results.

Historically, `poll` had an advantage to `select` on sparse sets of file descriptors – when there were few file descriptors monitored but some of them had big file descriptor numbers. In Linux 2.6 (late 2003), there were some optimisations introduced to close that gap.

Benchmarks run with delay=`5ms` should be the ones that start showing different performance characteristics between I/O multiplexing methods, as seen in fig. 5. This can be explained by the fact that there is going to be a bigger amount of requests (there is less latency) and this increase in number of requests translates into an increase in the number of I/O operations.



(a) RPM Throughput of tests with delay=5ms and no connection reuse

(b) RPM Throughput of tests with delay=5ms and connection reuse

Figure 5: Comparing RPM throughput observed with delay=5ms, with logarithmic scale for RPM.

First of all, it is not surprising that in a non-pipelining request model[19], less delay is correlated with a higher throughput, that was the expected behaviour and the one seen in the tests.

We can see that behaviour when using a low amount of concurrent connections is quite similar to what we have seen in the other two experiments.

However, it is noticeable a different behaviour for the `threaded` when there is a delay of 5ms. An explanation for that could probably lie in the fact that there is much more thread contention. This was not such an issue in the previous situations because the delay was greater and threads were able to stay longer

[19] A pipelining request model is one where the client continuously issues requests, without first waiting to the server to send its response.

in a *waiting* state.

We can start noticing larger deviations between `poll` and `epoll`[20] once we use a bigger amount of concurrent connections. We will see the difference in performance looking at `10.000` and `20.000` concurrent connections.

Unfortunately, it is not possible to test `select()` with more than 1024 connections (as it has been described in section 2.4.1). Also, it is worth noting that the `threaded` approach could not be tested with such amount of concurrent connections because it needs a thread per each connection and thread creation is limited by the kernel.

It is there where `epoll` starts showing an edge, as we can see in fig. 6.

[20]We cannot compare them to select, because as it has been described in the theoretical section 2.4.1, it is limited to tracking 1024 file descriptors – so in this case we are limited to tracking 1024 concurrent connections.

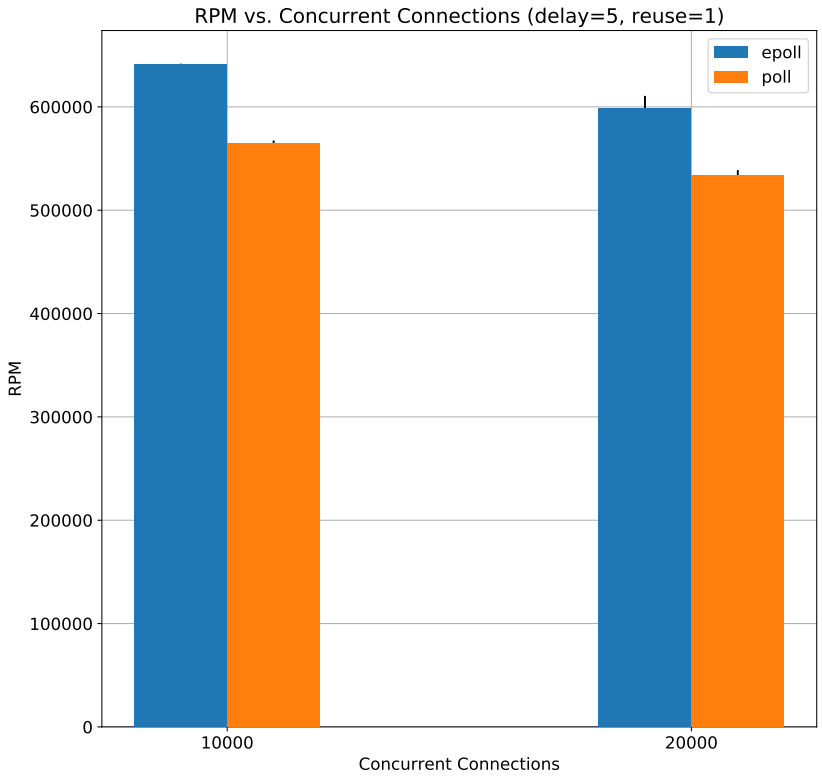


Figure 6: RPM comparison of `epoll` and `poll` when tracking a high number of connections and reusing connections with a test duration of 120s.

Even if we are just comparing relative performance between `epoll` and `poll`, it is interesting to note the behaviour of the test, here 20.000 connections achieve worser throughput than 10.000 connections. This can be explained by many factors. One of them is that we are using tests with a limited time, and there is an associated overhead of opening the connections – this means that it is a fixed cost for the benchmark, thus increasing the test run time should offer *better* throughput.

In addition, it is worth mentioning that being able to reliably send the maximum possible amount of packets over thousands network connection from a single host is not an easy feat: there are limits around the maximum number of connections per destination IP and destination port; there is an overhead

by using TCP to send and receive packets (both in terms of added space from headers and increase in packets throughput from sending `ACK`s). For those reasons it is difficult to focus in absolute throughput with this test setup.

It is also important to note that when not using a connection reuse model and tracking a large number of connections, the comparison between `epoll` and `poll` can be a mixed bag. This is explained by the fact that we need to add and remove each socket that we are using to `epoll`, and that requires 2 different system calls per each socket. At a high number of concurrent connections, this additional overhead in multiple `syscall`s starts to show.

3.3.2 Connection Reuse

After analysing throughput data from both tests with connection reuse and tests that use create a new connection for every request, we can see in fig. 7 that using connection reuse can offer greater throughput across every tested delay point.

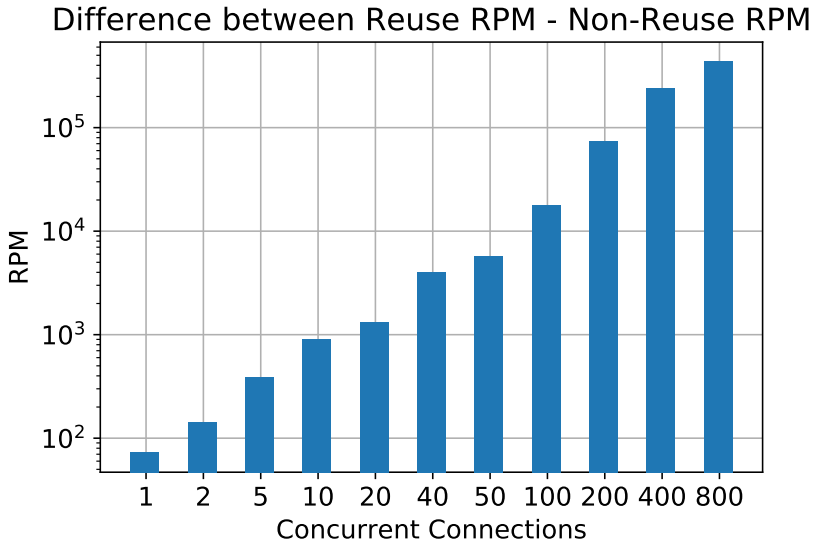


Figure 7: Graph showing throughput difference between tests that reuse connections from those that do not, across all tests performed, with logarithmic scale for RPM.

As we can see, in this case that the increase in throughput is observed in every I/O multiplexing method tested.

This phenomenon can be explained by the fact that there is an associated overhead in the creation of a socket – not just in the creation of the associated file descriptor (this is quite inexpensive), but, every time a new socket connection is opened the kernel has to perform the 3-way TCP handshake[21] [10].

Also, there is another source for potential delays (and, thus, throughput decreases). The server may process and accept the connection request with some delay. When doing the operation potentially hundreds of thousands times, even if it takes a small time, delay adds up significantly[22].

3.3.3 CPU usage

To analyse CPU usage of the I/O multiplexing methods, we will be focusing in *total CPU time* and a bit on *kernel CPU time*.

User CPU time is the CPU time that CPU cores have been executing the executable code. *Kernel CPU time* is the CPU time that CPU cores have been executing Kernel code associated with system calls required by our process [11]. We define *total CPU time* as *User CPU time* + *Kernel CPU time*.

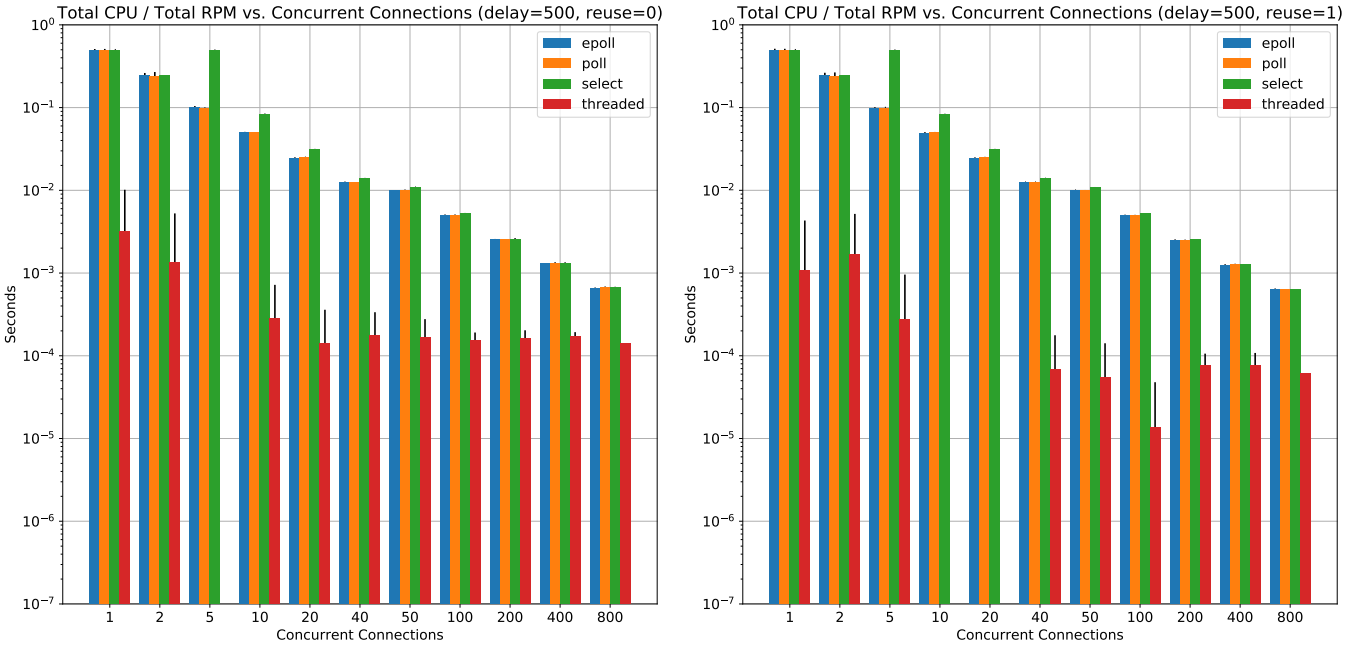
It must be taken into account that `select`, `poll` and `epoll` perform busy polling, so that it is a bit complicated to compare them to the `threaded` method (especially when delay figures are higher)[23]. They are doing busy polling because it optimises for lower latency and thus, it provides an increase in throughput.

[21] TCP 3-way handshake is such an overhead that there is a proposal of new protocol, QUIC[9], in draft status, that offers similar features as TCP, implemented in user-space on top of UDP, that are able to achieve 1-way handshake for establishing connections or, even, without handshake.

[22] Take for example, an `accept` operation that takes `0.1ms`, in some test runs, there were on the order of 10^5 connections, so that translates into a delay of 10s. That is 10% of the runtime of a single benchmark.

[23] It is worth noting that `select`, `poll` and `epoll` can also be used using a *blocking* approach, where the busy polling is performed by the Kernel rather than on user space.

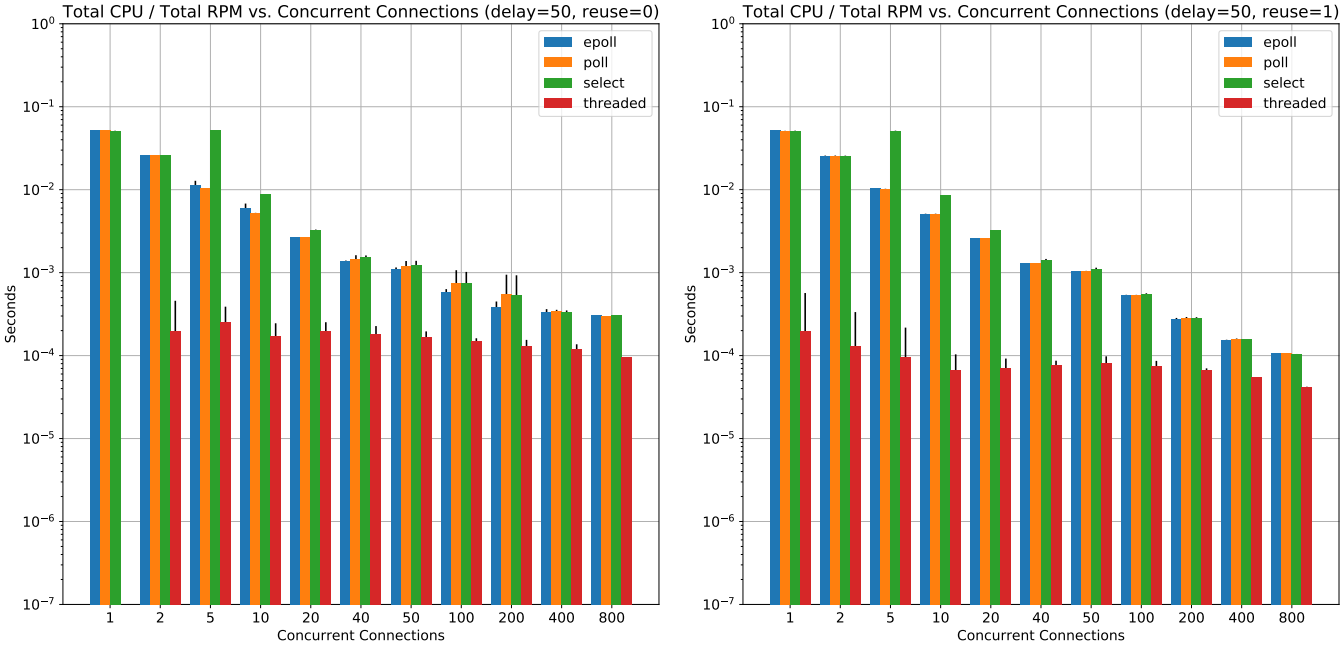
Unfortunately, the `threaded` approach can only work in a blocking way.



(a) Total used CPU seconds per request with delay=500ms and no connection reuse

(b) Total used CPU seconds per request with delay=500ms and connection reuse

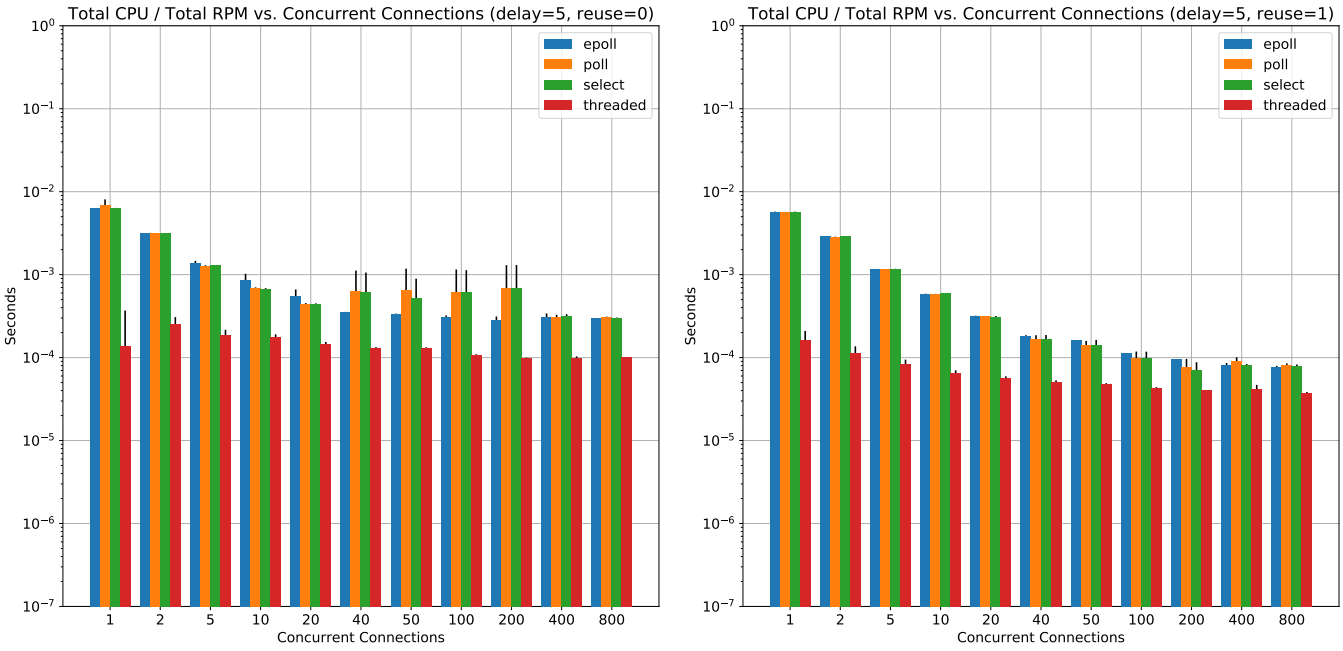
Figure 8: Comparing total used CPU seconds per request observed with delay=500ms, with logarithmic scale for seconds



(a) Total used CPU seconds per request with delay=50ms and no connection reuse

(b) Total used CPU seconds per request with delay=50ms and connection reuse

Figure 9: Comparing total used CPU seconds per request observed with delay=50ms, with logarithmic scale for seconds



(a) Total used CPU seconds per request with delay=5ms and no connection reuse

(b) Total used CPU seconds per request with delay=5ms and connection reuse

Figure 10: Comparing total used CPU seconds per request observed with delay=5ms, with logarithmic scale for seconds

Results are presented in fig. 8, fig. 9 and fig. 10. Firstly, we can notice that frequently `select` is the method that uses most total CPU time per request performed, especially with higher delays.

`threaded` is a special case, because, as we mentioned previously, operations block, so no CPU time is spent on waiting. This is the reason why it has the lowest CPU usage per request.

Test without connection reusing do use more CPU time per request, compared to those that do employ connection reuse. This is largely explained by the added overhead, for similar reasons as the ones stated in section 3.3.2.

When testing at high concurrent connections, with connection reuse, it seems like `epoll` is consistently more efficient than `poll` and `select`. This makes

sense because as we described previously, there should be less overhead when calling `epoll_wait` compared to the other two system calls, as we do not need to pass around the list of file descriptors and events we want to be notified of each time.

`epoll` has an expected CPU usage curve, especially when we also look at the kernel CPU time used – as we can see in fig. 11. `epoll` kernel CPU usage stays constant until around 100 concurrent connections. This makes sense because there is the penalty of having to create the `epoll` file descriptor – that is a fixed cost, and does not depend on the number of file descriptors will be tracking. After that, just n system calls have to bee issued, one per connection.

The interesting behaviour is after 100 concurrent connections. Kernel CPU time starts to rise, and seems to grow linearly. This could be explained by one of the following reasons:

- `epoll_wait` is slower when is tracking more file descriptors
- There is a non-negligible amount of added overhead when creating more sockets or doing more concurrent operations

It is worth noticing, however, that `select`, `poll` and `epoll` tend to have similar CPU performance characteristics and it does not seem like we can extract more interesting conclusions. A more in-depth analysis (probably dedicated kernel profilers) should be done to be able to determine if this behaviour is explained by one of the reasons above, or a combination of them.

When looking at `threaded`, also in fig. 11, we can see that the lower the delay, the higher the CPU usage per request. It is mostly due to the additional thread context switching overhead. It has a different curve to the rest of the methods because kernel time increases with the number of concurrent connections, as there is more context switching between threads.

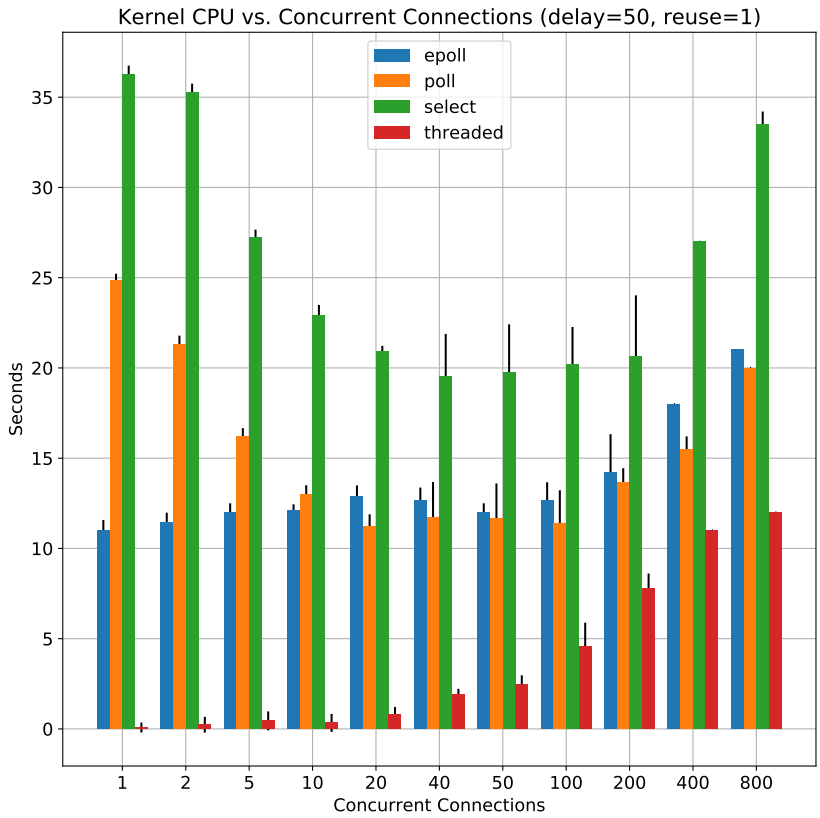


Figure 11: Comparing total kernel CPU time observed with delay=50ms and connection reuse.

3.3.4 Method Analysis

After analysing the data, we can extract the following conclusions regarding the methods, as guidelines to use when selecting which multiplexing facility we want to use:

- Threaded I/O multiplexing can be useful with a small-to-medium number of concurrent connections. The higher limit remains limited by CPU usage, OS limitations and context switches. It is easy to reason about because it uses the same paradigms.

- `select` and `poll` are quite interchangeable. First one is only useful for a small number of concurrent connections (1024) and the latter one can scale to higher numbers. They are also useful because they are portable – they work on every POSIX-compliant operating system.
- If portability is not a concern, or a large number of concurrent connections need to be tracked, `epoll` is the most efficient solution for tracking multiple connections.
- Code used to handle I/O multiplexing with `select`, `poll` and `epoll` (at least level-triggered) is quite similar – and follows the same structure, so it should be easy to migrate between all of them to overcome each issues.
- If greater efficiency when handling a large amount of concurrency is necessary, it is recommended to design the program in a way that is possible to abstract away the use of different multiplexing facilities, this should help, for example, to use `epoll` in Linux-based systems and `kqueue` in BSD-based ones.

4 Conclusions

The main objectives of this Bachelor's thesis were understanding I/O multiplexing facilities, its differences and similarities and which benefits Non-Blocking I/O provides. Also, another objective was developing a benchmarking suite for all the methods mentioned in the report. All the objectives set for this project have been achieved.

Across the tests, we have been able to see the advantages of reusing connections, as it highly increases the throughput and reduces the relative CPU usage per request. This is something to take into account when designing protocols and networking systems.

4.1 Future Work

There are some interesting tests and lines of work that were not pursued in this Bachelor's project due to size and time constraints, but that could be pursued.

- Comparing the same tests with another two variables: request payload size and response payload size, to see if and how affect the results.
- Comparing the same tests with mixing highly active connections with some quiet ones
- Comparing the same tests in BSD systems to compare `epoll` to `kqueue` [12], which are similar in nature and first one was inspired by the former one.
- Test Asynchronous Input/Output (AIO) system facilities and compare them with the ones tested in this project. Coincidentally, in the later stages of this project, a new kernel API (`IOCB_CMD_POLL`) was released that allows polling for events using the AIO subsystem. This allows for greater performance, because it can map a circular buffer of the event in the process address space, which highly reduces the number of required system calls – and thus the number of context switches.

- Test AIO/NIO implementations abstracted by 3rd-party libraries, such as `libevent`, `libuv` or `Boost.Asio`.
- Compare the tradeoffs between performance and program complexity, maintainability and developer productivity that each method offers. Not every facility is easy to reason about, some of them even have hidden complexity. It is important to understand the implications of each method not only in performance but also if it makes developers more productive or less, if cause more or less bugs, etc.
- In the recent years, there has been a lot of research in the area of networking completely done in user-space (thus bypassing the kernel entirely), such as MegaPipe [13]. It would be worth comparing the throughput and performance characteristics between in-kernel networking and off-kernel networking.

References

[1] “Project Loom: Fibers and Continuations.” [Online]. Available: https://wiki.openjdk.java.net/display/loom/Main

[2] “Measuring Context Switching And Memory Overheads For Linux Threads,” Sep 2018. [Online]. Available: https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

[3] *select(2) Linux User’s Manual*.

[4] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. San Francisco, CA, USA: No Starch Press, 2010, ch. 61.13.1 Out-of-Band Data, p. 1283.

[5] ——, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, 1st ed. San Francisco, CA, USA: No Starch Press, 2010, ch. 63. Alternative I/O Models, pp. 1325–1374.

[6] *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) - Redline: IEEE Standard for Information Technology – Portable Operating System Interface (POSIX®) Base Specifications*. IEEE, 2018, no. n.º 7. [Online]. Available: https://books.google.es/books?id=PKH-vQEACAAJ

[7] *epoll(7) Linux User’s Manual*.

[8] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud’hommeaux, H. W. Lie, and C. Lilley, “Network Performance Effects Of HTTP/1.1, CSS1, and PNG,” in *ACM SIGCOMM Computer Communication Review*, vol. 27, no. 4. ACM, 1997, pp. 155–166.

[9] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-17, Dec. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-17

[10] “Transmission Control Protocol,” RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[11] “Mastering Linux performance: CPU time and CPU usage,” Mar 2018. [Online]. Available: https://jaroslawr.com/articles/mastering-linux-performance-cpu-time-and-cpu-usage/

[12] J. Lemon, “Kqueue-A Generic and Scalable Event Notification Facility.”

[13] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, “MegaPipe: A New Programming Interface for Scalable Network I/O,” in *OSDI*, vol. 12, 2012, pp. 135–148.

5 Annex

5.1 Running the test

Source code for both the server and the client has been provided with this Bachelor's thesis. In this section we will describe how to set up and run a test.

Both components of the test include a `Dockerfile` so that they can be run easily without needing any kind of dependency (other than Docker itself). However, it is worth noting that using this setup for the tests will probably incur in some added overhead from Docker itself.

5.1.1 Running the server

To run any test, we first need the server up and running.

It can be executed easily by just typing:

```
$ go run server/server.go
```

It is worth noting, that the server will be listening for connections on port TCP 7777, so it will need to be free before running this program.

5.1.2 Running a test

The test client is prepared to run either all the tests or a subset of tests. It allows configuration of delay times and number of concurrent connections through command line flags.

We can see the full list of command line flags below:

- `-ip`: The IP address the client will use in benchmarks
- `-port`: The port the client will use in benchmarks

- `-testWaitSeconds`: The cool-down period in seconds between each test
- `-delays`: The comma-separated list of delays in milliseconds that will be tested
- `-connections`: The comma-separated list of connections that will be tested
- `-threaded`: Run the `threaded` test
- `-select`: Run the `select` test
- `-poll`: Run the `poll` test
- `-epoll`: Run the `epoll` test
- `-all`: Run all tests

An example execution of a test that will all tests with 5 and 10 concurrent connections, with 50ms of delay to localhost would be invoked using:

```
$ ./code -connections 5,10 -delays 50 -ip 127.0.0.1
```

The trace information will be stored in `/tmp`.