CSE 511 Project 1 Report

Quinn Burke, Ata Fatahi, Sushrut Shringarputale

October 31, 2018

Contents

1 Introduction

This project serves as a introduction to the different ways the Linux Operating Systems allows asynchronous Input Output (IO) and the difference between using one design choice over another. In this project, we develop a client-server implementation with three different types of servers and have the client connect to each kind with very specific workloads. We compare the results of using synchronous IO against asynchronous IO, as well as experiment with two different modes of asynchronous request handling.

This project was evaluated on the Linux machines located in W204 of the Westgate building. In order to build the project, follow the instructions here or in the README.md file.

2 Building and Running

To build the project simply run `make`. This will build the server as well as all the clients that we tested with the server.

To run the server, use

```
1 ./server <option>
```

where option is one of `1, 2, or 3`.

1. Runs the server with blocking IO on sockets and files
2. Runs the server with asynchronous IO using polling on sockets and blocking IO on helper threads for files
3. Runs the server with asynchronous IO using event notification (signals) and blocking IO on helper threads for files.

To run a client, just run one of the generated client binaries, although in case the clients are not on the same system as the server, the IP address of the server may have to be manually changed in the code for that specific client.

3 Design and Implementation

The server is implemented in three ways, differentiated based on how it listens on the network socket for connections and data. In the evaluation section, we will compare these three methods of performing synchronous vs. asynchronous IO and present our findings from this project.

3.1 Multi-Threading (MT)

In this section, we implement a classic server model where the main "thread" listens for connections and worker threads actually perform the necessary communication.

The main thread blocks on a socket for incoming connection requests. Once a connection has started, it accepts the connection and passes along the resulting socket file descriptor to a helper thread, which handles the incoming data and writes out the data to the socket as needed. Once it has finished executing, the thread cleans up and exits.

Every helper thread has to service some form of request (GET, PUT, INSERT, DELETE). These requests can be served directly from the Least Recently Used (LRU) cache if possible. If the data is not serviceable from the cache, the thread issues a blocking IO request to the disk "database", which is the source of truth to serve any data.

3.2 Event-Driven with Polling (EP)

In this section, we have to implement an event driven system to serve the requests coming from the client. Therefore, we have a single main thread which uses asynchronous IO to server the requests. For each client connection, a socket is created and the main thread calls epoll() system call to check the events on it and serve the request.

3.3 Event-Driven with Signaling (ES)

This part involves a single main thread (with an associated listening socket) that waits for data, which can be associated with either new connections, existing connections, and be either incoming or outgoing data. On receipt of data, a signal is sent to the main thread notifying it to take some action on the data. The signal handler then raises a flag and pushes some data to an event pipe, that can be read by the main thread (outside of the signal handler). The main thread periodically checks the flag to see if there are any events needing attention, reads data from the pipe (event type and associated socket), and takes necessary action. The main thread services all events in the order received from clients. It does this outside of signal handlers because some needed functions are not async-signal-safe and cannot be used within signal handlers.

The main thread then does checks (valid request, cache checks, etc.) and issues I/O requests accordingly. Meanwhile, the helper threads are constantly polling the global task queue for I/O requests to handle (in a blocking manner). After completing an I/O request, the helper thread signals the main thread to forward the correct return data back to the client (this outgoing signal handler behaves similarly as above). The key components are the global task queue (that the main thread adds tasks to and the helper threads pull from), the use of a flag to give a cue to the main thread of pending events to be read on our event pipe, and the locks used to maintain synchronization between the main thread and helper threads.

3.4 Database and Cache

The database is simply a text file (`names.txt`) that the helper threads can search through for data as needed. Each entry in the database is represented as a `(key, value)` pair, and each entry is separated by a newline character. Once data has been found in the database, or in the case of an INSERT added to the database, it is also added to the cache to serve requests with temporal locality. Each time a thread connects to the database, it acquires a lock on the database which forces mutual exclusion between the threads and enforces integrity of the database and cache.

The cache is an in-memory LRU cache implemented as a doubly linked list. Entries may be inserted into, moved within, or deleted from the cache on each request sent to the server.

4 Evaluation

In this section, we explain the evaluation we did for all 3 types of server architectures and compare the results.

4.1 Experimental Setup

Infrastructure. We first started using department lab machines, but unfortunately, the connection between two servers was so slow specially for part 2 and part 3. Since we didn't have enough time, we decided to run the experiments on a faster infrastructure to save time. Therefore, we used AWS virtual machines for our experiments. The VMs we used are two *m5d.large* that each has 2 vCPUs and 8 GB of memory and running Ubuntu Server 16.04 LTS.
Workload. The workload we use, is determined by several knobs such as the ratio of GET/PUT requests and the KEY-VALUE sizes. The GET/PUT ratio varies is either 0.1, 0.5 or 0.9. The arrival rate is following a uniform distribution with the average 50 requests per seconds with a concurrency level of 25. The Key-Value pair size also is either 100B, 1KB or 8KB. The popularity distribution for keys follows a Zipf distribution with 1.345675 which results in having a 90:10 popularity fraction. That is, 90 percent of the workload is targeting 10 percent of keys.
Methodology. For each experiment, we have a set of 1000 unique keys which their popularity is described above. As the workloads we use consist of GET and PUT requests, we first INSERT all the keys into the database. This is two-folds, first, it fills the cache and hence brings the system into a steady state, and it prevents from getting errors and misses upon PUT and GET requests respectively. Since 90 percent of the whole workload is targeting 10 percent of keys (100 keys), and the cache hit rate should be 90%, we set the cache size to be 100 resulting a 90% hit ratio. Also, for part 2 and 3, the size of thread pool is set to 50.

4.2 Multi-Threading (MT) Evaluation

In this section, we present the results for MT architecture for three different size including 100B, 1KB, and 8KB and different GET/PUT ratio including 0.1, 0.5 and 0.9.

4.2.1 Key-Value Size: 100B

The results are as folow:

Table 1: Blocking Sockets with Multi Threading, KV Size: 100 Bytes

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	4065	6.12	6.14	6.86	6.97
0.5	6849	3.49	3.46	4.7	5.9
0.9	7092	0.54	0.29	0.81	1.09

The request arrival rate is as follow:

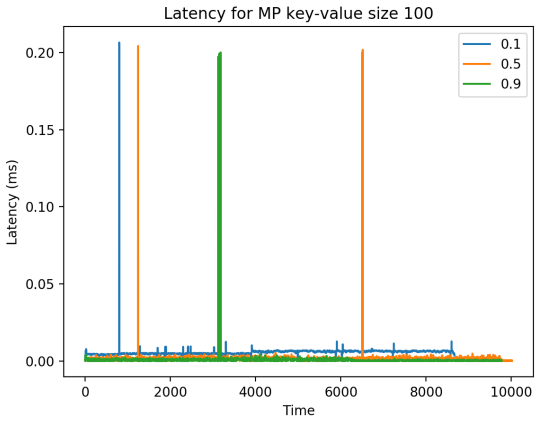

Figure 1: Response time for different GET/PUT ratios on MT server for 100B key-val

4.2.2 Key-Value Size: 1KB

For size 1KB:

Table 2: Blocking Sockets with Multi Threading, KV Size: 1 KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	1190	20.8	21.1	22.9	23.16
0.5	2057	12.04	12.04	15.7	17.6
0.9	7874	3.23	3.07	5.7	7.1

The request arrival rate is as follow:

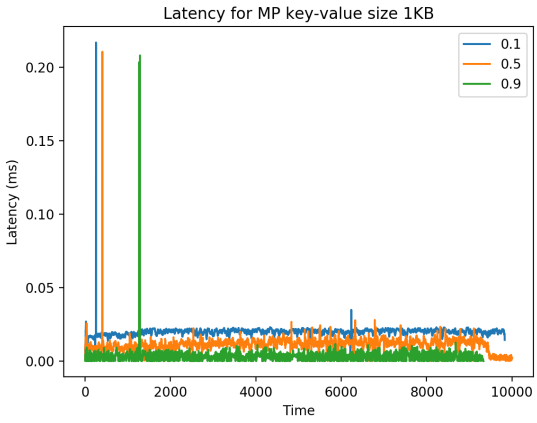

Figure 2: Response time for different GET/PUT ratios on MT server for 1KB key-val

4.2.3 Key-Value Size: 8KB

For size 8KB:

Table 3: Blocking Sockets with Multi Threading, KV Size: 10 KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	111	225	227	249	258
0.5	188	131	133	178	256
0.9	770	131	133	177	256

The request arrival rate is as follow:

Figure 3: Response time for different GET/PUT ratios on MT server for 8KB key-val

As the results presented in tables show, the maximum throughput for MT architecture is 7874 request per second and it's achieved when the key-value size is set 1KB and the GET/PUT ratio is 0.9. In this case, the median response time is however, 3.07ms, and the 99th response time is 7.1 ms.

4.3 Event Polling (EP) Evaluation

4.3.1 Key-Value Size 100B

The result:

Table 4: Async Sockets with Edge-Polling, KV Size: 100 Bytes

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	7.47	0.43	0.33	1.2	1.9
0.5	10.3	0.37	0.24	1.2	2.07
0.9	17	0.21	0.07	0.85	1.65

The request arrival rate is as follow:

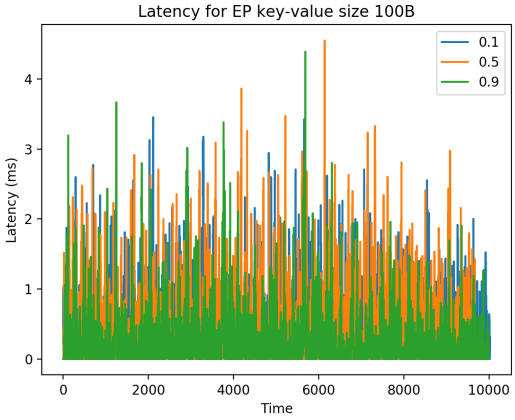

Figure 4: Response time for different GET/PUT ratios on ES for 100B key-val

4.3.2 Key-Value Size 1KB

The result:

Table 5: Async Sockets with Edge-Polling, KV Size: 1KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	8.25	0.42	0.33	1.1	1.68
0.5	9.8	0.33	0.23	1.02	1.64
0.9	18	0.20	0.06	0.83	1.54

The request arrival rate is as follow:

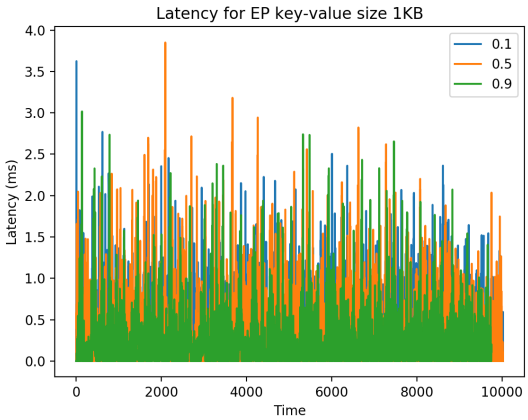

Figure 5: Response time for different GET/PUT ratios on ES for 1KB key-val

4.3.3 Key-Value Size 8KB

The result:

The request arrival rate is as follow:

Table 6: Async Sockets with Edge-Polling, KV Size: 8KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	7.8	0.47	0.38	1.25	1.90
0.5	10.4	0.35	0.25	1.07	1.87
0.9	18.6	0.22	0.09	0.84	1.54

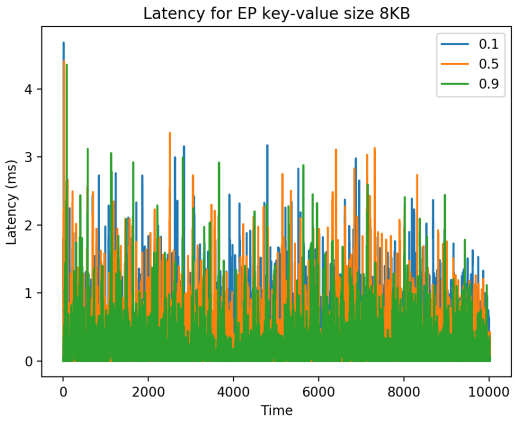

Figure 6: Response time for different GET/PUT ratios on ES for 8kB key-val

As the results presented in tables show, the maximum throughput for MT architecture is 18.6 and it's achieved when the key-value size is set 8KB and the GET/PUT ratio is 0.9. In this case, the median response time is however, 0.22ms, and the 99th response time is 0.84 ms.

4.4 Event Signalling (ES) Evaluation

4.4.1 Key-Value Size 100B

The result:

Table 7: Async Sockets with Edge-Polling, KV Size: 100 Bytes

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	9.2	2.57	2.4	5.009	5.8
0.5	10.8	2.27	2.11	4.71	5.76
0.9	23.1	1.04	0.92	2.67	3.53

The request arrival rate is as follow:

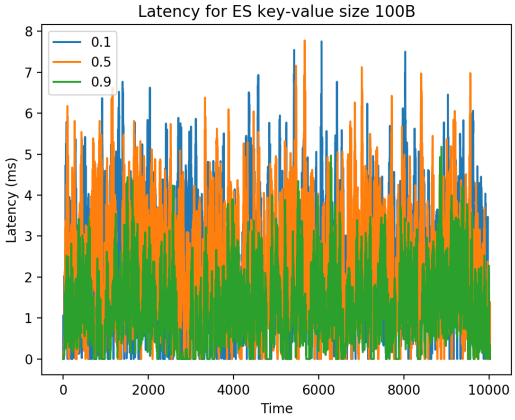

Figure 7: Response time for different GET/PUT ratios on ES for 100B key-val

4.4.2 Key-Value Size 1KB

The result:

Table 8: Async Sockets with Signals, KV Size: 1 KB

Ratio\Metric	Throughput (req/sec)	avg. rt (ms)	med. rt (ms)	95th rt (ms)	99th rt (ms)
0.1	8.3	1.07	0.93	2.64	3.54
0.5	10.9	2.09	1.94	4.44	5.56
0.9	26.3	1.05	0.91	2.6	3.6

The request arrival rate is as follow:

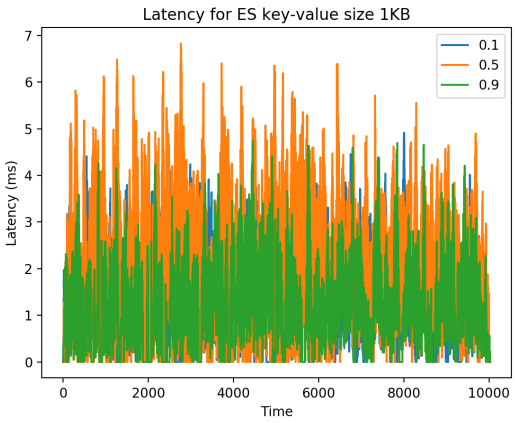

Figure 8: Response time for different GET/PUT ratios on ES for 1KB key-val

As the results presented in tables show, the maximum throughput for MT architecture is 26.3 and it's achieved when the key-value size is set 1KB and the GET/PUT ratio is 0.9. In this case, the median response time is however, 0.91ms, and the 99th response time is 3.6 ms.

5 Conclusion

We demonstrate three different ways the Linux Operating System allows asynchronous I/O and evaluate the relative performance across each method for a variety of workloads. We conclude that the server built with blocking and spawning new threads for each request has significantly higher performance across the three workloads. This significant difference is visible because the mechanism there spawns just as many threads are needed, and they don't perform a lot of concurrent operations. Thus, there is not much waiting needed on mutex locks when the thread performs its operations. On the other hand, the overhead needed from waiting on mutexes between all portions (accept connection, read request data, process request data, perform IO, and respond) require some overhead to ensure synchronization and no preemption. As the amount of concurrency required goes up, with clever synchronization, the async sockets would perform much better than the threads. It would make it more apparent if the disk IO was also served by asynchronous IO.