Optimizing Single Server Content Streaming

Nevin Zheng
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 12213
nlz@andrew.cmu.edu

Carlos Gonzalez
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 12213
cggonzal@andrew.cmu.edu

ABSTRACT

In recent years, a multitude of content streaming services which allow distributed users to consume their favorite songs, movies, and other media have emerged. Since the introduction of Netflix and YouTube, the amount of content delivered via the internet has increased every year. Users prioritize low latency, high quality, and a lack of waiting for content to download or buffer. While large companies may employ complex distributed architectures to efficiently stream their content, this paper seeks to compare common single-server architectural patterns in order to determine which patterns are most suitable for simultaneously streaming audio and video. We evaluate the performance of our servers by measuring the relationship between client-side latency distribution and request rate. Our goal is to compare commonly used server architectures in order to find the best performing server for streaming. We demonstrate that certain server architectures lead to better performance and should be preferred in content streaming applications.

1. INTRODUCTION

The popularity of streamed content has caused companies such as Netflix, Amazon, Apple, and others to invest heavily into streaming technologies. On the server side, A large majority of technologies that the typical consumer will interact with daily are built upon HTTP and the traditional client-server interaction model.

Streaming content generally comes in either audio, video, or a combination of the two. Many advances have been made in file encoding that can improve the user experience. For example, file compression can reduce a file's size without reducing perceived quality while allowing forward error correction to tolerate the loss of transmitted data. In this paper, we focus primarily on server architectures and technologies which have enabled the proliferation of streaming content that requires both audio and video such as movies.

In this paper we compare several popular server streaming architectures based on event driven and multithreaded architectures. Our goal is to find an optimal server architecture from among the most popular server architectures and frameworks such as Node.js, Deno, Rust + Tokio, and a multithreaded threadpool based architecture. We evaluate performance based on how quickly a server is able to process a fixed number of requests per second which we varied between 100 requests per second to 1000 requests per second. Surprisingly, our multithreaded server outperforms all other architectures in both the 100 req/s and 1000 req/s scenarios. The multithreaded server is able to outperform all other implementations by several orders of magnitude in the 99.9% of latency during the 1000 req/s test benchmark. We discuss these results in section 5.5 and propose further studies to be done in section 5.6 which allow for a more rigorous comparison between our servers in a real world situation.

2. BACKGROUND

A simple media content server can be created by using the HTML audio and video tags and having a server directly serve the requested file. But then the following questions arise:

1. How do we handle the buffering that is done so the client does not have to wait until most of the file is downloaded in order to begin playing the file?
2. What protocols exist so that our server can provide the client with only the relevant parts of the audio or video content that need to be buffered?
3. How do we make sure that our client-side code is portable across browsers?
4. What structure does our content need to have on the server side so that we can maximize throughput?
5. Which server architectures should be used in order to minimize the client side latency of streamed content and allow services to scale to serve millions of concurrent users?
6. How do we adapt to clients with slower internet connections?
7. What are the common streaming architectures?

8. How do we efficiently send data to the client?

Question 5 is the focus of this paper and is answered thoroughly in sections 3 - 7. Questions 1, 2, 4 and 6 are answered in section 2.1. Question 3 is answered in section 2.2. Question 7 is answered in section 2.4 and question 8 is answered in section 2.3.

2.1 Streaming Frameworks

In response to the lack of a standardized protocol to stream content, large companies have created several different protocols to answer these questions. YouTube and Netflix have relied on Dynamic Adaptive Streaming over HTTP (DASH) in order to scale to meet their users needs. Apple instead uses HTTP Live Streaming (HLS) for both their audio and video services and Microsoft has relied on Microsoft Smooth Streaming [1].

These frameworks all rely on having the same content on the server at different bit-rates (i.e. high quality, medium quality, and low quality) and partitioning the content into small chunks called *segments* that can be delivered to the client as needed. This allows the server to adapt to a client's connection by serving the highest quality possible, one segment at a time, without the client getting stuck "pause buffering." For example, if a client is watching a movie at a high quality but their internet suddenly becomes slower, the server can switch to serving segments that are of a lower bit rate so that the client can continue watching the movie without having to pause to buffer. This method of dynamically changing a stream's quality to meet the client's needs is referred to as *Adaptive Bit-rate Streaming*. See figure 1 for an example directory structure. See [2] and [1] for more details.

Each protocol has a slightly different way of describing the structure of its content on the server side but most use some form of a *manifest* to describe how their data is partitioned. Even though all of these protocols have differences in how they handle the structuring of their data on the server side and how they dynamically adapt bit rates based on a client's connection, they all solve the client-side buffering problem in a similar but slightly different way. The lack of a standardized framework to handle buffering on the client side led the W3C to propose the Media Source Extension in 2013 which extends the audio and video HTML tags and provides a common ground for handling buffering on the client side regardless of the server side specifics.

2.2 Media Source Extension

The standard HTML video and audio tags do not facilitate the buffering of media. The Media Source Extension [3] was proposed by the W3C in order to facilitate the management of buffering video and audio streams by forcing browsers to support the MediaSource and SourceBuffer Javascript objects. The MediaSource object represents a source of media data (audio or video) that can be attached to a video or audio tag in order to provide a source of content. Additionally, the MediaSource object has a SourceBuffer property that handles fetching the data from the server and placing it into a buffer that the MediaSource object uses to provide the content to the user. These objects provide a standardized way for developers to write portable media-driven code across browsers. See figure 2 and the consumer sections below for further details.

```
└── movie
    ├── high-quality
    |   ├── segment1.mp4
    |   ├── segment2.mp4
    |   └── segment3.mp4
    ├── low-quality
    |   ├── segment1.mp4
    |   ├── segment2.mp4
    |   └── segment3.mp4
    └── medium-quality
        ├── segment1.mp4
        ├── segment2.mp4
        └── segment3.mp4
```

Figure 1: An example of the structure of a directory holding segmented data. Here the movie is offered in high, medium, and low quality each of which is split up into 3 segments. If a client's connection degrades, the next segment that is sent to the client can be taken from a bit rate with a lower quality. In practice a movie usually consists of segments of about 10 seconds which causes it to be split into hundreds of segments.

2.3 Codecs

The format in which data is stored on the server plays a key role in how efficiently it can be distributed. For example, a minute of uncompressed 1920 x 1080 video recorded at 30 frames per second would require 14.93 gigabytes of storage, not including audio. As a result, both audio and video are coded and decoded using standardized methods called *codecs* which allow for efficient coding and decoding of data.

Content is fed into the *coder* method of a codec and the output is then stored inside a *container* such as MP4, WebM, and Quicktime (MOV) which is served to the client. When a client needs to decode the data that is inside a container it is given, the codec is used to decode the data and present it to the user. We want to emphasize that a codec's job is to compress and decompress data while the container's is to simply hold data that has been compressed by a codec. When the data

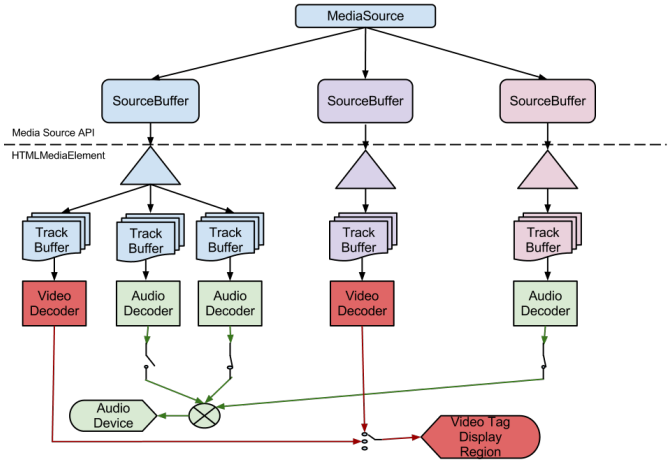

Figure 2: The Media Source Extension API as described in [3]. A single MediaSource object, shown at the top of the figure here, is attached to a video HTML tag as described in the consumer sections later in the paper. Three SourceBuffers are created and added to the MediaSource object. The leftmost SourceBuffer handles buffering and decoding both audio and video data. The middle SourceBuffer buffers and decodes video data. The rightmost SourceBuffer handles buffering and decoding audio data. When the browser requires more data to display to the user, it looks in the SourceBuffers for data, decodes it, then sends it to the audio device if it is audio data or displays it to the user in the video tag display region if it is video. The client side Javascript called, described in the consumer sections later in this paper, handles the queries made to the backend to make sure the buffers are always full.

stored inside a container must be uncompressed to play the audio or video, it is decompressed by the codec and presented to the user. The difference between containers and codecs is a common source of confusion. See figure 3 for a visualization.

Audio codecs allow for efficient compression and decompression of audio data. The most common audio codecs are MP3, AAC, Opus, and Vorbis. In our tests we use AAC as it is commonly used for audio content that will be stored alongside video such as when storing movies. See [4] for further details. Similar to audio codecs, video codecs allow for efficient compression and decompression of video data. The most common codecs are AV1, AVC (H.264), and VP9. In our tests we use AVC (H.264). See [5] for further details.

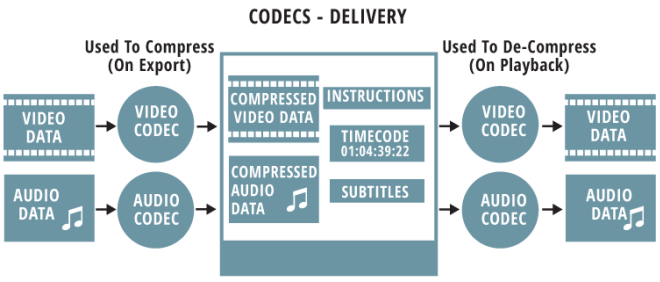

Figure 3: Example of codec pipeline as shown in [6]. The raw video and audio data shown on the left is compressed using their respective codec and the compressed output is stored in the container, here shown as the large square in the middle of the figure. This container is what is served to the user. When the data needs to be played in the browser, it is again passed through the codec in order to decompress it before being played to the user.

2.4 Common Server Architectures

Most streaming service architectures in use today are either multi-threaded with a thread pool or event driven [3] [1] [2]. A multi-threaded server handles requests by creating a shared buffer and a thread pool [7]. The main thread's job is to enter the server loop and place the file descriptor of accepted connections into the shared buffer. The threads in the thread pool will each take a connection request from the buffer and handle it. If the buffer is empty, the threads will block until a connection is placed into the buffer by the main thread. The process that threads in the thread pool use to handle a request is shown in figure 4.

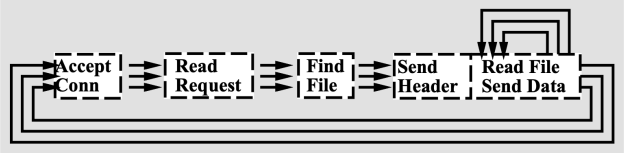

Figure 4: Example of multi-threaded server design as shown in [8]. Each thread will go through the steps shown here when handling a request. Note that our multithreaded architecture described in section 4.3 uses a thread pool which is not shown here.

Event driven architectures use the *select*, *poll*, or *epoll* system calls on linux, with *epoll* being the preferred modern syscall, *kqueue* on macOS, and *IO Completion*

Ports on Windows systems. Each of these facilities allow applications to asynchronously wait for a change to occur in a relevant set of file descriptors. When a change is detected, the request is handled by the server. This process is shown in figure 5.

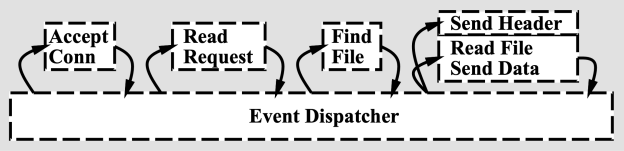

Figure 5: Example of event driven server design as shown in [8]. The event dispatcher will use *epoll*, *poll*, or *select* in order to wait for a change in one of the file descriptors that are in a specified set. When a change is detected, the request is handled by the server.

2.5 Summary

Although the Media Source Extension is still a draft, it is widely supported across browsers and provides a common starting point for implementing the client side portion of a streaming service. The server side part of a service will typically use a multi-threaded or event driven architecture that implements a protocol such as MPEG-DASH to serve segmented data. Additionally, a thorough understanding of codecs and containers is needed to understand streaming due to their ability to efficiently compress and decompress data. The subtleties involved with audio/video codecs and containers were our largest source of frustration when developing our tests.

3. HIGH LEVEL DESIGN

Our architecture consists of a distributor and a consumer. The distributor is a server that follows the event driven pattern shown in figure 5. The consumer is our client side HTML and JS files that handle creating a MediaSource object, making the relevant content requests to the server, and pushing the segmented content into the SourceBuffers. The distributor and consumer follow the standard client-server request and response pattern. While content distribution can be done in many ways, this simplified model captures the fundamentals of streaming and excludes complex optimizations.

3.1 Distributor

Our distributor is a static file server which supports GET requests for content and the Partial-Content HTTP header. Application level in-memory caching of content is not considered and all content is read from disk. The distributor can be used in two modes. Firstly, the client can make repeated partial ranges requests for portions of the same file. Secondly, it can serve a file tree in the form of Figure 1. However, there is no support for a manifest containing metadata on the segments, which indicates that the client must be aware of the files it is requesting. Adaptive streaming based on quality of service metrics are not considered. For our experiment, we configured our load testing tool to request manually segmented portions of a video file.

3.2 Consumer

Our consumer consists of an HTML file with a video tag and an external Javascript file that does the heavy lifting. We rely on the MediaSource and SourceBuffer objects provided by the Media Source Extension [3] in order to provide buffering capabilities for audio and video with the MP4 file's relevant codecs for each. A single SourceBuffer is used to hold both audio and video in order to eliminate making subsequent calls to our server for the same MP4 file.

To provide thorough testing and assess the throughput performance impact of using the Media Source Extension, we created two separate consumers. The first consumer we tested only made asynchronous calls to our distributor server without pushing the fetched results into the relevant SourceBuffers. The second consumer architecture we tested would make the asynchronous calls and also push the results into the SourceBuffer. We did not see a measurable performance impact when pushing into the buffer so all of our testing was performed using the second approach since it is how streaming services are implemented in practice.

3.3 Summary

We have designed and implemented a simplified model of content streaming using a static file server, called distributor, which supports both the Partial-Content HTTP header as well as serving file trees in the style of Figure 1. This distributor is queried by a front end consumer which requests the relevant segments of an MP4 file and plays them to the user. Production content servers can support additional features and functionalities such as adaptive-streaming based on quality-of-service metrics, changing language-tracks, subtitles, and etc. However, in essence, all forms of content streaming are variations on GET requests for Partial-Content.

4. IMPLEMENTATION

4.1 Distributor

Our distributor implementation is built with Rust, Tokio, and Hyper. Rust is a multi-paradigm programming language designed for performance, concurrency, and memory safety [9]. Tokio is an asynchronous run

time for Rust [10]. Hyper is a fast and correct HTTP implementation built on top of Tokio [11]. Using this technology stack allows developers to quickly implement and deploy highly performant internet services. The implementation follows the template given in figure 5. All of the servers we implemented act as static file servers that utilize the Partial Content header. File accesses are direct to disk with no intermediate caching layer.

4.2 Consumer

A MediaSource object can be created and attached to a video tag with id "movie" and have a SourceBuffer initialized with the following code:

```
let movieTag = document.getElementById("movie
    ");

let myMediaSource = new MediaSource();

movieTag.src = URL.createObjectURL(
    myMediaSource);

const movieSourceBuffer = myMediaSource.
    addSourceBuffer('video/mp4; codecs="mp4a
    .40.2,avc1.64002a"');
```

Once this is done, the consumer makes repeated AJAX calls to the server via the fetch API using the "Content-Range" header to request only the needed parts of the MP4 file. The server responds to each AJAX call with the relevant segments of an MP4 file which are then pushed into the SourceBuffer. In order to help the server cope with load, the consumer only makes requests when there is less than 15 seconds left of buffered content on the client's browser.

4.3 Multithreaded Server

Our multi-threaded server uses a thread pool to handle incoming requests. It is written in C and uses pthreads. The main thread accepts incoming client requests and places them into a shared buffer. A fixed number of worker threads are spawned when the server starts and block until there are requests in the shared buffer. Once a request is placed into the shared buffer, the worker threads handle the requests and send a response back to the specified client. Then the thread will look again into the shared buffer for the next request to be processed and block if it is empty.

4.4 Node.js

Next we use Node.js [13]. Node is a popular Javascript runtime that allows Javascript to be run outside the browser and is built on top of libuv[14]. Node uses a single threaded event loop to dispatch and service asynchronous requests. It uses a thread pool for tasks that are either computationally expensive, or those that can-

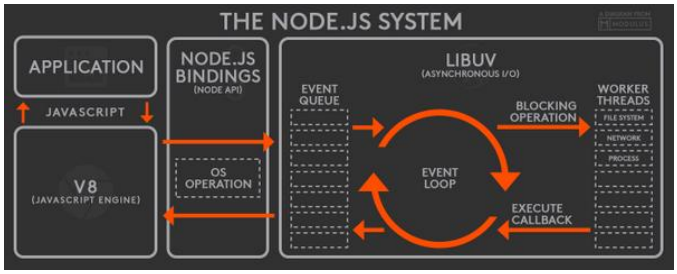

Figure 6: Node.js' architecture [12]. Applications are written in Javascript which are interpreted and serviced by the V8 engine. Node.js extends V8 by providing bindings for libuv. The libuv event loop is single threaded. Some heavier weight tasks will be run in a thread pool outside the event loop. We could not find the original source of this diagram, however it aligns with our knowledge of Node.js' internals.

not be done asynchronously, such as file I/O.

4.5 Deno

Lastly we use Deno [15]. Deno is very similar to Node.js, except that it it written in Rust and Tokio. In the functional block diagrams 6 and 7. You can imagine swapping libuv for the Tokio runtime, Node and Deno both require use of a scheduler for asynchronous tasks. From a developer standpoint, Rust and Tokio is a safer and easier to work with than C and libuv. From the Application developer standpoint, Node and Deno are very similar. Both are interpreted languages and suffer overhead in translating between JavaScript and native code. The benefits of either lie primarily in productivity gained from using JavaScript, a simpler language to learn than C or Rust.

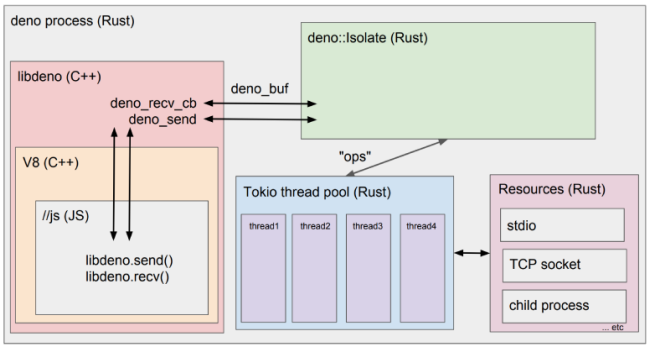

Figure 7: Deno Architecture Diagram [16]. Like Node Deno is also built on top of the V8 JS engine with library bindings to Rust and tokio.

4.6 Summary

Our distributor is a simple content streaming server server using Rust, Tokio, and Hyper. The consumer uses the MediaSource and SourceBuffer Javascript objects to query the server for the relevant content required by the client. We are using a simple HTTP GET protocol that is extended with the HTTP Partial-Content header to support more immediate playback and buffering. Our multi-threaded server spawns a fixed number of threads when the server is started. Then, every incoming request has its file descriptor added to a shared buffer that is handled by the worker threads. In addition we implemented static file servers using Node.js and Deno, which are commercially available Javascript server side runtimes.

5. EVALUATION

We evaluate and compare our multithreaded server, distributor, Node.js, and Deno servers with benchmarks that vary the requests per second from 100 req/s to 1000 req/s. Our goal was to find an optimal server architecture for streaming content that requires both audio and video. We find that the C multithreaded server significantly outperforms all other implementations, in some cases by several orders of magnitude.

For all of our tests we use our Consumer as the client side implementation that creates the MediaSource object, makes the queries for content, and pushes the content into the SourceBuffers.

5.1 Tools Used

For our load testing tool we selected Vegeta [17]. Vegeta is a versatile HTTP load testing tool written in Golang which applies a constant request rate to HTTP services. It supports a rich set of HTTP options as well as a report and plotting tool.

We are utilizing HDRHistograms [18] to visualize our tail latency distributions. HDRHistograms visualize the proportion of requests made that are served within a certain time.

5.2 Environment

For the sake of simplicity, we are running our experiments on a single machine that runs both clients and the server communicating over localhost. Our setup consists of a quad core Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz running Ubuntu 18.04, rustc 1.51, node 14.6.1, Deno 1.9.2, and Vegeta 12.8.4. Future iterations of the experiment could run clients and servers on different machines, more closely emulating a real world service.

5.3 Experiment

We are interested in examining how an increase in request rates impacts the latencies of the server in the 99% and higher tail latencies as shown in figures 9, 10, 11, and 12.

On our test machine, we spin up our server with a directory of content to serve. We then launch Vegeta with a varying set of parameters and content ranges. Vegeta allows you to specify a set of HTTP requests to query and use. This feature allows us to request different portions of different files concurrently, which is what happens on a real server.

5.4 Results

Our results are shown in figures 9, 10, 11, and 12. Figure 9 shows the results of all of the servers when put under a load of 100 requests per second. Similarly, figure 10 shows the results of the servers when placed under a load of 1000 requests per second. These figures show that the Node server outperforms the deno server, especially as load increases. A surprising result comes from the performance gap between the multi-threaded server that uses a thread pool and the other servers. Due to this large gap in performance, we included figures 11 and 12 that only show the multi-threaded server's performance when under a load of 100 and 1000 requests per second, respectively.

5.5 Discussion

We initially expected our Rust server and C multi threaded server to perform similarly, and for Deno to outperform Node.js, with Rust and C outperforming Deno and node. Instead, Under a load of 100 request/second we observed Rust under-performing all the other servers, and the C server outperforming with Deno and Node performing comparably. In the 1000 request/second test we observed Deno in last place, outperformed by Rust, with Node in second, with the C server still outperforming. Results are presented in Figures 8 and 9.

Our results surprised us. We believe that some combination of the following issues may have contributed to the performance gaps.

Firstly, Each server architecture has different amounts of overhead. The Rust/Tokio/Hyper stack provides support for building production-ready, fully-featured web applications, and the tokio scheduler introduces another possible source of overhead. Node and Deno are both scripting languages that introduce the additional overhead of interpreting JavaScript and calling into the asynchronous runtime. In Node.JS' case JavaScript has to call into the node API to perform OS operations. In Deno's case, it has to exchange messages with the deno rust runtime. In contrast, the C multithreaded server is bare-bones, having only the capabilities to send the requested files without performing security checks or caching. Additional work could be conducted to quantify and find sources of overhead in each architecture.

Secondly, The design goals of each architecture are different. Tokio, Node.JS, and Deno are all designed to efficiently handle many thousands of concurrent clients connecting over networks without Quality of Service guarantees. For example, one developer reported using Tokio to concurrently download 100,000 files [19]. Another developer achieved over a million clients concurrent connection in Node.JS [20]. As Deno is a newer project, we were not able to an example of Deno being deployed in a high concurrency application or experiment, but we expect that it would perform well. Although experimentation is required, we expect that the C multi-threaded server will not be able to handle such a high number of concurrent clients since the fixed size shared buffer will become inundated with requests and cause the clients to timeout as requests are dropped. Our experiment applied relatively low constant request rates to each server and plotted the latency distributions. It could be the overhead that supports high concurrency for Rust, Deno, and Node leads to poorer performance in our evaluation, but favors our C multi-threaded server. Additional work could evaluate server performance when the number of concurrent clients is varied. Having this information could give us the full picture into the relationship between request rate, number of concurrent clients, and latency.

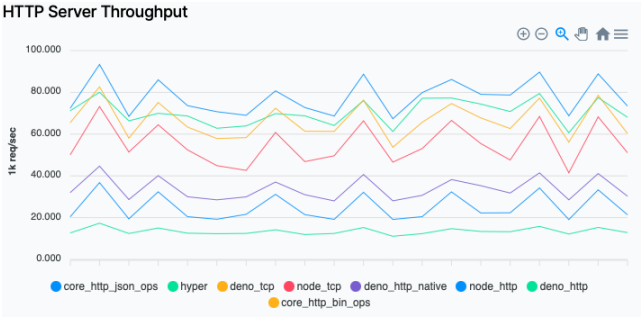

Figure 8: Result of an HTTP Server Throughput benchmark conducted by the Deno development [21]. Ten keep alive connections perform requests serially, and as quickly as possible.

Thirdly, We believe that our choice of 100 and 1000 requests/second may be too low to sufficiently stress the multi-threaded C server. We are using benchmarks performed by the Deno development team as a point of reference [21]. In the benchmark we are highlighting, 10 Keep Alive connections make requests as quickly as possible in order to estimate the servers maximum throughput. Figure 8, presents the throughputs achieved by servers implemented in Deno, Node, and Hyper. We note that the throughputs for Deno, Node, and Hyper in this experiment are approximately 10,000, 20,000, and 80,000 requests per second respectively, nearly two orders of magnitude higher than the our highest request rate 1000 requests per second. Although the benchmark they conduct is a different scenario, it does suggest that we are not placing enough pressure on our servers in order to make a fair comparison. We expect the C server to either have higher latencies at higher request rates or cause client timeouts at high latencies.

Fourthly, The single server setup likely introduced contention between our load tester and server process, which may have affected latency. However, the Deno benchmarks above likely run on a single machine as well. Regardless, future iterations of the experiment could go fully distributed in order to both increase request rate and reducing contention.

5.6 Future Work

In future work, we aim to design and conduct additional experiments to explore each of the discussion points and hypotheses mentioned in 5.5.

We first hypothesize that overhead is one reason for the performance gap between Rust and C. In order to explore this we would need to profile each application or framework to characterize the sources of overhead in each server across a request-response life cycle. While the performance profiles could not be directly compared, it could reveal the relative costs of overhead such as: Tokio Scheduling, JavaScript Interpretation, or JS to asynchronous run-time communication costs.

We believe that our second, third, and fourth proposal all have the same solution: Shifting to a more distributed testing environment that can handle more concurrent clients, and higher request rates. Our second proposal is to examine the effect of concurrent clients on request latency. A distributed setup along with some way to simulate slow and intermittent connections would give us insight into how each architecture scales with the number of concurrent clients, a primary motivator for the design of Tokio, Node.JS, and Deno which our current evaluation methodology does not consider.

Our third proposal is examine the scalability of our servers as the request rate increases. A distributed setup would enable much higher request rates while eliminating resource contention stemming from a single machine setup, our fourth proposal.

5.7 Summary

We compared the distributor server with event driven servers written in Node and Deno – two event driven server architectures used by popular streaming applications. We also compared our distributor with an architecture that uses a multi-threaded approach with a thread pool, which was found to significantly outperform all others. Further studies should be done to investigate the drastic performance differences found between event driven and multi-threaded architectures in the case of content streaming.

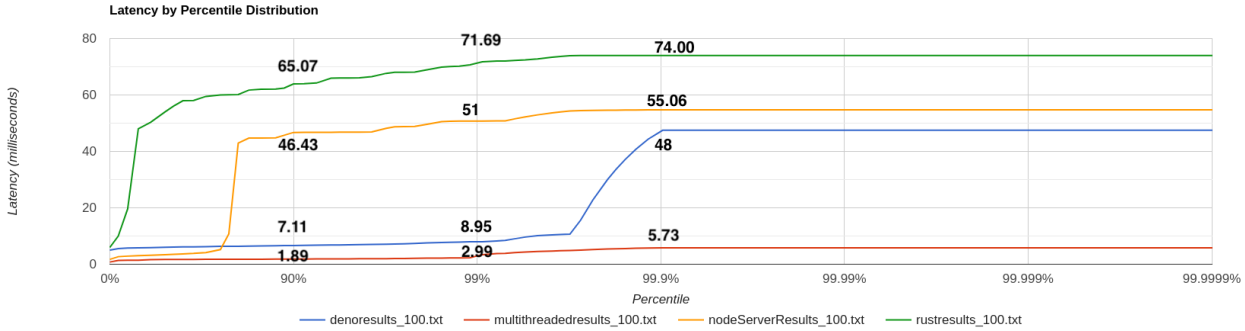

Figure 9: Performance of servers under load of 100 requests per second for 5 seconds.

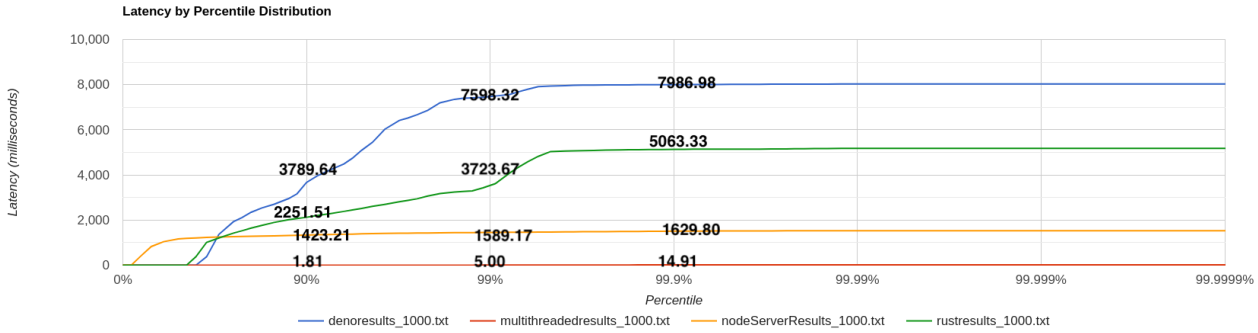

Figure 10: Performance of servers under load of 1000 requests per second for 5 seconds.

6. RELATED WORK

[22] surveys the performance of the different Adaptive Bit Rate (ABR) algorithms that are used on sites such as Twitch and YouTube. Many of the tested sites are known to use MPEG-DASH and thus provide robust and efficient ABR. While this paper does not focus on ABR, the Media Source Extension is flexible enough to adhere to any ABR since it does not impose restrictions on how the browser fetches data. Thus, our testing framework can be used to evaluate all of these sites.

[23] highlights the effects that rateless codes could have on single server streaming to users with diverse codec support. Our architecture only supports browsers that can interpret the AVC (H.264) and AAC codecs, but the increase in performance from rateless codes is worth investigating for services that require serving millions of concurrent users.

[24] investigates the needs of a content delivery video-on-demand service that chooses to use a peer to peer architecture that sits behind a proxy server. When the proxy server needs to retrieve a chunk of video that it does not have cached, it queries one of the peer to peer nodes which communicate to find the needed chunk. Even though this peer to peer to approach gives benefits such as less memory required per node, we suspect that the complexities that it will cause for a streaming service at a large scale will outweigh the benefits since it becomes difficult to make predictions about the consistency of the system. [25] also studies video on demand services that use a peer to peer architecture similar to those in [24] and outlines the problems of scaling these services. We believe that our proposed architecture can be more easily scaled across worker nodes using cloud computing resources such as those provided by AWS.

The event-driven architecture approach shown in [8] is similar to ours. Further speed ups may be achieved by focusing on how *accept* is being used as discussed in [26] and using techniques mentioned in [27] and [28].

7. CONCLUSION

This paper presented a novel server architecture for streaming video and audio content. We compare the performance of the architecture with 3 popular architectures in use today that use either multi-threaded or event driven architectures. We compared the performance of the servers when placed under varying loads and presented the results. We found that our results did not align with our initial expectations. Namely, our Rust server significantly under-performed the C Multi-threaded server, in some cases by several orders of mag-

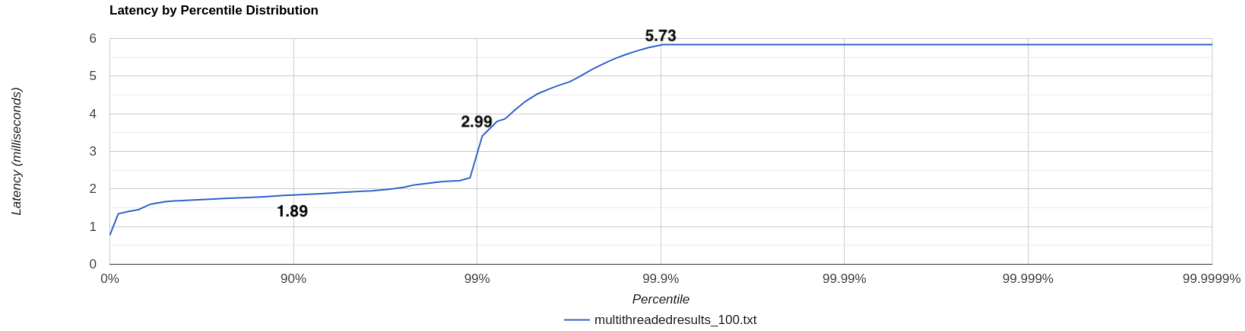

Figure 11: Multithreaded server performance under a load of 100 requests per second for 5 seconds.

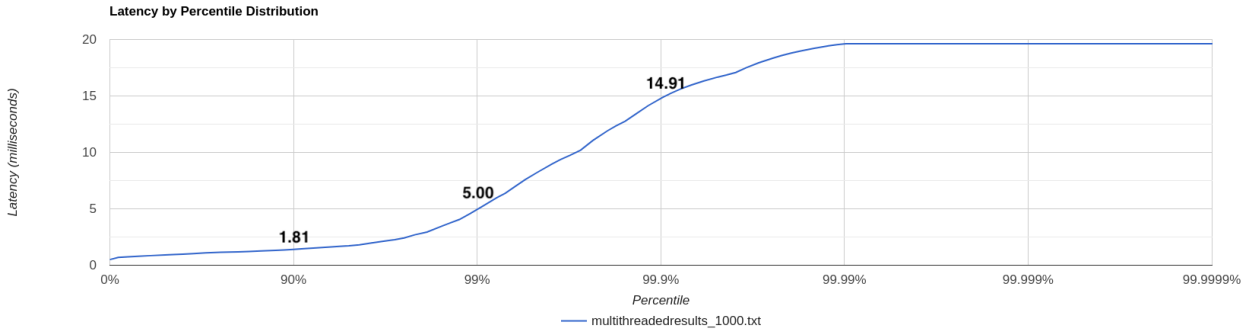

Figure 12: Multithreaded server performance under a load of 1000 requests per second for 5 seconds.

nitude. Further study is required to resolve this difference in performance. Our current data is not sufficient to make any definitive statements regarding any performance advantages or disadvantages of any of the servers we tested. We proposed further studies in order to more rigorously study the topic.

References

[1] Christopher Mueller. *MPEG-DASH vs. Apple HLS vs. Microsoft Smooth Streaming vs. Adobe HDS*. 2015. URL: `https://bitmovin.com/mpeg-dash-vs-apple-hls-vs-microsoft-smooth-streaming-vs-adobe-hds/`.

[2] Paul Berberian. "How video streaming works on the web: An introduction". In: *Canal-Tech* 24 (Jan. 2018). URL: `https://medium.com/canal-tech/how-video-streaming-works-on-the-web-an-introduction-7919739f7e1`.

[3] Matthew Wolenetz et al. "Media Source Extensions". In: *W3C* (Nov. 2016). URL: `https://www.w3.org/TR/media-source/`.

[4] Mozilla Development Network. *The "codecs" parameter in common media types*. 2020. URL: `https://developer.mozilla.org/en-US/docs/Web/Media/Formats/codecs_parameter`.

[5] Mozilla Development Network. *Web video codec guide*. 2020. URL: `https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Video_codecs`.

[6] Odin Lindblom. *How to Choose the Right Codec and Container for Your Video Workflow*. 2020. URL: `https://www.videomaker.com/article/c03/18165-how-to-choose-the-right-codec-and-container-for-your-video-workflow`.

[7] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 3rd. Pearson, 2015. ISBN: 013409266X.

[8] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: An Efficient and Portable Web Server". In: *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. Monterey, CA: USENIX Association, June 1999. URL: `https://www.usenix.org/legacy/events/usenix99/full_papers/pai/pai.pdf`.

[9] Assorted. *Rust*. 2020. URL: `https://www.rust-lang.org`.

[10] Assorted. *Tokio*. 2020. URL: `https://tokio.rs`.

[11] Assorted. *Hyper*. 2020. URL: `https://github.com/hyperium/hyper`.

[12] Assorted. *nodejsDiagram*. 2016. URL: https://stackoverflow.com/questions/36766696/which-is-correct-node-js-architecture.

[13] Assorted. *nodejs*. 2020. URL: https://nodejs.org/en/.

[14] Assorted. *libuv*. 2020. URL: https://libuv.org.

[15] Ryan Dahl et al. *denoland*. 2020. URL: https://deno.land/manual@v1.9.2.

[16] Ryan Dahl et al. *denoarch*. 2020. URL: https://deno.land/manual@v1.9.2/contributing/architecture.

[17] Tomás Senart. *Vegeta*. 2020. URL: https://github.com/tsenart/vegeta.

[18] Gil Tene. *HDRHistogram*. 2020. URL: http://hdrhistogram.org.

[19] Pat Shaughnessy. *Downloading 100,000 Files Using Async Rust*. 2020. URL: http://patshaughnessy.net/2020/1/20/downloading-100000-files-using-async-rust.

[20] real identity unknown Caustik. *Node.js w/1M concurrent connections!* 2012. URL: https://blog.caustik.com/2012/08/19/node-js-w1m-concurrent-connections/.

[21] Various. *Deno Continuous Benchmarks*. 2021. URL: https://deno.land/benchmarks.

[22] Melissa Licciardello, Maximilian Grüner, and Ankit Singla. *Understanding video streaming algorithms in the wild*. 2020. arXiv: 2001.02951 [cs.NI].

[23] Yao Li and Emina Soljanin. *Rateless Codes for Single-Server Streaming to Diverse Users*. 2010. arXiv: 0912.5055 [cs.IT].

[24] Soumen Kanrar and Soamdeep Singha. "Content Delivery Through Hybrid Architecture in Video on Demand System". In: *Ingénierie des systèmes d information* 24.3 (Aug. 2019), pp. 289–301. ISSN: 2116-7125. DOI: 10.18280/isi.240309. URL: http://dx.doi.org/10.18280/isi.240309.

[25] Manjaiah D.H Hareesh.K. *Peer-to-Peer Live Streaming and Video On Demand Design Issues and its Challenges*. 2011. arXiv: 1111.6735 [cs.NI].

[26] Tim Brecht, David Pariag, and Louay Gammo. "accept()able Strategies for Improving Web Server Performance". In: *2004 USENIX Annual Technical Conference (USENIX ATC 04)*. Boston, MA: USENIX Association, June 2004. URL: https://www.usenix.org/conference/2004-usenix-annual-technical-conference/acceptable-strategies-improving-web-server.

[27] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. "A Scalable and Explicit Event Delivery Mechanism for UNIX". In: *1999 USENIX Annual Technical Conference (USENIX ATC 99)*. Monterey, CA: USENIX Association, June 1999. URL: https://www.usenix.org/legacy/event/usenix01/cfp/banga/banga.pdf.

[28] David Pariag et al. "Comparing the Performance of Web Server Architectures". In: *ACM* (Mar. 2007). ISSN: 978-1-59593-636. URL: http://course.ece.cmu.edu/~ece845/docs/pariag-2007.pdf.