A WebSocket-based Approach to Transporting Web Application Data

Ross Andrew Hadden

March 24, 2015

A thesis submitted to the
Division of Graduate Studies and Research
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
in the Department of Computer Science
of the College of Engineering and Applied Science
March 24, 2015
by

Ross Andrew Hadden
B.S., University of Cincinnati, Cincinnati, Ohio (2014)
Thesis Adviser and Committee Chair: Dr. Paul Talaga
Committee members: Dr. Fred Annexstein, Professor, and Dr. John Franco, Professor

Abstract

Most web applications serve dynamic data by either deferring an initial page response until the data has been retrieved, or by returning the initial response immediately and loading additional content through AJAX. We investigate another option, which is to return the initial response immediately and send additional content through a WebSocket connection as the data becomes available. We intend to illustrate the performance of this proposition, as compared to popular conventions of both a server-only and an AJAX approach for achieving the same outcome. This dissertation both explains and demonstrates the implementation of the proposed model, and discusses an analysis of the findings. We use a Node.js web application built with the Cornerstone web framework to serve both the content being tested and the endpoints used for data requests. An analysis of the results shows that in situations when minimal data is retrieved after a timeout, the WebSocket method is marginally faster than the server and AJAX methods, and when retrieving populated files or database records it is marginally slower. The WebSocket method considerably outperforms the AJAX method when making multiple requests in series, and when making requests in parallel the WebSocket and server approaches both outperform AJAX by a tremendous amount.

Contents

List of Tables

List of Figures

1 Background

The Internet has a long and storied history, but over the past few decades the HyperText Transfer Protocol (`HTTP`) has remained one of the most popular transports for data communication. `HTTP` (as well as the SSL-encrypted variant `HTTPS`) is the protocol that most web servers and browsers use to send data back and forth to each other. When a browser makes an `HTTP` request to a web server, the web server then sends a response to the request.

1.1 Server

Originally when clients made `HTTP GET` requests to load web pages, web servers would render the pages with all desired data present, and a new page would be requested with new content when data needed to be changed or updated. This works quite nicely when a website is just a collection of static pages. The pages may contain hyperlinks which, when clicked, take users to another page. The user's browser client makes another request to the same server, which carries out the same process and responds with the new page content.

Server-side programming languages like PHP Hypertext Preprocessor (widely known by the recursive acronym `PHP`) can make this process much easier to develop. Prior to such server-side languages, for example, web developers had to duplicate content on every static page just to have the same header and footer. With `PHP` they are able to include files such as a header and footer on every page. A more common approach eventually became to have a layout file which has the outside structure and shared content between, and include each specific page at a certain point within the layout template. This method has been widely adopted by most modern web frameworks. Although such techniques vastly improved the field of web development, they did so within the same confines of the static content era.

1.2 AJAX

Such a call-and-response workflow is still typical today for such content as markup, scripts, stylesheets, and images. As the need arose to load more dynamic data, however, better methods for loading in additional data were constructed. With the popularization of `AJAX` starting in 1999, this familiar approach to sending and retrieving data was used in a very similar manner. After the client loads a page, requests are made to obtain additional data as desired, without requiring a browser to refresh or change pages. This lets developers tie into a model similar to how the browser loads scripts, stylesheets, and images (via `<script>`, `<link>`, and `<img>` `HTML` tags, respectively), but for any arbitrary data. The static content era ultimately evolved into one where data did not need to be present upon initial page loads, and could instead be populated asynchronously.

This era of dynamic content has not only greatly improved the workflow for typical websites, but it has also completely re-imagined what is possible. Now that the web facilitates changing and refreshing data without loading entirely new pages, websites have been able to transcend their original purpose and become full-fledged applications. Projects that were formerly only possible in the native operating system software world are now not only possible as web applications, but already a reality.

1.3 WebSocket

In 2011, a standard was released for yet another method of transporting data between web servers and clients. This protocol—known as the WebSocket protocol—allows for full-duplex communications channels over a single `TCP` connection ("RFC 6455 - The WebSocket Protocol" 2015). Clients open a bidirectional connection with a server which is not terminated after the typical request-response cycle. Instead, the connection remains open, and either side may broadcast communications at any point.

In addition to bringing some newly conceivable workflows into fruition, the WebSocket

protocol also made it possible to better some that were already achievable before its inception. For instance, instead of a page making ten `AJAX` requests, it could receive all of the data from those ten requests through one single ongoing WebSocket connection. Unfortunately the WebSocket protocol did not immediately flourish outside of a few niche areas, such as game development and multimedia applications. This is likely due to Microsoft Internet Explorer not supporting it until version 10, which came out a year after the finalized WebSocket specification ("Can I Use" 2015, *websockets*). Even though Mozilla Firefox and Google Chrome had supported the protocol months before it was even finalized, in 2011 Internet Explorer still had the highest browser market share, and thus supporting it was deemed necessary by many developers ("Top 5 Desktop Browsers from Q3 2008 to Q1 2015" 2015). This is market share is depicted in Figure 1.

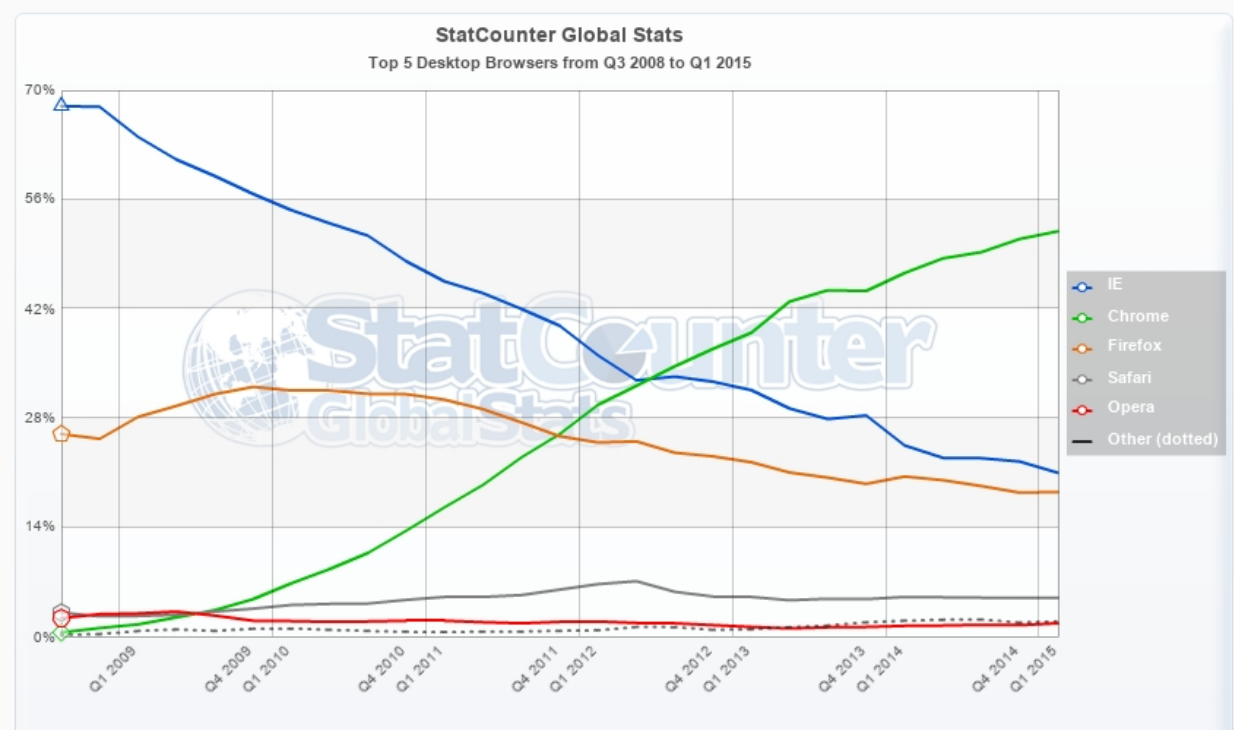

Figure 1: Desktop browser market share, Q3 2008 to Q1 2015.

2 Motivation

AJAX calls are made from JavaScript code, which initiate an asynchronous call to a specified endpoint. Once the data is received, a callback gets executed which has access to the data (or error) returned by the server, as well as some request and response metadata. While this process is very well accepted and works quite well, there are actually some places where it falls short.

The process begins with the server serving a requested page to a client. It's not until the client has received the entire response that it starts requesting more content. Next the browser makes requests for all of the external stylesheets referenced in the `<link>` tags in the `<head>` section of the `HTML` page, and any `<script>` tags that may be there as well. After these requests have been made (though not necessarily before the responses have been received), browsers make requests for `<script>` and `<img>` tags within the `<body>` section of the page. It is not until all of these resources have been loaded by the browser that the scripts begin to execute. It is worth noting that any `<script>` tags in `<head>` would actually execute *before* the rest of the page has finished loading, however it is a well-established best practice to place all scripts at the end of the `<body>` section ("Front-end Code Standards & Best Practices" 2012, *General Coding Principles*). This is because developers generally want to execute code only once the entire page has been rendered and its resources fetched, in order to avoid such occurrences as the dreaded "Flash Of Unstyled Content" (FOUC) and other things that negatively impact user experience (Souders 2007, 43).

After all external resources have been loaded, the JavaScript is executed and AJAX requests may finally be initiated. When the web server receives an AJAX request, it performs whatever processing or external calls are necessary, and then returns the response. This is theoretically a rather large gap in the process, as these external calls are not even initiated until the browser has loaded and parsed all of the above resources and makes requests for more data. It seems reasonable to surmise that this gap may in fact be mitigated, and therein lies the

primary motivation of this study.

Another factor in the above process is that of request latency. The time it takes for an HTTP message to be sent and received is a well-known bottleneck—one which can really only be abated by obtaining a faster connection to the Internet. If the number of requests being made could be reduced, however, the total latency would inherently decrease as well. Of course, the server *could* just do all of the rendering itself, which would limit the number of requests down to one. However this is potentially even worse than making too many requests, because the user would see only a blank white page until the server returns a response.

2.1 Cornerstone

Together with colleague Sean Clark, I made a back-end web framework called Cornerstone. Built on Node.js—the server-side JavaScript platform—Cornerstone makes writing APIs and web servers much more convenient. It transpiles code written using features from ECMAScript 6 (ES6) to code that can run in current, modern JavaScript environments, such as the v8 JavaScript runtime that powers Node.js. This means that developers are able to employ the use of syntax and concepts from newer versions of JavaScript that are not necessarily implemented yet in many environments.

Having direct access to a framework like Cornerstone as a laboratory put me in a convenient position for approaching some of the shortcomings discussed above. Not only could I experiment with implementing useful features, but I could also try them in large-scale applications to see how well they may affect such situations. This symbiotic relationship between Cornerstone and its features has served as another motivation of this study, as any gains in this territory would directly benefit the framework.

3 Overview

As discussed previously, the primary motivation for this study was to reduce a rather large latency gap between serving an initial web page and providing it data. This was achieved by employing the use of WebSockets in a creative way.

When the web server receives a request for a route from a client, it immediately sends a response back to the client with as much content as it can without fetching external data. If a route needs to make requests to get additional data to be used, these requests are initiated before the response has been sent. Unlike a typical web server, however, Cornerstone does not wait until the fetched data has been received in order to send a response to the client. Instead, it sends the received data through to the client as the data arrives, via WebSockets. The client is still free to do whatever it needs to with the data, and the data itself can still be anything at all, such as JSON or HTML.

This process has several inherent advantages over the typical AJAX workflow. For one thing, there is no gap between the server's initial response and the initiation of follow-up data requests. Requests for more data are made slightly before the initial response has been sent. The client never actually initiates requests for more data, and thus it does not have to wait for a web page to finish completely rendering in order to do so.

This process also effectively reduces the number of requests made. The server sends just as much data back via responses, but the requests themselves don't actually get made by the client, and therefore do not have any latency at all. Only one request ever really gets made, which is the very first request the client made to the server to load the main web page.

4 Details

There are a lot of pieces working together to make this work. The Cornerstone Node.js framework is itself built on top of another popular framework, Express, which gives it a very

solid foundation in terms of a powerful web framework. This has served as a great origin for adding even more convenient features, especially when coupled with transpiling the code with 6to5, providing access to many useful parts of ES6.

Promises are one such useful feature. Promises are objects used for deferred and asynchronous computations ("Mozilla Developer Network" 2015, *Global_Objects/Promise*). They make writing asynchronous code much more favorable and manageable than the callback-heavy alternative, and allow for passing around composable handlers for success and failure states of functions.

4.1 Process

Normally when data is passed from a controller to a Handlebars view template in Cornerstone, the values are substituted in-place when the template is rendered. For example:

```
// someController.js
let someController = {
    someRoute(req, res) {
        res.view({
            foo: "hello",
            bar: "world"
        });
    }
};

{{! someController/someRoute.hbs }}
<p>{{foo}}</p>
<p>{{bar}}</p>
```

This results in the following being rendered to the client:

```
<p>hello</p>
<p>world</p>
```

When one or more of the data values passed to a view template is a promise, however, it initiates the streaming feature. Consider the following example:

```
// someController.js
let delayedData = new Promise(function(resolve, reject) {
    // resolves the promise after 5000 ms
    setTimeout(function() {
        resolve("delayed data!");
    }, 5000);
});

let someController = {
    someRoute(req, res) {
        res.view({
            foo: "hello",
            bar: delayedData
        });
    }
};
```

In this situation, `bar` would not actually be rendered to the client. If `{{bar}}` were to be used in the template anyway, it would just be rendered as `[object Object]`, which is the result of JavaScript coercing a `Promise` into a `String`. Instead what happens is that Cornerstone serves the page before the promise has been resolved (which in this case means before the 5000 ms timeout has finished).

Once the client loads the page, it initiates a WebSocket connection to the server, using socket.io. There is an authorization handshake stage at the beginning of the socket connection in which Cornerstone establishes a bidirectional mapping between sessions and WebSocket connections, so that the socket connection of any given client may be accessed within any route handler. This bidirectional mapping is how Cornerstone knows which socket to send data to for each request. It also provides a way to access a client's session data from the scope of a socket connection, though that functionality does not come to play within this workflow. For each promise, the server binds a one-off listener for WebSocket connections, which tests for whether the connected client is the same client that is waiting for data. When this test passes, the server attaches a success handler for each promise that emits the resolved value of the promise to the client, through the socket connection. It then destroys the listener, which has at this point served its intended purpose and is no longer necessary.

Once the data gets to the client, a developer has several options on how to use the data. Since the data is just being sent through a WebSocket connection, the communication events may be bound to by the client in JavaScript, which provides access to the raw data being sent by the server. This is the most powerful option, as the client can do whatever it wants with the data.

Another method of using the data sent via the WebSocket transport is to use Cornerstone's `{{{stream}}}` Handlebars helper. This helper is very simple in implementation. It just returns a `<var>` HTML tag with a `data-promise` attribute set to the key of the promise.

For example, given the situation outlined above where `bar` is a promise being passed to a view, consider a view file with the following markup:

```
<p>{{{stream "bar"}}}</p>
```

The above view markup would be rendered and served to the client as the following HTML:

```
<p><var data-promise="bar"></var></p>
```

The purpose of the `<var>` tags is to serve as a placeholder for the deferred data to be rendered in. When a client receives a socket broadcast with the data for `bar`, it replaces this placeholder with the received data. The way that the placeholder replacement works is by using the key sent in the WebSocket broadcast to select the `<var>` tag with the matching `data-promise` attribute. This HTML-output method does not work well for all situations, of course, but when HTML or raw data needs to be displayed it is rather convenient.

5 Data

At its core, the testing performed is done so in an attempt to find the fastest or best available method for serving dynamic web pages to users. In most of the test scenarios, these pages contain data populated from external requests, where the client makes a request to the server for data. The data can be anything from a simple boolean or numeric value to the contents of an immense number of documents from a database. The need to deliver data to a client is a common one, and these tests were carried out with that in mind.

5.1 Methods

All of the testing scenarios executed are broken up into three different testing methods. These will henceforth be referred to as the "server", "AJAX", and "stream" workflows.

5.1.1 Server

The first of these methods, the server workflow, is implemented by having the server carry out all ancillary requests itself. After receiving a request from a client to load an endpoint that requires additional data, the server makes all of the necessary requests. Once all of the relevant data has been fetched, it serves the page to the client that requested it, with all pertinent data already included within the markup. Such a workflow is not always applicable

in web applications—especially for those with dynamic content—however this is nevertheless both a common and mature procedure for delivering data to clients. The longer the server takes to serve a page, the longer the user's client remains in a vestigial pending state. Thus the server workflow is meant to be more of a control case, against which the other two methods may be benchmarked in a recognizable fashion.

5.1.2 AJAX

The AJAX workflow is perhaps currently the most common standard for websites serving content any more elaborate than simple static markup. When an endpoint receives a request from a client that requires additional data, it serves back minimal markup without the data present. The client then makes AJAX calls to request the ancillary data. As the client receives the responses for this additional data, it does what it needs to do with it (which is in this case render it to the page and perform the necessary testing calculations).

5.1.3 Stream

The stream workflow employs the process detailed earlier in this paper, which is theoretically a middle-ground between the two aforementioned workflows, with the advantages of AJAX and the speed of the server workflow. In this process, the web server makes requests for any external data as soon as a client hits an endpoint, not unlike the aforementioned server workflow. The server then immediately responds to the client with minimal markup without the data present, which is more akin to the above AJAX workflow. As the server receives the responses for the additional data, it sends it to the client through an established WebSocket connection.

5.2 Tests

The tests carried out are enacted in such a way as to simulate common scenarios encountered in typical web applications. When a request is made for data, the server creates a `Promise`. The `Promise` becomes resolved once the request in question is done, with the data from the completed request. This `Promise` API is used for every test, even where it is unnecessary, for scalability and consistency.

The independent variable in these tests is the time between an endpoint receiving an initial request for a page, and the initiation of requests for external data. This intentionally-altered time is essentially the time that lapses until requests can made for data. The dependent variable is the time between an endpoint receiving an initial request for a page and a client receiving all of the data it needs. This is the time it takes for a client to be completely finished with its interactions with the server. Each individual scenario may also contain several variants, in which a secondary variable is altered in an attempt to establish trends within the scenario.

The way the dependent variable is measured is similar between each testing method. When an endpoint receives a request from a client, it immediately saves a reference to `Date.now()`, which "return[s] a `Number` value that is the time value designating the UTC date and time of the occurrence of the call to `now`," where the time value "is measured in milliseconds since 01 January, 1970 UTC" (ECMA International 2011, sec. 15.9.4.4). This `start` value is passed along with the markup for the pages served in each workflow. Once each testing method is complete, it calls a function `CS.diff`, which records a new value of `Date.now()`, and measures the difference between the two time values.

The only difference between workflows is the determination of the point at which the process is considered to be complete. For the server workflow, `CS.diff()` is run as soon as the browser executes the JavaScript code on the served page. Since the `<script>` tags occur immediately before the closing `</body>` tag, the code is loaded and run in an optimal manner ("Front-end

Code Standards & Best Practices" 2012, *Optimize Delivery of CSS and JavaScript*). The AJAX workflow runs `CS.diff()` once it receives all of the responses it is expecting for any given testing scenario. The stream workflow does the same as the AJAX one, though naturally it receives its data through a different transport.

5.2.1 Local file

In the local file scenario, a file is read from the local filesystem of the Cornerstone web server. The size of the file is altered between tests, ranging from an empty file to a significantly large one. This is a very common situation in web applications, which often need to read local files.

The files are made during an initialization process using the GNU `truncate` command and stored in `/tmp`, which is a `tmpfs` filesystem residing in memory on the server. When requested, these files are read from the filesystem using the asynchronous Node.js command `fs.readFile`. After a file is finished being read into the working memory of the web application server, it resolves the request `Promise` with the contents of the file.

Since the `truncate` command merely allocates the size for a file and does not fill it with contents, all of the created are "empty". To get around this, we actually send the raw `Buffer` data to the client, as opposed to the `UTF-8` encoded contents. For example, an "empty" file of size 4 B would be sent to the client as:

```
{
    "type": "Buffer",
    "data": [
        0,
        0,
        0,
        0
```

```
    ]
}
```

This is how `Node.js` represents octet streams, where `data` is an `Array` of bytes corresponding to the "raw memory allocation outside the V8 heap" ("Node.js v0.12.0 Manual & Documentation" 2015, *Buffer*).

See Figure 2 for the data captured for this test case.

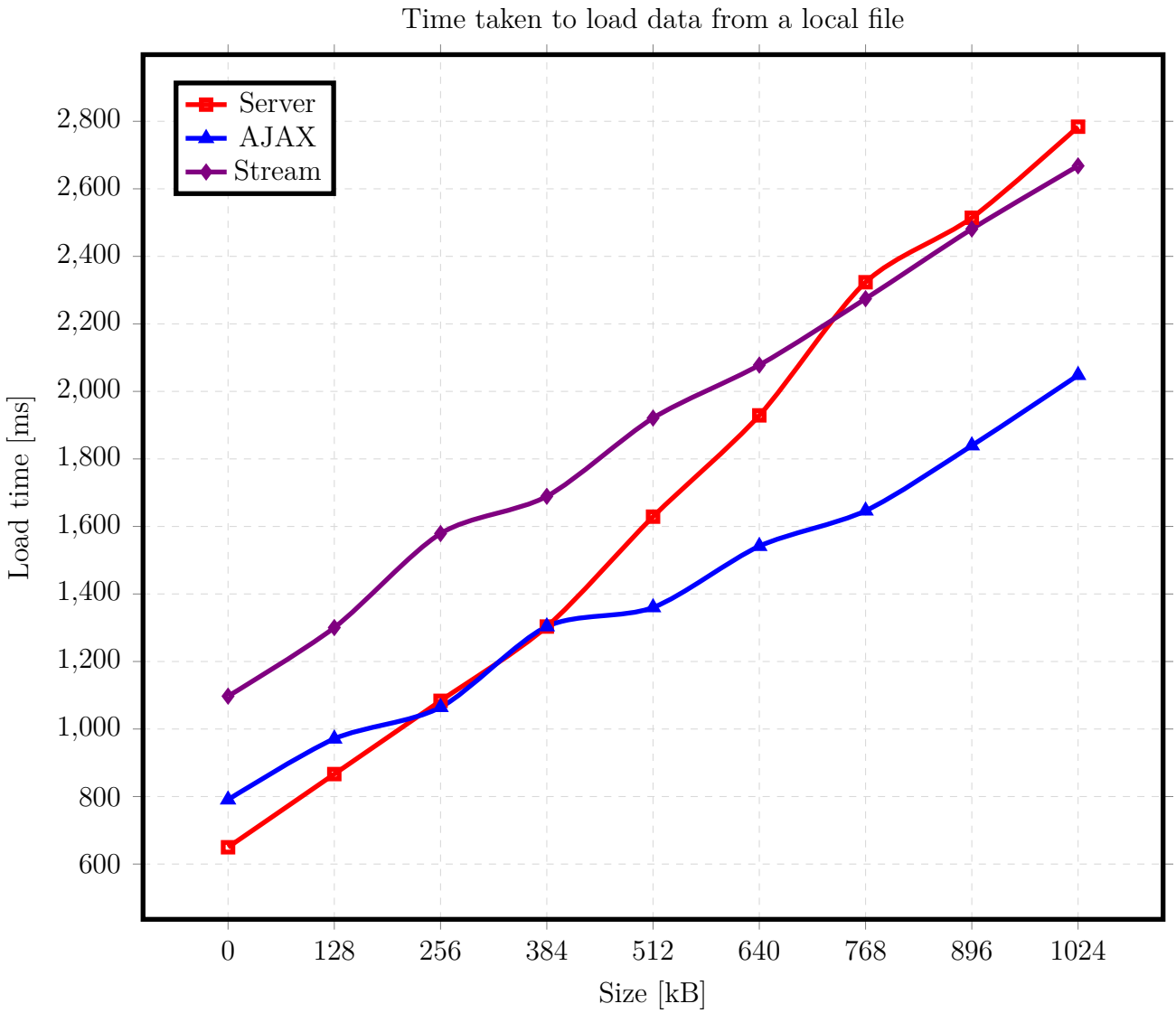

Figure 2: Local file workflow data.

5.2.2 Local database

Perhaps the most common scenario being tested is retrieving data from a local database. For our testing we are using a `mongodb` instance, with spurious data stored within several collections. Each collection contains a varying number of documents in the form of:

```
{
    "x": n
}
```

where `n` is an incremental value between `0` and the number of entries inside the collection.

See Figure 3 for the data captured for this test case.

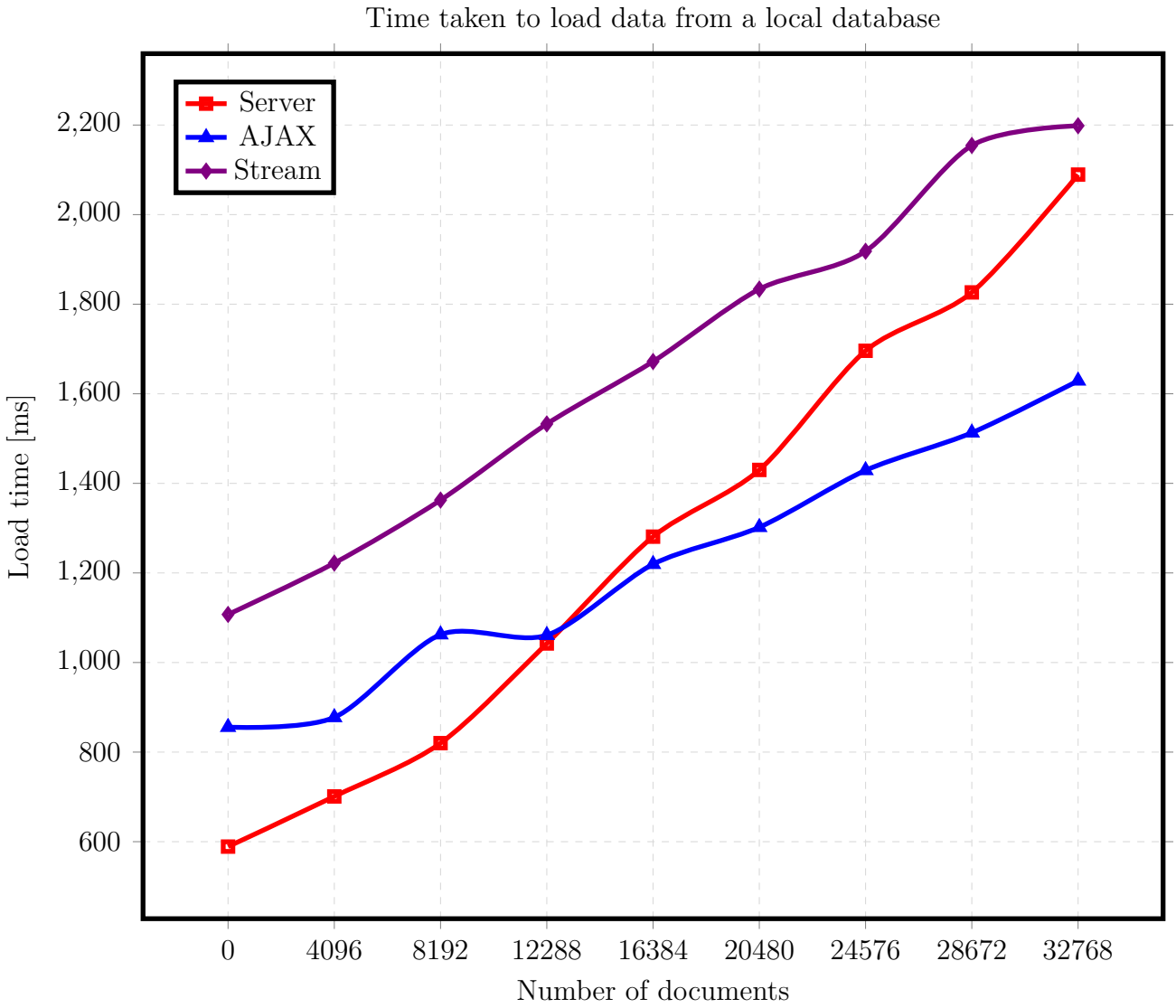

Figure 3: Local database workflow data.

5.2.3 Timeout

In the timeout scenario, the server sends responses to the client after a (varying) timeout. When a client hits the endpoint, the server runs a `setTimeout`, after which the response's promise is resolved. Unlike the previous scenarios, this is a largely contrived workflow on its own, however it is meant to simulate the often asynchronous nature of web applications.

See Figure 4 for the data captured for this test case.

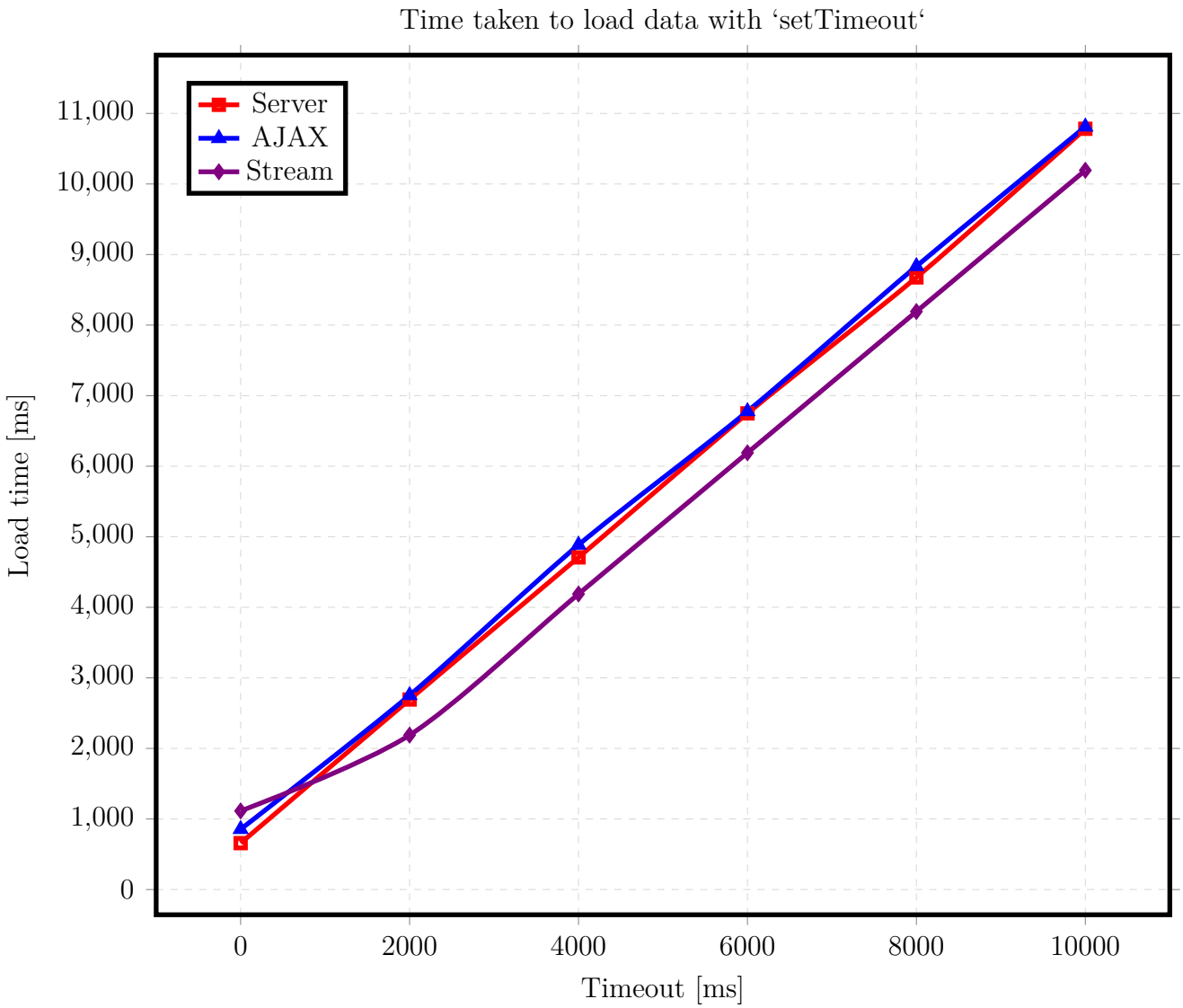

Figure 4: Timeout workflow data.

5.2.4 Series

The series test scenario involves having multiple requests that need to occur sequentially. That is, a request can not begin until the request that precedes it has been successfully resolved. The requests received are simply resolved after a timeout exactly like those in the timeout test. Instead of varying the timeout length, however, the value of the timeout is always 0 ms. Here rather it is the number of serial requests performed that changes between

trials.

See Figure 5 for the data captured for this test case.

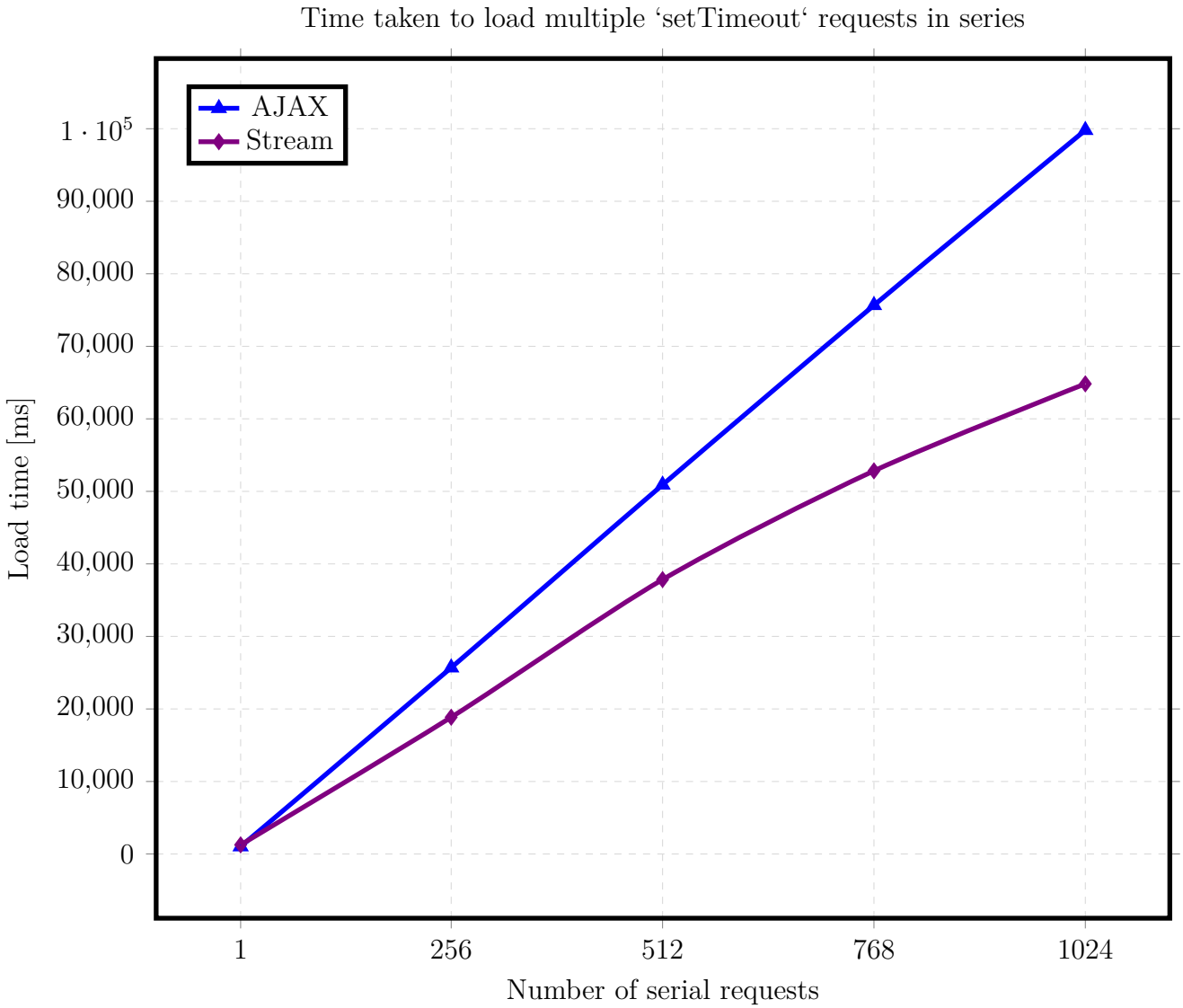

Figure 5: Series workflow data.

5.2.5 Parallel

Similar to the series test, the parallel test scenario deviates in that the requests are all initiated at the same time. This is a much more common scenario than the series test in practice, as requests are often autonomous. The number of requests performed is still altered

between different variants as in the series test, however the timeout duration is 2000 ms instead of 0 ms.

See Figure 6 for the data captured for this test case.

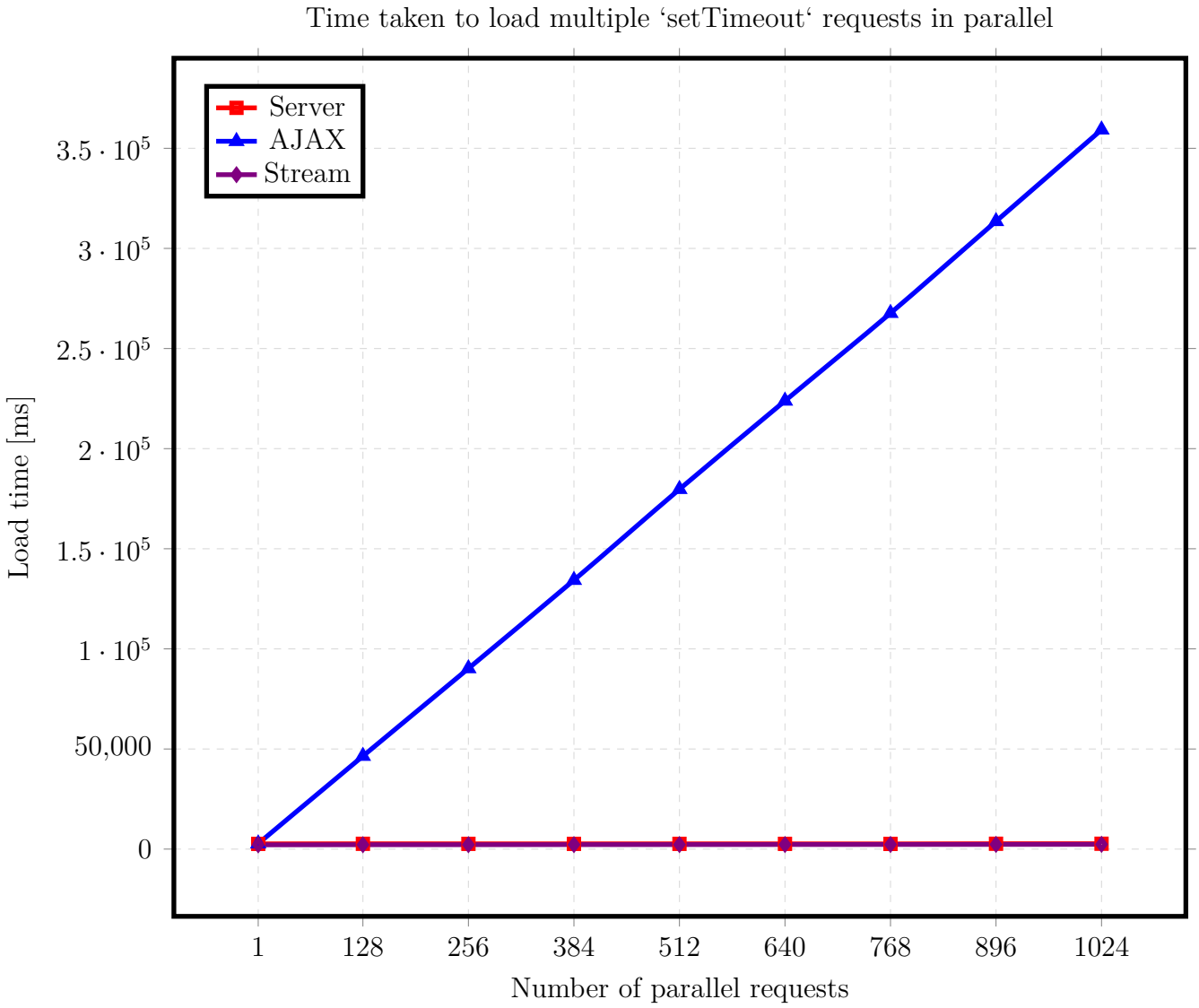

Figure 6: Parallel workflow data.

6 Analysis

6.1 Explanation

Since the request latency itself is not easily controlled as previously mentioned, the only sensible option is to limit the number of requests, which is the core concept behind all of the executed testing scenarios. The process outlined in the AJAX section requires $n + 1$ HTTP requests to a web server, where n is the number of necessary AJAX requests, and naturally $n + 1$ HTTP responses. This means that there are $2n + 2$ total data transmissions.

The process detailed in the stream section improves upon this by limiting the number of transmissions. Establishing the WebSocket connection is an event that happens only once per page. After that point, the connection remains open and may be used indefinitely. The stream workflow therefore only requires 2 requests to a web server. One request is made for the initial page endpoint, and one is made to establish a WebSocket connection. As there are still the $n + 1$ responses present in the AJAX workflow as well as one for the WebSocket connection, there are a total of $n + 2$ responses, and thus $n + 4$ total data transmissions.

The server process effectively limits those numbers down to 1 request and 1 response, however there are inherent problems with pursuing such an extreme as well. Most notably, users are presented with a functionless blank page until the response is received.

6.2 Results

The file and database tests (see Figure 2 and Figure 3, respectively) yield comparatively poor results for the stream method. Interestingly enough, the timeout test (see Figure 4) is consistently better for the stream workflow than for the server and AJAX workflows, however slightly.

Based solely on these three tests, the success and failure of the stream method seems to

be due to two distinct factors: transmitted data size and request duration. In the timeout test for example, the data that is being transmitted to the client is minimal. In fact the transmitted data is merely a small integer (the number of milliseconds the timeout was made for). The file and database tests are sending increasingly more loads of data to the client, as they are sending the contents of files and collections of varied lengths. It seems logical to conclude based on these tests that the stream method performs better when less data is transmitted.

The timeout test does better for the stream workflow in all cases except the first, which is when there is a timeout of 0 ms. I believe this is because at 0 s, there is not enough time for the stream method to benefit from bypassing the gap between loading the page and making the request. That is, by the time the client loads the page and establishes a WebSocket connection to the server, it is essentially equivalent to the AJAX method for a timeout of 0 ms in that in makes a request which is immediately resolved. This is because although the timeout finishes immediately after the original endpoint was hit, it still requires the additional request of establishing the socket connection in order to receive the data. In the trials with the higher timeouts, however, there is a period of time where the stream method benefits from making the timeout call as soon as the endpoint is hit. The data supports this theory because in every case of the timeout test (except the 0 ms case), the stream method is actually around 600 ms faster than the other two methods. This time of 600 ms happens to be around the time it takes to load a basic page from the web server.

It is worth noting that in the timeout test every trial for the stream workflow is extraordinarily close between repeated trials. Repeated trials in every other scenario were found to vary by around 100–200 ms between tests without any variables changing, and the values seen in the data section are averages of these numbers. The timeout trials were instead found to vary by only 0–3 ms between trials. The timeout test is the only test found to exhibit this behavior, and only for the stream method.

The series and parallel tests show much more promising results for the stream method. In the series scenario, the results of the AJAX and stream workflows very quickly diverge. Whereas the AJAX method yields values on a linearly-increasing scale, the stream method instead appears to adhere to more of a logarithmic scale. Here the benefits of WebSockets really start to become evident, as depicted in Figure 5. The reason the stream method performs so much better than AJAX when operating with serial requests is that there is only one single WebSocket connection, which remains open throughout the entire endeavor. In contrast, each one of the requests in the AJAX method makes a separate call to the server, creating a new connection. In the latter variant, there are 1024 separate AJAX requests made to the server, while there is only ever one WebSocket connection.

The series testing scenario is a logical extension of the theory that this entire paper was created for, which is that the gap between requests may be mitigated or even entirely circumvented by making intelligent uses of WebSockets. It takes this concept to an extreme, in which there are many such gaps omitted. There is overhead in making requests, such as the SYN and ACK handshake between client and server, as well as the HTTP headers that need to be sent with each payload. In bypassing the latency gaps, this overhead is eliminated as well.

The parallel testing scenario is even more intriguing. There is almost no overhead to the server and stream workflows. In fact, due to the relatively tremendous load time of the AJAX method skewing the range set of the data, the overhead is not even noticeable in Figure 6.

What is not clear based purely on the data is the reason why the AJAX method performs so poorly. This is actually because web browsers have a maximum concurrent connection limit per given hostname ("Browserscope" 2015, *network*). The tests were all performed in Chromium 42.0.2311.22, which has a limit of 6 concurrent connections. When the browser is asked to make a large number of asynchronous requests it opens as many connections as it can, opening more as previous requests are resolved. Since the parallel scenario employs a timeout duration of 2 s, each batch of connections lingers for that long before being resolved.

Thus depending on the web browser used, requests are made at a velocity of around 2–8 requests per second. The server and stream workflows are really only limited by the maximum heap size and memory availability in their runtime environments, which means their velocities are virtually infinite.

These findings for the parallel test were quite unexpected, as the concurrent connection limit was not considered when the implementation was conceived. The underlying concept is the same as it is in the series test, however, which is that the stream method does better when there are more request-response cycles for it to bypass.

7 Conclusion and Future Work

7.1 Conclusion

In summary, we may conclude that the viability of the streaming method for retrieving data is directly dependent on the situation. When making multiple requests in parallel, if the number of requests is greater than the maximum allowed concurrent connections for a given browser, it is much faster to use WebSockets as a data transport, or serve all data within the initial request to the server. If the number of requests is below this threshold then there is no tangible benefit to either method, however as this threshold varies between browsers, the web application has little control over limit. When making multiple requests in serial, WebSockets are also a more optimal data transport than AJAX, is the former does not have to observe as many request-response cycles. The WebSocket approach does not have to wait for a web page to fully render before making ancillary data requests. This benefit is realized more when making requests that return small amounts of data, such as in the timeout test. Conversely, when sending large amounts of data such as in the file and database tests, AJAX outperforms the WebSocket approach.

7.2 Future Work

Although we name the titular data transport the "stream" method, it is actually not benefiting from the use of "streams", merely using them as a one-time (though reusable) transport. Thus the research done in this thesis may be furthered by taking more advantage of the streaming nature of WebSockets in several ways. The main area of interest is that the data could be streamed to the client as it is read by the web application. Most of the tests carried out in this thesis would not be affected by this change, however if implemented intelligently the file test could take heavy advantage of this feature. Instead of returning the entire contents of a file to a client upon request, the web server could rather easily be made to stream the contents of the file as it is read. This would of course be irrelevant for files below a certain unascertained size threshold, however for larger files this could be an impressive enhancement. More tests would need to be created around this idea, to find the situations in which it is successful and those in which there is no payoff.

Another notable area that could be explored is sending data updates to clients after an initial data payload has been consumed. An important need for dynamic web applications is to constantly display updated data to users. The common approaches to doing this are to make repeated AJAX requests on an interval, implement "long polling" (which is when a client makes requests on a long interval that the server only resolves when data has changed), or to use the suggested WebSocket implementation. There are already many published studies on the comparison of these methods, but research could be done to relate it to the entire request-response cycle discussed within this thesis, and integrated with many of the testing scenarios covered within.

References

"Browserscope." 2015. http://www.browserscope.org.

"Can I Use." 2015. http://caniuse.com.

"Cornerstone." 2015. https://github.com/ConnectAi/cs.

ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

"Front-end Code Standards & Best Practices." 2012. Isobar. http://isobar-idev.github.io/code-standards.

"Mozilla Developer Network." 2015. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.

"Node.js v0.12.0 Manual & Documentation." 2015. Node.js. https://nodejs.org/docs/v0.12.0/api.

"RFC 6455 - The WebSocket Protocol." 2015. Internet Engineering Task Force. http://tools.ietf.org/html/rfc6455.

Souders, Steve. 2007. *High Performance Web Sites - Essential Knowledge for Frontend Engineers: 14 Steps to Faster-Loading Web Sites*. O'Reilly.

"Top 5 Desktop Browsers from Q3 2008 to Q1 2015." 2015. StatCounter. http://gs.statcounter.com/#desktop-browser-ww-quarterly-200803-201501.

"socket.io." 2015. http://socket.io.