Experiments with application throughput in a browser with full HTML5 support

Marat Zhanikeev a)

Department of Artificial Intelligence, Computer Science and Systems Engineering, Kyushu Institute of Technology, 680–4 Kawazu, Iizuka, Fukuoka, 820–8502 Japan

a) maratishe@gmail.com

Abstract: This paper proposes and tests in practice a new application design for browser-based streaming. Various streaming-related technologies are made possible under HTML5. This study measures the bounds of throughput which HTML5 applications can expect to achieve in practice.

Keywords: application throughput, WebSockets, WebWorkers, live streaming, HTML5

Classification: Multimedia Systems for Communications

References

1 Introduction

Web applications (webapps) today are limited by the HTTP request-reply cycle, also known as the pull paradigm [10]. Browsers are single-threaded. Binary data is not supported natively, so it has to be encapsulated in a data structure like JSON [2].

Because of these limitations, browsers cannot support streaming natively. One has to install a plugin (like Flash) to stream video, for example.

Browsers are completely different under HTML5 [3]. Two-way sockets are supported through the WebSockets API [5]. The use of WebWorkers [7] creates a multi-threaded application environment, which on modern computers uses multiple CPUs (cores). Native binary data can now be handled without encapsulation, which makes it possible to write/read binary data to/from local files [6].

This makes the new browser a very attractive application environment. However, no measurement study has been performed yet to define its performance margins. This paper performs such a study as a preliminary stage of a bigger research on multi-source live streaming in browsers similar to the well-known P2P live streaming [9]. The main focus of this study is to measure end-to-end throughput which can be supported by the browser depending on the technology used.

WebSockets have already been mentioned in literature on live video streaming in browsers [4, 5]. The main focus, however, is on a single-socket streaming. To control streaming quality, current research proposes adaptive streaming [12], where streaming rate is adjusted in realtime following changes in throughput. For example, in [11] such a method is considered for a browser-based TV.

This study shows that browsers under HTML5 can also use the methods from the literature on P2P live streaming. Two groups of methods exist: the methods that modify the BitTorrent protocol [8] and the methods which use the substream method [9]. This study can be used as a performance reference for both these methods.

It should be mentioned that WebWorkers are no longer restricted by the Same-Domain Policy and can sent requests to any domain. This makes it possible to implement P2P applications in browsers.

Regardless of the webapp explained above, this study is generic and can be viewed independently from the end application.

A disclaimer is due about the HTML5 standard [3]. Although the standard itself is not yet official, most of the new technologies mentioned in the standard have already been implemented by popular browsers today. Since some of contributors to the HTML5 drafts are from Google, the Chrome browser seems to be the early adopter of the new features. It is unlikely that the functionality mentioned in this paper will disappear from the final version of the HTML5 standard.

The software used for this measurement study, including the development release of the Chrome browser are made public and can be downloaded freely.
at [1]. The README with the code explains how to install, configure, and run the code.

2 Push vs pull

This section briefly explains the trivial reality of the push versus pull argument.

Let $d$ be one-way delay, $s$ the size of a chunk of data (chunksize), $m$ the number of chunks, and $S = sm$ the total volume of data. Then, with a given throughput $r$, the total time $T$ to receive all the data is:

$$T = m \left( \frac{s}{r} + 2d \right) = \frac{S}{r} + 2md,$$

(1)

which heavily depends on the value of $d$. The spirit of the push method is then:

$$\lim_{m \to 1} \left( \frac{S}{r} + 2md \right) = \frac{S}{r} + 2d.$$  

(2)

Also, for large $S$ we can ignore the one-time $d$:

$$\lim_{S \to \infty} \left( \frac{S}{r} + 2d \right) = \frac{S}{r}. $$

(3)

The trivial conclusion is then that push is better for streaming than pull simply because lowering the number of round trips improves end-to-end throughput of data.

3 The new design

Fig. 1 shows the new design of a streaming webapp. The webapp dispatch part at the top is the traditional part where the user enters a URL and the webapp itself, written in javascript (JS), is dispatched to client side. The rest is the new functionality.

WebSockets API needs a socket server listening on a different port, were the socket itself is opened by the client using the IP address and port of the server. In this paper, the server is written in PHP and uses the traditional

![Fig. 1. The new design using HTML5 capabilities.](image-url)
fork() method, where each client socket is fork-ed and becomes a process independent of the main server thread. The server part of the handshake is implemented in accordance with the description in [5].

Past handshake, the child server thread waits for a data request for a substream as in the traditional substream P2P method [9], i.e. with the starting position, chunksize and step. The step normally equals the total number of substreams that make the full stream.

On client side, WebWorkers are spawned by the Control for each substream. Each WebWorker becomes a new thread independent from the main browser thread. WebWorkers can create their own WebSockets. WebSockets receive binary data using the Uint8Array datatype [6].

Since WebWorkers run on independent threads, communication with the main thread is done by passing messages. Variable passing in messages can be done both by copy and by reference [7]. Both cases are tested later in this study.

Buffer in Fig. 1 is a generic module where data from multiple streams is aggregated into a usable form. In practice, data can be written to a file [6] or be fed into the buffer of a VIDEO tag. Data syncing as part of a cloud storage service is also possible. The practical uses for the aggregated data are out of scope. In this implementation, the Buffer module only shows the numeric status of aggregation and its progress in time.

4 Experimental setup

Fig. 2 shows the actual network setup used for the experiments. The environment has 1 Gbps physical capacity. No other traffic apart from the streaming itself exists in the network, thus, making it possible to measure the maximum achievable performance. Both server and client machines are 4-core 2.4 GHz Intel i5 machines. Portable version of Chrome browser (version 26 dev) is part of the code at [1].

Experimental setup uses two variables: chunksize and streams. Chunksize, randomly selected between 500 b and 100 kb, represents various operational environments, where smaller chunks make streaming more manageable in realtime. Streams, randomly selected from the list \{1, 2, 3, 5, 7, 10\}, denotes the number of parallel substreams. Total size of data is 100 Mb for

![Fig. 2. The actual network and software setup used for the experiments. The physical capacity of the network is 1 Gbps.](image-url)
all experiments.

Each run also randomly selects the streaming method from the following list. The pull is the traditional method, where each chunk requires a separate request. The push method uses multiple WebSockets but does not use WebWorkers, instead relying on the asynchronous nature of WebSockets themselves (AJAX). The push.worker.pass.by.copy method spawns a WebWorker which then creates a WebSocket for each substream, but passes binary data by copy, while the push.worker.pass.by.ref method passes binary data by reference.

5 Results

Fig. 3 shows all the results in one multi-plot figure. Over 5000 experiments were conducted, making sure that at least 25 runs per each combination of parameters were collected. The figure shows average performance for each set of parameters. Upper row shows performance for push methods and the lower row compares push methods to the pull method via the ratio of respective throughputs. Throughput unit is Mbps and chunksize unit is bytes.

Upper row of plots shows that throughput for push methods reacts to chunksize but saturates at about 2.9 (about 794 Mbps) past 10 kb chunksize. Use of multiple streams can further improve throughput to about 3.0 (near 1 Gbps). Minor advantage in throughput can be achieved for multiple streams if data is passed by reference. However, there is little difference between push
and `push.worker.pass.by.ref` results.

Lower row of plots in Fig. 3 shows that push has major advantage over pull. Even with the largest chunksize, the difference is still 2 orders of magnitude. Increasing number of streams has very little effect on the push/pull ratio.

6 Conclusion

This study shows through practical experimentation that browsers under HTML5 can support multiple streams with very high throughput. In fact, the results show that native support for binary data is extremely efficient and can operate at the line rate – 1 Gbps in the case of this study. Aggregation of data using multiple parallel substreams can further improve throughput.

In future work, the proposed design will be tested for practical application scenarios like P2P-style live video streaming. Although this study showed virtually no performance surplus from using WebWorkers, they are the only choice for P2P-like stream of aggregation from multiple sources. It is also expected that multistreaming will have a more stable performance compared to the single-stream case, as was previously shown in the literature on P2P live video streaming.