SCALABILITY OF HTTP PACING WITH INTELLIGENT BURSTING

Kevin J. Ma, Radim Bartoš, and Swapnil Bhatia

Department of Computer Science, University of New Hampshire, Durham, NH 03824
{kjma, rbartos, sbhatia}@cs.unh.edu

ABSTRACT

While streaming protocols like RTSP/RTP have continued to evolved, HTTP has remained a primary method for Web-based video retrieval. The ubiquity and simplicity of HTTP makes it a popular choice for many applications. However, HTTP was not designed for retrieving data with just-in-time tolerances; HTTP servers have always taken an as-fast-as-possible approach to data delivery. For media with known bandwidth constraints (e.g., audio/video files), HTTP servers can be enhanced and optimized by taking these constraints into account. For these data types, we present our architecture for an HTTP streaming server using paced output. We discuss the scalability advantages of our HTTP streaming server architecture and compare it with traditional HTTP server response times and bandwidth usage. We also introduce an intelligent bursting mechanism and consider its effects on end user experience.

1. INTRODUCTION

The popular definition of multimedia has evolved over time, as advances in technology have enabled new types of media interaction. The current focus of mainstream multimedia is undoubtedly streaming video. Over the past few years, Internet-based streaming video has become a commoditized fixture of modern culture. Schemes have been proposed to combat network congestion [1], however, network bandwidth has largely increased to meet the needs of video, and the performance bottleneck has moved back to the streaming servers. While server networking has been studied [2], streaming server architecture requires additional scrutiny.

From a networking perspective, video delivery breaks down into streaming vs. download. Streaming is typically associated with RTSP [3] and RTP [4]. Download is typically associated with HTTP [5], but divided into two categories: straight download and progressive download, with the latter using range requests to retrieve data in chunks. A third HTTP option, which we propose as *HTTP streaming*, uses paced data output. (Client side pacing, using range requests, is another HTTP option [6], however, it requires custom client software, which negates the ubiquity advantages of HTTP.)

Our results show resource efficiency advantages with our HTTP streaming architecture, over straight HTTP download. HTTP streaming also maintains the ubiquity advantage of HTTP, over protocols like RTSP/RTP or RTMP[1] [7]. Ubiquity is a key factor as we consider the future of multimedia. As browser-based dominance continues, HTTP will continue to play a crucial role in multimedia's evolution, and HTTP needs to evolve to meet the changing needs.

While RTSP/RTP have a number of streaming related enhancements, two of the key features are network resilience and bandwidth management. The former relies on frame-based packetization and unreliable UDP transport to allow graceful degradation during packet loss. However, with the advances in network infrastructure, failures are less frequent, and most content providers would prefer to ensure high quality, with no packet loss, using TCP. For the latter, the pacing employed by our HTTP streaming architecture brings bandwidth and quality management to HTTP. RTSP/RTP also has the disadvantage of requiring the creation of multiple UDP connections between server and client, which many firewalls do not allow.

Performance evaluations exist for traditional Web Servers at a network level [8], but not at the architectural level. Other reports confirm the performance of the download model compared to other streaming schemes [9]. In this paper we dive a step deeper and compare different HTTP architectures (i.e., HTTP streaming vs. HTTP download), from a video delivery perspective. HTTP download is still widely used for video retrieval on both desktop and mobile platforms. Most RTMP and RTSP/RTP servers support HTTP tunneling for its firewall traversability. Windows Media™ and QuickTime® desktop clients support both RTSP/RTP and HTTP. Windows Media™ mobile player, until recently, only supported HTTP, and QuickTime® mobile player continues to only support HTTP download[2]. In this paper, we compare the characteristics of our own HTTP streaming server implementation with that of the de facto standard Apache HTTP server. We also introduce intelligent bursting and detail its effect on HTTP streaming performance.

2. APACHE HTTP SERVER ARCHITECTURE

Traditional HTTP servers are optimized for delivering web page content, which typically consists of many small files. Small files typically imply short-lived connections. For video, however, file sizes are large and connections are much longer lived. With short-lived connections, the number of concurrent connections is much smaller. Servers which are optimized for fewer concurrent connections may suffer from head-of-line blocking in the request queue, with traffic primarily composed of long-lived video streams.

The Apache Web Server is the de facto standard in open source Web serving. Apache spawns a bounded number of new processes to handle incoming requests, where each request is assigned to its own process in a run-to-completion model. Output data is sent as fast as possible, with fairness between active connections (processes) managed by the underlying OS.

3. ZIPPY HTTP SERVER ARCHITECTURE

The Zippy streaming server we developed uses a single thread for managing all sessions, rather than a process per connection. Ses-

[1]While the ubiquity of RTMP in Web-based video distribution could easily be argued, the proprietary nature of RTMP make it inaccessible to server and protocol developers.

[2]While iPhone™/iPod® Touch use range requests, the default is to request the entire file, in a degenerate case that mimics straight download.

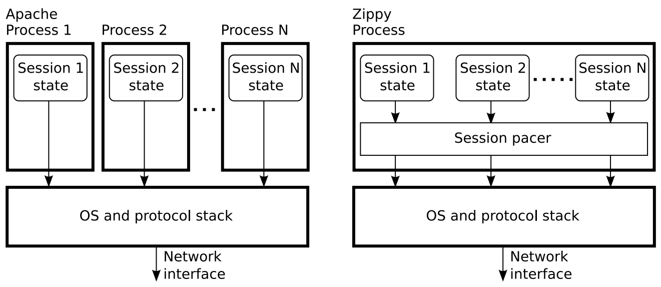

Fig. 1. Comparison of Apache and Zippy architectures.

sion state is managed by our session pacer rather than by individual processes. Fig. 1 shows the difference between the Apache multi-process architecture and the Zippy single thread architecture. With Zippy, connection fairness is explicitly enforced by the session pacer, rather than the OS scheduler. While other single-threaded HTTP server options exist, Zippy is focused on using fair access to resources, via the session pacer, to better support large numbers of concurrent long-lived connections. The large number of parallel sessions provides an advantage over servers capped by process limits.

The session pacer maintains sessions S_i in a heap, ordered by the sessions' next absolute send time. Absolute send times are calculated as current wall clock time plus the pacing delay minus any overhead: $T_{send(i)} = T_{now} + \delta_{pace(i)} - k_i$. The pacing delay is calculated using Zippy's fixed chunk size and a known constant bit rate (assumed for simplicity) for the session: $\delta_{pace(i)} = c/r_i$. The constant bitrate is derived from the file size divided by the file duration: $r_i = s_i/d_i$. Overhead includes processing latency and catch up delays as described below.

Zippy employs intelligent bursting mechanisms: one to catch up sessions when network latency inhibits chunk sends; another to decrease playback latency for media files. During periods of network congestion, full or near-full TCP buffers may cause partial sends or send failures. In such an event, future pacing delays are shortened to help the sessions catch up[3]. This adaptive bursting is used to prevent network issues from causing underrun.

User experience can also be enhanced through client buffer preload. To combat jitter and prevent underrun, video file playback typically will not commence until a sufficient amount of video has been buffered. With paced output, the playback buffering latency negatively effects user experience. This can be avoided by bursting the initial portion of the media file.

Zippy manages bursting by monitoring bandwidth thresholds to prevent bursting sessions from interfering with the minimum bandwidth requirements of paced sessions. Bursting uses only excess system bandwidth, divided evenly between all bursting sessions or only high priority sessions.

4. EXPERIMENTAL RESULTS

For our experiments, Zippy and Apache 2.2.6 were installed on a server with a 2.2 GHz Core2 Duo CPU and 2 GB RAM, running FC8. (The Apache version corresponds to the default httpd installation for the FC8 distribution used.) To ensure that the test client is not a limiting factor, test client software was installed on a machine with

[3]While a larger chunk size (rather than a shorter delay) could be used for catch up, if network congestion is the cause of the failure, then the TCP window is most likely limiting how much data can be sent.

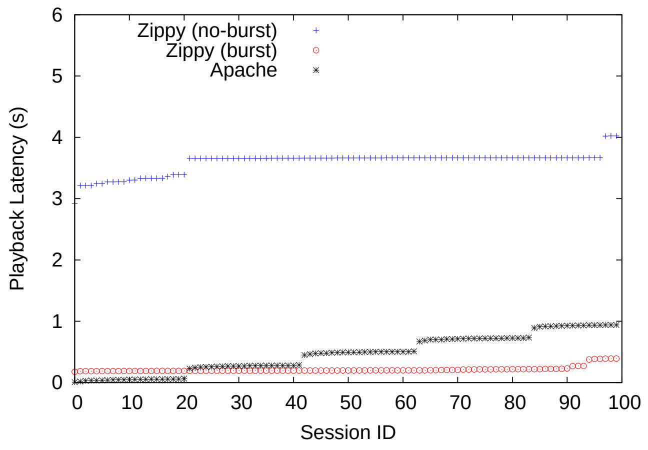

Fig. 2. Playback latency for 100 concurrent sessions.

dual 2.5 GHz quad core Xeons and 4 GB RAM, running RHEL5.1. The machines were connected via a Gigabit Ethernet network.

The tests were performed using a 1 MB data file. A constant bit rate of 400 kbps was assumed, which gives the file a duration of 20 seconds. The client buffering requirement was assumed to be 4 seconds (or 200 KB). Client underruns checks were performed against the known constant bit rate, for each packet after the initial buffer load (i.e., first 200 KB).

The test client is a multithreaded application which spawns a thread per connection. The connections are initiated as a flash crowd, with 500 microseconds between each request. Timestamps were recorded relative to the start of the test, as well as relative to the first TCP connection attempt. A sniffer monitored actual bandwidth used. As scalability was our main focus, for each test, we examined the performance of 100 and 1000 concurrent connections.

4.1. Playback Latency

We consider playback latency as the amount of time required to send enough data to fill the client buffer. Given our assumption of a 4 second buffer, streamed output without bursting should take less than 4 seconds to send the 200 KB. For a single straight download, over Gigabit Ethernet, 200 KB should take about 2 milliseconds, plus overhead. Figs. 2-3 show the playback latencies for each of 100/1000 sessions, respectively. The latencies are offsets from the first TCP connection request, in seconds, sorted from low to high.

In Fig. 2 the Zippy no-burst line, as expected, is consistently just below 4 seconds. The Zippy burst line shows a much lower latency, but with similar consistency across all sessions. The first 20 Apache connections are must faster than Zippy (burst or no-burst). The first 20 Apache connections take about 60 milliseconds ($\sim$ 3 milliseconds per connection). Taking into account overhead, the Apache results are as expected. The rest of the Apache plot, however, looks like a step function. The steps represent the head-of-line blocking and latency of run-to-completion download.

In Fig. 3 we can see that Apache performance is noticeably worse, compared to 1000 sessions. At a certain point, Apache's head-of-line blocking delays begin to cause TCP timeouts and the TCP back off causes more significant latency penalties.

With 1000 sessions, the total bandwidth requirement goes up significantly, which inhibits Zippy's ability to burst. We can see this

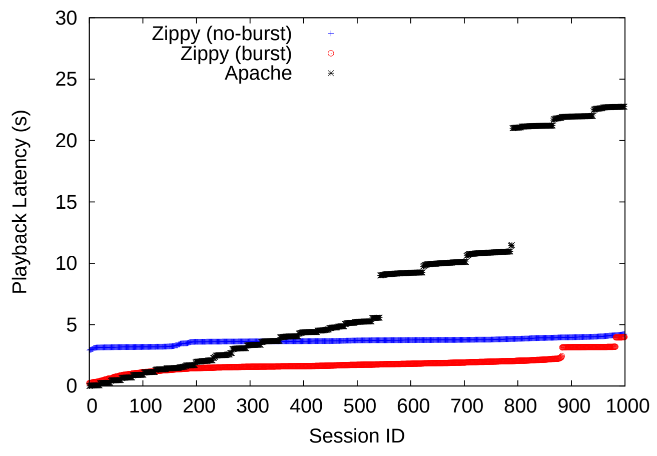

Fig. 3. Playback latency for 1000 concurrent sessions.

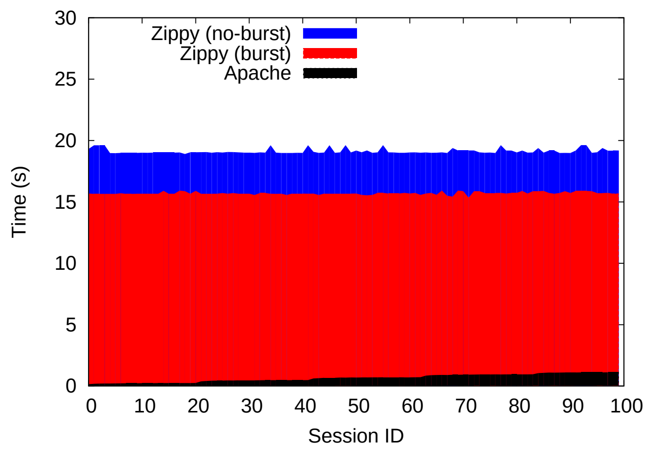

Fig. 4. Download time for 100 concurrent sessions.

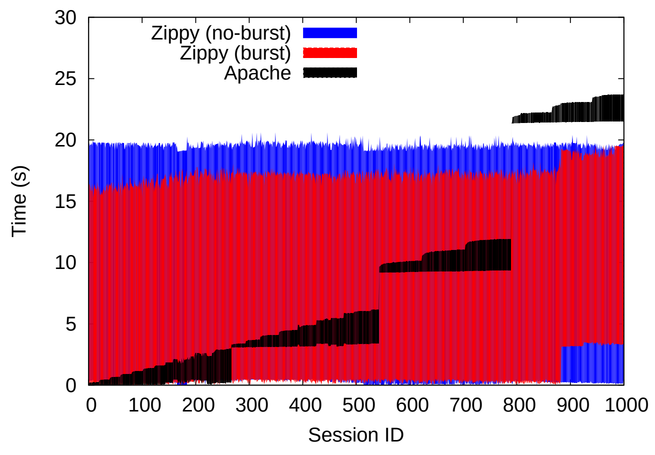

Fig. 5. Download time for 1000 concurrent sessions.

in Fig. 3 as the playback latency for Zippy burst and Zippy no-burst converge. However, the worst case for both bursting and not bursting is still significantly better than Apache.

Apache is faster than Zippy (burst or no-burst), for 20 or fewer connections. This is due to the default Apache process limit for the given machine. The Apache process limit maybe manually increased, however, the strain on system resources is great, when managing 1000 processes. Zippy consumes far fewer resources at 1000 concurrent sessions, and its consistency in processing all sessions in parallel gives it a noticeable advantage in response time.

4.2. Download Time

We consider download time as the relative time at which the entire file download completed. Given our assumptions of a 20 second file duration, streamed output without bursting should take less than 20 seconds from the time the HTTP connection is accepted. For a single straight download, over Gigabit Ethernet, 1 MB should take about 10 milliseconds, plus overhead, from the time the HTTP connection is accepted. Figs. 4-5 show the download start and end times for each of 100/1000 sessions, respectively. The times are offsets from the start of the test in seconds, sorted from low to high.

In Fig. 4 the Zippy no-burst line, as expected, is consistently just below 20 seconds. The Zippy burst line is consistently at about 16 seconds, which takes into account the 4 second burst, followed by pacing thereafter. The Apache download times are dwarfed by the paced completion times. In the worse case it takes little more than 1 second to complete the straight download, which is as expected.

In Fig. 5 we can see again that for 1000 sessions, Zippy performance is about the same, but Apache does noticeably worse. The last 100 or so bursted sessions did not have enough excess bandwidth to really burst, however we can see that those sessions still beat the no-burst deadlines. Apache, on the other hand, due to the exponential backoff in TCP, takes significantly longer to download the last 200 or so connections. Even though the total time to actually download is less, the user perceived time is quite high.

For larger files, straight download latency gets worse, and more TCP timeouts occur. Compounding this is that many types of clients (esp. mobile) are unable to buffer entire files, which causes TCP back pressuring. This only exacerbates head-of-line blocking issues.

4.3. Bandwidth Usage

We consider bandwidth usage as an aggregate for the entire server. Given our assumptions of a 400 kbps second constant bit rate, streamed output without bursting should require 400 kbps per active connection. Figs. 6-7 show the bandwidth used in the 100/1000 sessions cases, respectively. The bandwidth (in Mbps) is calculated, over time, as an offset (in seconds) from the start of the test.

In Fig. 6 the Apache plot is clustered within the first second and close to the practical capacity of the Gigabit Ethernet network and the OS protocol stack. The Zippy burst plot also has a marker close to the network limits, at the very beginning, representing its burst, then periodic bursts of data are seen. A similar pattern of periodic bursts is seen for the Zippy no-burst plot, but shifted to the right, given the longer duration. The end times times for the Zippy burst and no-burst plots are at the expected 16 and 20 seconds, respectively and the calculated average bandwidth used, over the full 16/20 seconds, is close to the expected 40 Mbps.

The irregular burstiness of Zippy plots is an artifact caused by data send clustering and offset sampling. Data send clustering occurs when all sessions are initated at the same time, as with our flash crowd scenario. This synchronization manifests itself as bursty

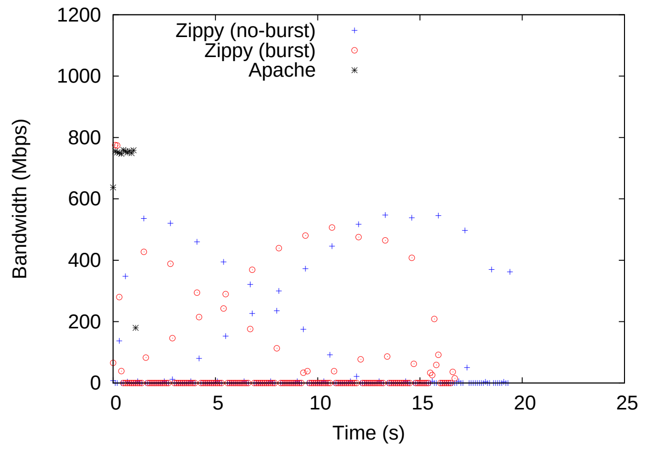

Fig. 6. Bandwidth usage for 100 concurrent sessions.

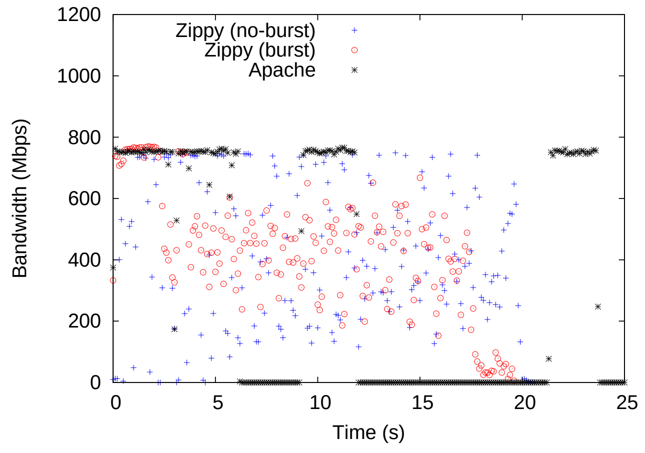

Fig. 7. Bandwidth usage for 1000 concurrent sessions.

bandwidth usage. Average bandwidth used is actually much lower due to the pacing delays. Offset sampling is the difference between pacing rate and bandwidth sampling rate. When the burst crosses a sampling boundary, a high and low bandwidth measurement are seen; the offset sampling rate ensures that boundaries will be crossed at different points within the burst.

In Fig. 7 the Apache plot is again always at maximum bandwidth, with holes representing the TCP backoff. The Zippy burst plot shows the burst at the beginning and tails off at about 16 seconds. The Zippy no-burst plot is relatively evenly distributed.

5. CONCLUSIONS AND FUTURE WORK

We have shown the scalability value of a single threaded, paced architecture for HTTP streaming. This architecture enables connection fairness for a larger number concurrent connections, while still maintaining the ability to use greedy delivery for a smaller number of concurrent connections. Traditional HTTP servers are optimized to service short-lived connection requests, but they are suboptimal for long-lived connections (e.g., live and on-demand audio/video, as well as other emerging live data streams). The browser continues to be the preferred medium for distributing all types of streaming media and as such, HTTP servers need to evolve to incorporate optimizations for these new classes of realtime media. We believe that our architecture is a step in that direction.

We continue to explore new aspects of HTTP streaming scalability. We believe that combining some of the streaming advantages of RTSP/RTP with the ubiquity, simplicity, and robustness of HTTP provides an optimal solution for practical deployments, especially in the case of mobile devices. We are evaluating different greedy bursting schemes, for maximizing bandwidth usage, as well as investigating their effects on different traffic profiles (including different types of audio/video files, as well as alternative types of streaming media, e.g., microblogging or ticker data). We are also looking into different jitter injection mechanisms (similar to BFD [10]) to combat data send clustering, without impacting user experience. The future of multimedia is going to include more categories of realtime data and new modes of interactivity. With HTTP the likely transport mechanism, these new traffic patterns need to be studied so that HTTP server architecture can be properly optimized for the future.

6. REFERENCES

[1] C. Chen, Z. Li, and Y. Soh, "TCP-friendly source adaptation for multimedia applications over the Internet," *Journal of Zhejiang University - Science A (JZUS-A)*, pp. 1–6, February 2006.

[2] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey, "Server Network Scalability and TCP Offload," in *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005, pp. 209–222.

[3] H. Schulzrinne, A. Rao, and R. Lanphier, "Real Time Streaming Protocol (RTSP)," RFC 2326, Internet Engineering Task Force (IETF), April 1998.

[4] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550, Internet Engineering Task Force (IETF), July 2003.

[5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Internet Engineering Task Force (IETF), June 1999.

[6] N. Färber, S. Döhla, and J. Issing, "Adaptive Progressive Download Based on the MPEG-4 File Format," *Journal of Zhejiang University - Science A (JZUS-A)*, pp. 106–111, February 2006.

[7] L. Larson-Kelley, "Overview of streaming with Flash Media Server 3," February 2008, *http://www.adobe.com/devnet/flashmediaserver/articles/overview_streaming_fms3_02.html.*

[8] T. Shinozaki, E. Kawai, S. Yamaguchi, and H. Yamamoto, "Performance Anomalies of Advanced Web Server Architectures in Realistic Environments," in Proceeding of IEEE International Conference on Advanced Communication Technology, 2006 (ICACT 2006)*, Feb 2006, pp. 169–174.*

[9] Y. Won, J. Hong, M. Choi, C. Hwang, and J. Yoo, "Measurement of Download and Play and Streaming IPTV Traffic," IEEE Communications Magazine*, pp. 154–161, October 2008.*

[10] D. Katz and D. Ward, "Bidirectional Forwarding Detection," Internet Draft Version 8 (draft-ietf-bfd-base-08), Internet Engineering Task Force (IETF), March 2008.