Model and Implementation of a Bitrate-Peeling Audio Stream System

Matt Crawford D.J. Gallagher Thomas Michon

December 18, 2006

Abstract

In this paper, we investigate and model a *bitrate peeling* audio streaming method as a way of dealing with the bandwidth and congestion constraints imposed on network data streams. We suggest that, in contrast to constant bit-rate systems with their associated audio encodings, bit-rate peeling and its optimized encoding offers an adaptive response to fluctuations in available bandwidth and is still capable of playing data continuously to the end user at a high perceived quality level. To make these comparisons, we propose a simplified model of audio data and of each streaming method. We then simulate our proposed method against a workload model that both limits available bandwidth and imitates congestion.

1 Introduction

Streaming is the primary paradigm for delivering multimedia over a network. By updating the media in sync with consumption, it starts playback sooner and limits download speed to what is useful. End users expect to receive and play audio and video at a constant rate without delays or interruptions in the presentation. Users also expect their media to be high-quality, a perception based on the amount of detail available in the signal (which is linked to the data rate). In this paper, we investigate audio streaming only, though some of the concepts (such as layering) can also be generalized for video streaming. There are a wide variety of protocols and algorithms for audio streaming in use by various institutions and available as packages from various companies and organizations. Using one of these protocols as a primary reference, we create a simplified model of an audio stream with specific properties, and investigate techniques to provide audio content to the end user at a high perceived quality and with minimal interruption.

To handle the obstacles of congestion and limited bandwidth experienced on a real network while maintaining continuous playback for the end user, it is desirable to be able to effectively implement *bitrate peeling* in streaming audio data. Bitrate peeling is a mechanism for coping with data lossage, which the Xiph.org Foundation has said it aims to include in future releases of the open-source Ogg Vorbis audio codec. A bounty is currently on offer for the source code implementation. It involves the division of a single frame of audio data into many layers of quality by partitioning the base units (frequencies, wavelets) of the given encoding scheme. At a minimum, the client needs just one layer per frame in order to play continuous audio, but at a significant loss in quality. Depending

on available bandwidth, the stream server can adjust the number of layers per frame on the fly, ensuring that the client experiences the greatest possible quality during playback.

This method of streaming has several advantages over alternatives (such as proprietary ones from Apple and RealNetworks that require multiple encodings of the same media to be stored). It does not take up any additional space. It does not impose computational overhead for "on-the-fly" re-encoding of audio, since it is trivial to drop unwanted layers from the stream (the client does have to process more information in reassembly, however). Lastly, the stream server will be able to nimbly and subtly adapt its transmission rate to accommodate network strain, because peeling offers fine-grained control over the quality level of the outgoing audio.

2 Simplified model of audio data

In this paper, we treat audio data as a sequence of *frames* representing the audio data for a duration of time t. The *stream server* sends audio frames sequentially to the *stream client* over the network. When the *stream client* receives these frames, it rebuilds the audio data and plays it to the end user. Playback on the client computer occurs at a rate ν measured in frames-per-second, which represents the minimum rate at which frames must arrive at the client. Frames added ahead of the current location of playback are acceptable and generally preferable; however, if playback reaches the end of the last available frame, there will be a gap in the audio, which is not acceptable. One thing to note is that ν is also the rate at which the audio is encoded in frames-per-second.

A frame represents the audio data corresponding to a fixed length of time, but there is no requirement on the amount of data it contains. In general with digital audio, the perceived quality of an audio signal is a diminishing-return function of the amount of data used to encode each time unit. The widely used term for this encoding density is *bitrate*, typically measured in bits-per-second. High bitrate encoding usually sounds better than low bitrate encoding, but quality comes at the price of larger total storage space for the audio data and greater minimum bandwidth required to transfer the data at rate ν across the network.

For the purposes of our model and experiment, we choose to refer to measure the data density ρ not in kilobits-per-second, but in data-samples-per-frame. This is because we have quantized our representation of audio into frames, and data does not have to be measured in bytes. The simple conversion to bit-rate is $b = k\rho r$, where k is a constant representing the size of a data sample in bytes.

2.1 Constant bitrate audio

If audio is encoded at a constant density ρ_C, then the resulting bit-rate b_C is also constant. Assuming frames are sent out at the minimum required rate ν, then the necessary bandwidth to avoid delays in playback at the client is simply b_C. Thus, when the stream server receives a request from a client to open a new stream, it starts sending frames at rate ν and the client begins playback immediately after receiving its first frame.

This idealized implementation of streaming has a few issues. The main issue is that the minimum bandwidth must be guaranteed for the entire duration of the stream session. Since, with a constant-

bit-rate encoding, the client requires the entire contents of each frame in order to play it, any strain on bandwidth would force the server to send frames at a rate less than ν, which would delay playback at the client.

The problem of bandwidth limitation can be solved in two ways. The first is by re-encoding the audio at a lower bit-rate. In exchange for a lower quality signal, the client receives enough audio data per time-unit in order to play continuously. However, unless this data is re-encoded for each client depending on each client's bandwidth considerations, all clients would be forced to suffer with the lowest-quality encoding. The second involves "buffering" some of the data at the client before the client starts playback. This usually only works if the length of the file is known, so that the entire audio is fully streamed to the client before playback reaches the last frame. Unfortunately, while this works for short files at a known bandwidth, it is not very helpful with "live" audio streams or fluctuating bandwidth.

Another drawback to the constant-bitrate system is that it does not handle congestion well. Although the bottleneck bandwidth may be sufficient, there is no good way to predict the throughput of the system (without approximating it via historical analysis), and the only real solution presented by the constant-bitrate system is to offer lower quality audio. Variable bitrate coding, which we will not explore here, is primarily a means of lowering the average bitrate while keeping perceived quality at a maximum.

3 Simulating a bitrate-peeled stream

The streaming model has 3 components: a *stream server*, a *stream client*, and a *network* that links the two. The server has an *audio source* from which it draws *frames* at a rate of 1 frame per second. The stream could be from any source, including a 'live' recording; in fact, this is very similar to adaptive multi-rate speech encoding in cellular phones. Frames consist of up to 32 *layers*, each containing 8 kilobits of information with which to refine the encoded signal, for a maximum bitrate of 256 kilobits per second. To make tracking easier, individual frames and layers are indexed (starting at 0).

3.1 The Server

The server is responsible for periodically taking a frame of audio from the source and sending this frame to the client. The simplest possible implementation of a server would simply take in frames from the source and send them to the client verbatim. This is not an ideal streaming solution because it does not account for either bandwidth or congestion. On a slow-moving network, this will cause frequent pauses or skips in what the client hears. However, this is exactly what most streaming protocols do: they choose from one or more available bitrates at startup, download the entire audio clip at that bitrate, and attempt to resolve any issues by buffering a large amount of audio data into memory before beginning playback. If these protocols used bitrate peeling, playback could begin sooner and less client resources would be consumed.

The server in our model can use any of a variety of bitrate peeling strategies. In each, the streaming application queries the network daemon for information on the size and percent consumption of its input queue, and from this determines the number of layers to send. The remaining, higher

layers collectively balk, and are discarded; they are less important than the ones preceding them. Different peeling criteria will work best under different load conditions, and will change the way the user perceives degradation in the quality of network service. Inadequate criteria will overflow the network queue, causing either a system error or begrudging accommodation by the network daemon (our simulation responds in the latter fashion).

3.2 The Client

The client is responsible for reordering and repackaging what it receives, and playing back the frames as timely and in as high quality as possible. We did not devote much time to perfecting the client, so our current implementation is fairly simple. It effectively handles jitter, or misordering of packets (a detail we were not able to encapsulate neatly in our TCP implementation) but does little to help protect itself from short pauses in playback. It continuously listens for incoming layers, which are placed on an in-memory layer queue and fed into a frame constructor. The constructor is optimistic (i.e. it tries to complete frames), but if it reads the beginning of a new frame it will boot the current frame under construction onto the frame queue for playback. A smarter client would perform pessimistic construction, consuming any available layers when the frame queue runs dry.

3.3 The Network

A focal point in our project was the creation of a network with realistic behaviors. While the client and server strive to manage complexity, the network generates it. Our model assumes the use of Internet Protocol, which it represents as having a per-packet failure rate equal to the assigned value for network congestion. The individual failures in turn affect our implementation of TCP streaming. Many streaming protocols use UDP rather than TCP, and are customized at the transport level; our choice of TCP/IP was based on its soundness and well-understood (but complex) behavior under load.

Each time it is loaded after a period of inactivity, the network immediately sends an initial window two packets wide. After this it will typically grind to a halt (given the bandwidth-delay product values at which we're operating) until both packets are acknowledged; subsequent windows may slide open wider as they receive acknowledgements in a steady stream. A dropped packet is represented as the receipt of a NAK instead of ACK, and results in a resend (and often in termination of the current window). This captures the essence of TCP slowstart and congestion avoidance behavior. Under optimal conditions, the protocol overhead is very small. Absent the use of bitrate peeling, however, congestive loading quickly leads to unacceptable quality of service, even when the leftover bandwidth exceeds the maximum bitrate of the audio stream.

4 Implementation

The first nontrivial peeling strategy we implemented attempted to bring the size of the network queue to a steady state. It did this by calculating the change in the queue size since the last frame was transmitted and adjusting the number of layers sent such that the queue size with the new layers added would be the same as the queue size when the previous set of layers had been

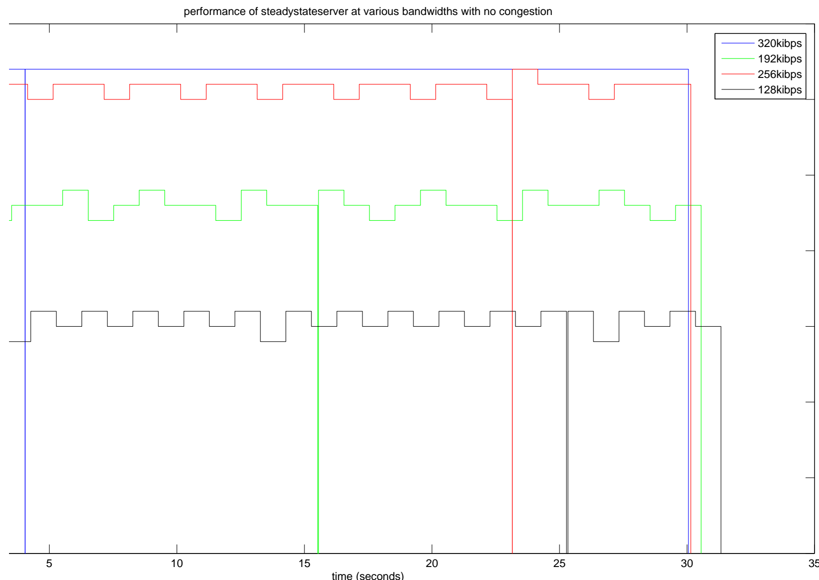

Figure 1:

added. This may also be seen as a form of derivative control. At the start of the stream, since there is no information about the previous queue size, the server simply attempts to send all layers in the frame. This delays the start of playback somewhat but allows the server to make a good determination of the performance of the network.

Figure 1 shows the performance of the steady state server, at several bandwidths, with no congestion. Note that the performance, measured by the number of layers included in each frame, is close to constant over time, indicating that the server found the maximum useful network bandwidth. On the other hand, Figure 2 shows the performance of the steady state server, again at several bandwidths, given 10% network congestion. Note that in this case the performance varies more over time and the dropouts (visible as brief downward spikes to zero) are more frequent.

Although the "steady state" server performed well when bandwidth was the only limiting factor, the presence of congestion severely harmed its performance. Our first attempt to compensate for congestion was what we call the "linear" (proportional) server strategy. In this strategy, the server determines how close to full the network queue is and scales the number of layers sent by the fraction of available queue space. While we had hoped that this strategy would improve performance under congested conditions without causing a significant decrease in performance when only bandwidth was a limiting condition, we found that it failed to achieve either goal. In fact, Figure 3 shows that even under no congestion, the linear server suffered frequent dropouts and tended to oscillate between fairly high and fairly low qualities, only beginning to converge in the last few seconds of the simulation. Under 10% congestion (see Figure 4, the linear server behaved even more erratically than the steady state server did, with frequent (and occasionally long) dropouts.

In an attempt to improve the performance of the linear server strategy, we tested several modifica-

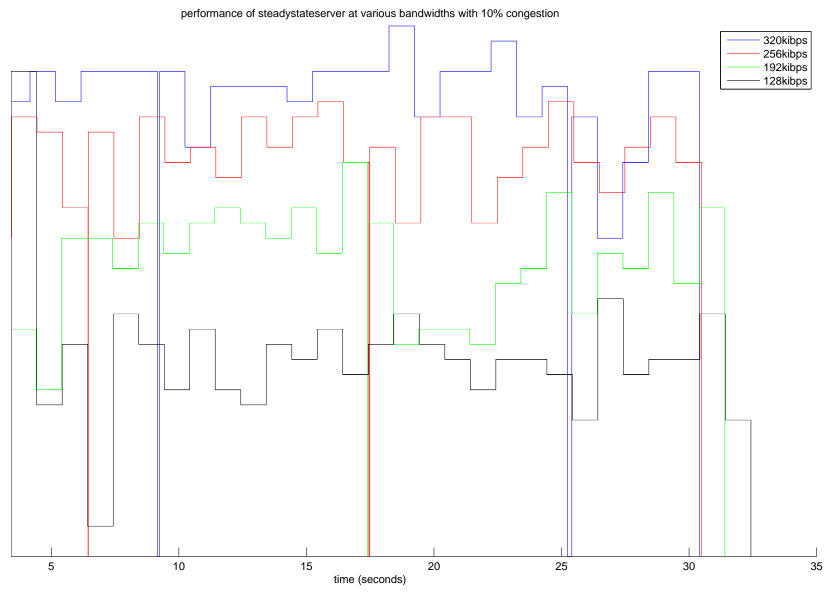

Figure 2:

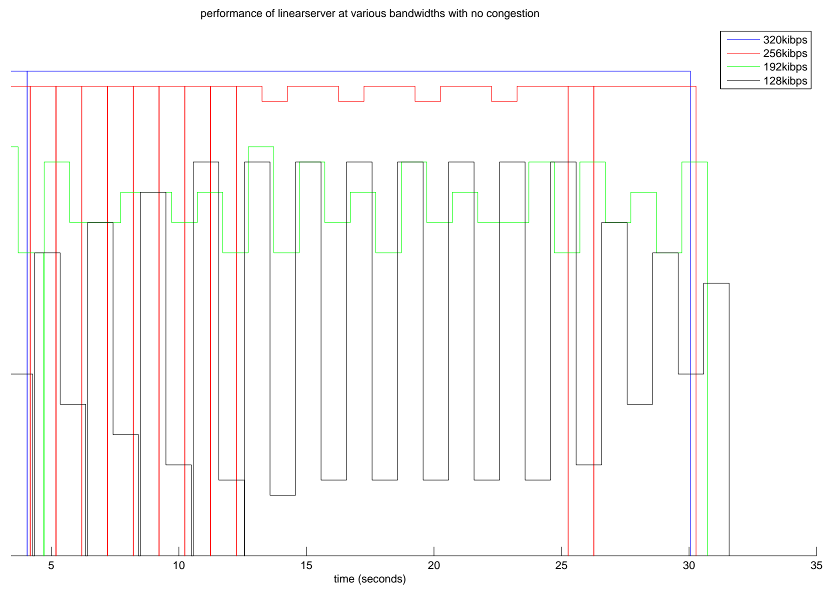

Figure 3:

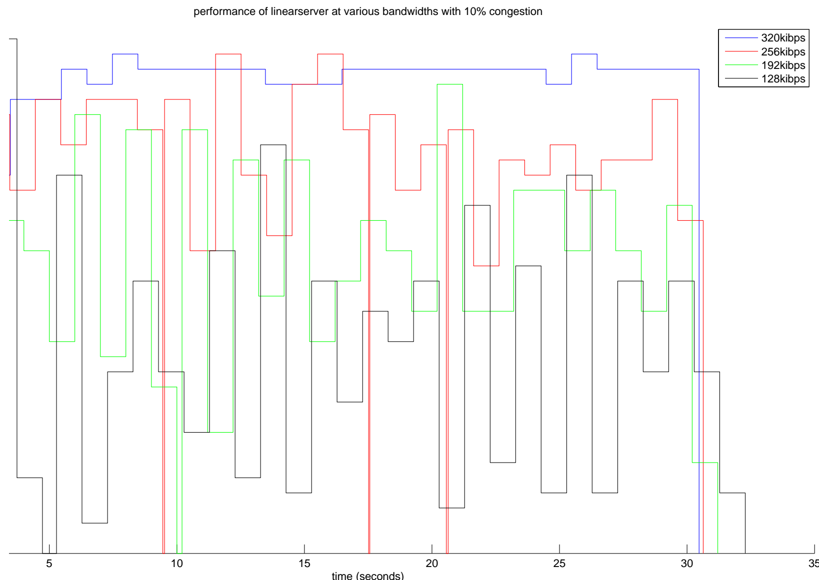

Figure 4:

tions such as scaling factors and raising the queue fullness to various powers. These measures were intended to make the server more responsive to changes in the network queue, but in actuality they failed to significantly improve on the initial linear strategy.

After unsuccessfully attempting to tweak the linear strategy, we more carefully considered the characteristics we would want the server's response to have. This analysis convinced us that a sigmoid function (similar to the cumulative normal distribution) of the queue fullness would behave in the desired manner. Initial results seemed promising; but when we tested the sigmoid strategy against others, we found that it did not result in the desired improvement. Again, we tested some variations in the sigmoid strategy, but found no success.

Even under congestion-free conditions (as shown in Figure 5), the sigmoid strategy suffered frequent slight dropouts. While it converged fairly quickly, the presence of continued dropouts, as well as the length of some dropouts, is troubling. Strangely, Figure 6 shows that the sigmoid strategy performs much better on a network with 10% congestion than on an uncongested network, especially with respect to dropouts. However, the performance does become very erratic in the presence of high congestion.

5 Results

We compared the strategies in a severely bandwidth-limited situation with no congestion to determine which strategy was best at compensating for low bandwidth. As Figure 7 shows, all strategies displayed some convergence towards the maximum quality sustainable given the available band-

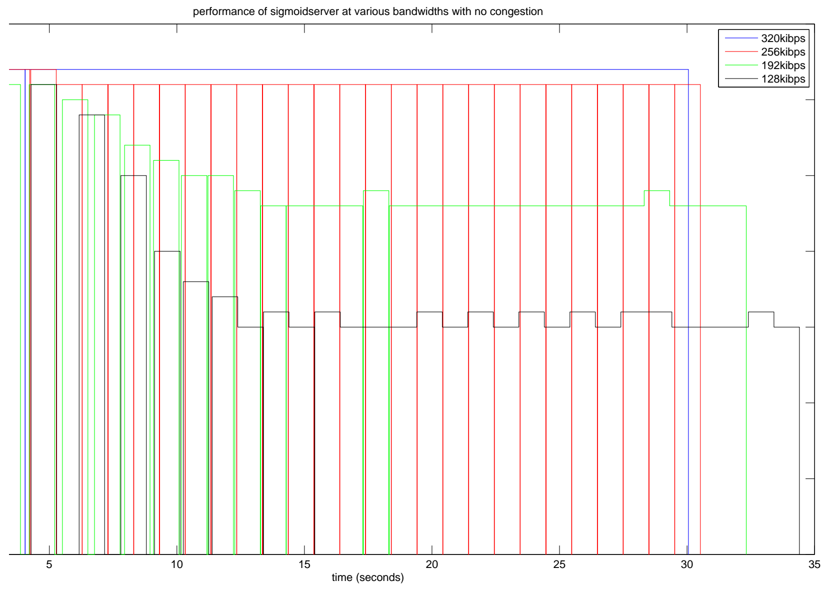

Figure 5:

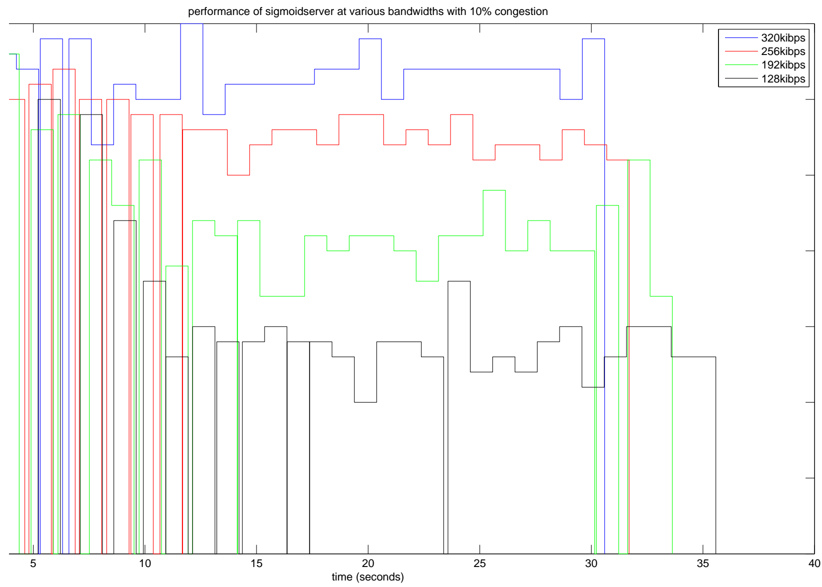

Figure 6:

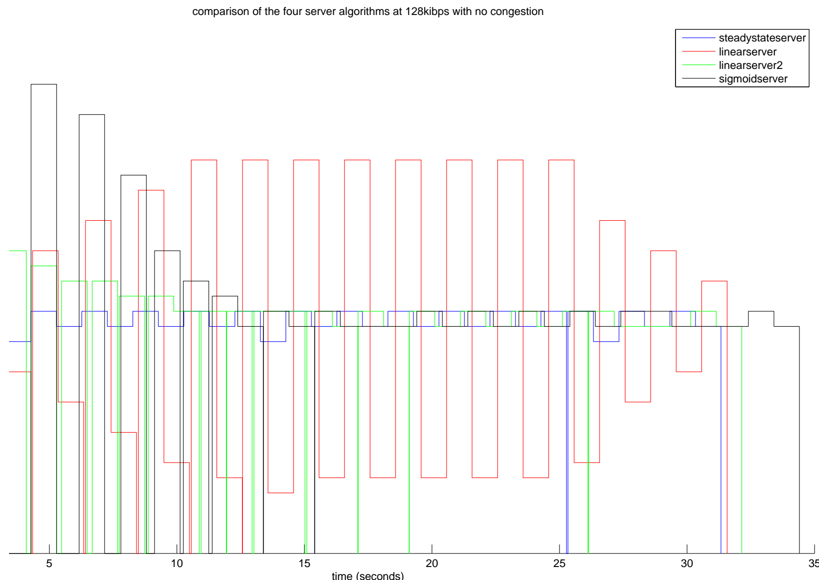

Figure 7: Comparison under bandwidth-limited conditions

width by the end of the simulation. However, the steady-state strategy converged the fastest and suffered the fewest dropouts. Surprisingly, the modified linear strategy converged second-most-quickly, though it did have frequent dropouts, while the original linear strategy was the slowest to converge but had fewer dropouts. The sigmoid strategy recorded middle-of-the-road performance, converging shortly after the modified linear strategy but with fewer dropouts.

In the presence of congestion, the trends previously observed become far less relevant. Given only 10% congestion with a bandwidth of 192 kbps, as shown in Figure 8, all four strategies behaved erratically and suffered fairly frequent dropouts. When congestion was increased to 20% (and bandwidth was held constant), only the steady state strategy and the linear strategy were worth evaluating. Even these (shown in Figure 9) had large fluctuations in quality, though the dropouts experienced here were not as frequent as those at 10% congestion.

6 Conclusions

Of the strategies considered in this paper, we find that the steady-state strategy yields the best performance. However, we were only able to achieve satisfactory performance in a congestion-free network. None of the techniques we attempted to use to compensate for congestion worked, and we believe this problem is far more complicated than it initially seems. Given a choice of functional congestion compensation methods, however, we would expect the simplest method to yield the best results based on our observations. Also, the relative success of the steady-state algorithm suggests some form of direct PID (proportional, integral, derivative) control may be appropriate where either bandwidth or congestion may be the limiting factor.

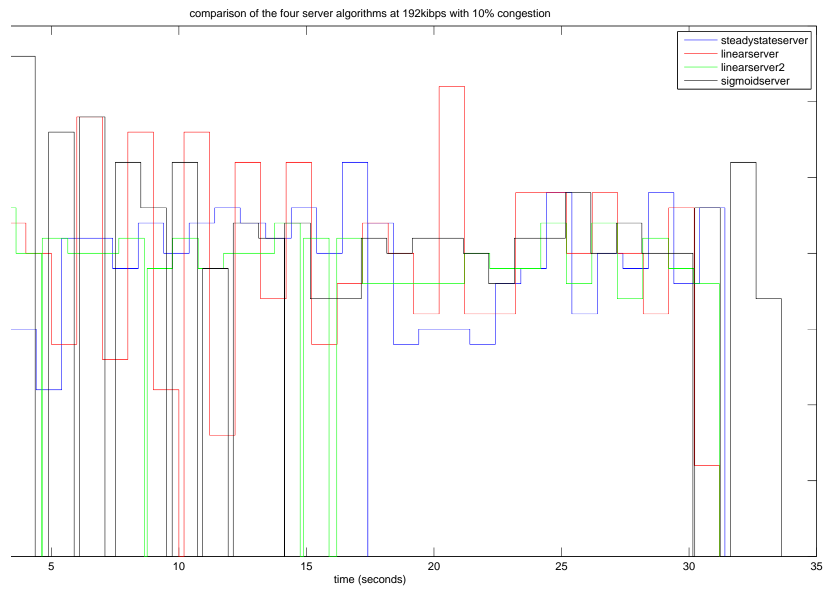

Figure 8:

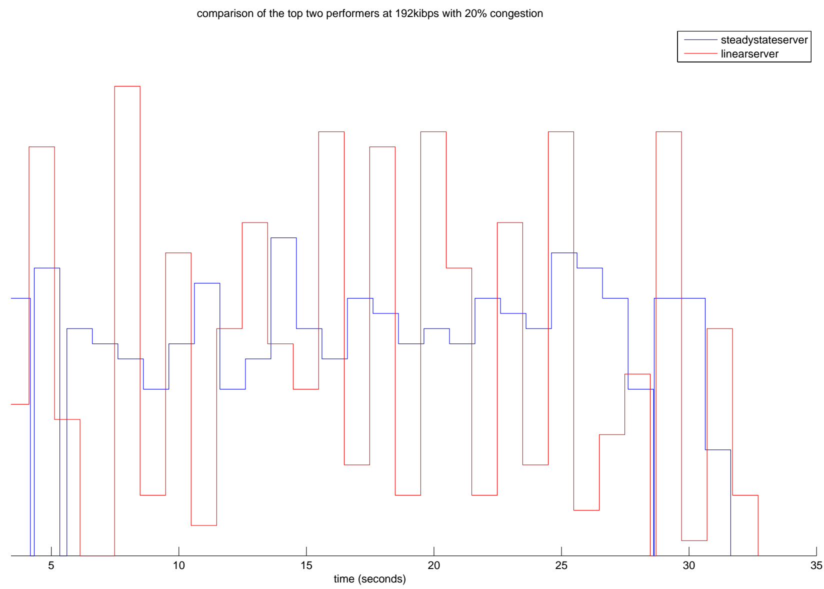

Figure 9:

7 Related Work

We are sure that much research into streaming media has been conducted over the years by the various companies that produce commercial streaming solutions, such as RealMedia, Apple's Quicktime, and Microsoft's Windows Media. However, this research is not available to the public because it is the basis for much of the perceived competitive advantage of each of these products. On the other hand, the developers of the open-source Ogg codecs have published considerable analysis of the potential benefits of bit-rate peeling, and it is their work that provides the inspiration for this paper.

8 Future Work

There is clearly much work still to be done on this topic. Given that we were unable to create a server strategy that successfully compensated for the effects of a congested network, the next obvious step is to research strategies for this. Naturally, it would also be necessary to check that a strategy still performed well under bandwidth-limiting.

9 Acknowledgments

We would like to thank Allen Downey for his help, encouragement, and patience in the completion of this paper. We are also indebted to Prof. Downey for the inspiration provided by his research on TCP/IP and by the source code to his discrete event simulator.