

# Audio On Demand extensions to Jungle Monkey

David Helder (dhelder@umich.edu)  
EECS 571 Project Report  
April 20, 1999

## Introduction

Video On Demand (VOD) has received quite a bit of attention from the research community in the past few years. However, on the Internet, audio is more common than video as seen in the rise of streaming audio applications (RealAudio [RA], Radio Moi [Moi], ShoutCast [SC], Spinner [Spin]) and the popularity of purchasing and exchanging MPEG Layer 3 format audio files (MP3.com [MP3.com], Lycos's MP3 search engine [Lycos]). For this reason, Audio On Demand (AOD) deserves some attention as well.

This paper describes the design and implement of AOD in Jungle Monkey, a bulk file transfer application based on the publish-subscribe model. Users can request high-quality audio streams from one of multiple servers and immediately receive a scheduling and QOS guarantee from a server. In addition, these guarantees are given for non-streamed files.

## Audio-On-Demand

Several VOD systems have been developed and VOD-specific scheduling algorithms proposed. However, AOD differs from VOD in several significant ways which should be considered in an AOD system:

- **Less bandwidth required.** High quality audio can be transferred using little bandwidth, while video still requires high bandwidth for even modest quality.
- **Audio files are short.** An audio file is often short in length and thus small in size. For instance, a song or a news clip may only be 3 minutes long. In contrast, a video may be an order of magnitude longer and larger such as in the case of movies. Another implication of the small size is that VCR functionality, a common feature in VOD systems, can be easily implemented using a buffer. (Though if audio files were large, such as an entire album of music, then VCR functionality would make more sense.)
- **Users tend to batch downloads.** Users often listen to several audio files, one after another. Because of this and because files are smaller, files are scheduled more frequently.
- **Users tend not to pay full attention to audio.** Users often listen to audio in the background, whereas users usually pay full attention to video. Often the user would be happy to listen to any song by a particular artist or in a particular genre.
- **Users demand high quality audio for music.** The rise of MP3 and other high quality audio formats show that when presented the trade-off between quality and size/bandwidth for music, users will choose quality.

Most VOD algorithms assume that once the user selects a video for playback, the server does not respond until it can playback the video. Thus one metric used to evaluate a VOD scheduling algorithm is number of reneges or defections – that is, the number of users who abort the request before the video is shown. This assumption has the nice property of allowing the server to make

decisions based on queue length for a particular video. On the other hand, it may be the cause of many defections – if the user does not know when the video will be shown, he may become frustrated and abort the request.

We have designed a system that schedules files and streams transfers as requests are received yet uses multiple servers efficiently. By scheduling on-the-fly and giving the user near immediate feedback, we hope to minimize defections. We now discuss the application in which we have implemented our design.

## **Jungle Monkey**

Jungle Monkey (JM) is a bulk file transfer application based on a publish-subscribe model written by the author. JM presents the user with a list of channels they can subscribe to. On each channel, users can request files offered for download and offer files for others to download – the client and server are integrated. Users can also create new channels. JM uses IP multicast for all networking. Files transfers use Reed-Solomon FEC [Riz] to provide reliable transfer. An alpha release of JM has been made under the GNU General Public License.

The original version of JM did no scheduling at all. When a server received a request, it immediately transferred the requested file. It ignored the amount of bandwidth being used and the transfer rate control mechanism was incorrect.

JM also has additional constraint that queue lengths are unknown. One optimization in JM is that a client won't request a file another client has just requested, thus a server does not know how many clients want a particular file. Many VOD scheduling algorithms depend on knowing queue lengths (MFQ [Agg96], MQL [Dan94]).

We modify JM to support AOD. This work includes adding streaming audio, a bandwidth scheduler, and more effective support for load balancing across multiple servers as well as some other minor necessary modifications.

## **Design and Implementation**

### **Streaming audio**

Several audio formats and playback utilities were evaluated to provide the audio streaming. The goal was to find something that could be integrated easily into JM – designing a new audio codec was beyond the scope of the project – as well as providing high quality audio so that people would want to use it. The audio playback applications examined included Real Audio, VAT/RAT, Net-Streamer, and several MP3 players. Most were quickly rejected because the source code or necessary programs were not freely available, the available source code could not be easily integrated into JM, or because the program was no longer maintained. Ultimately, I chose the audio format MP3 and the audio player *mpg123*.

The MPEG layer 3 audio format provides near-CD quality sound in very little space (about 1 megabyte for 1 minute of audio) [Fraunhofer]. (This encoding is popularly known as “MP3” and

the encoded files as “MP3’s”.) MP3 is fast becoming the de facto standard for music distribution on the Internet.

However, MP3 has several disadvantages. Though MP3’s provide good compression, streaming an average MP3 requires about 128 kilobits per second of bandwidth. MP3s are difficult to break up into atomic packets, which is required for Application Level Framing [Cla] based protocols like the Real Time Protocol [Sch]. This is because an MP3 is made up of several thousand frames, each frame dependent on previous ones. Because of the dependency, if a single frame is lost, the quality of playback may suffer for several seconds. Although RTP can be used to stream MP3’s, one implementor reports that if a more than 2% of the packets are lost, the content is “impossible to listen to” [mIR]. Unfortunately, packet loss this high is common over the Internet.

Another desirable property that MP3 does not have (as far as we can tell) is the ability to easily vary the bit-rate in real time. MP3 encoding is very computationally intensive and can be done in real-time only on the fastest computers. There may be a way to re-encode the MP3 on-the-fly, but we did not find a way.

Despite its disadvantages, MP3 was chosen. In order to make it “possible to listen to” over the Internet, it was necessary to write a reliable transport protocol, which is described in the next section.

For playback, JM uses the application mpg123. The advantage of mpg123 is that an MP3 can be piped to it’s *stdin*. Integrating this into JM required only a few lines of code. Support for other audio players will be added in the future.

### **Reliable transport**

MP3s are transferred using a resend-request-based reliable multicast protocol called the Jungle Monkey Reliable Multicast Protocol (JRMP). This protocol makes two important assumptions: 1) there is only one sender per channel and 2) the stream is sent at a constant rate. The advantage of the second assumption is that heartbeats are not necessary for clients to determine whether they have missed a packet. The disadvantage is congestion control cannot be provided by varying the bit-rate. Another issue is that the audio received is immediately played back – that is, there are soft real-time deadlines. We will now describe how the sender and receiver work and then discuss these issues.

There is one sender and multiple receivers. The sender sends packets to the multicast channel at a constant rate. It also listens on the channel for resend-requests. If a request is received, the requested packet is sent to the channel immediately. When the sender finishes sending the file, it continues listening on the channel for resend-requests. If the sender has finished transferring the file and hasn’t received a request for some time, it assumes all receivers have received the complete file and exits.

A receiver listens on the channel for packets. It keeps track of the sequence number of the next packet expected. When it receives a packet, it checks the sequence number to see if it is the expected packet. If the packet was not expected, it saves the packet in an early packet queue. Oth-

erwise, if it was expected, it writes the packet out and any following packets from the early packet queue.

Receivers time-out when they have not received the expected packet in an amount of time determined by the transfer rate. At the time-out, the receiver sends a packet to the channel requesting the missing packet. The receiver may send several requests. If it does not receive the packet after a set number of requests, it assumes the sender has failed and quits.

This protocol does not provide any sort of congestion control. Congestion control would require lowering the bandwidth, which would effectively slow the stream and receiver would miss its playback deadlines. There are several possible ways to work around this constraint. One is to simply provide congestion control in the traditional way and use buffering to minimize the impact on playback quality. This is effectively what TCP-based streaming implementations like ShoutCast do. Another solution would be to degrade the quality of the stream (e.g., send it at 20 kbs until the congestion has passed). As discussed in the previous section, this is not currently possible.

One difficulty encountered in implementing the timers for time-outs was overcoming the poor clock resolution. In Linux, the clock resolution is only 10 ms. If a thread is told to sleep for 6 milliseconds, it will actually sleep for 10 or 20 milliseconds (or even longer). One way to work around this is to schedule events based on absolute time, not relative time. Sleeping until a specific time performs better on average than sleeping a set amount of time. Periodic events (such as the send packet event for the sender) do not occur exactly at their scheduled time, but are close enough.

### **Outgoing packets**

In the original Jungle Monkey, each channel had a thread that sent a packet to the channel every three seconds. That packet may be an offer, request, or transfer announcement. The problem with this approach is that it does not scale well – the amount of bandwidth used would be proportional to the number of clients requesting and servers offering.

To remedy this, when the thread is to send a packet, it checks to see if an offer or request has been received recently (“recently” is a function of the number of requests or offers). If so, it suppresses the send. It is then possible to show that the maximum amount of bandwidth each channel can use is about 2 kbs. Suppressing transfer announcements is more subtle and is discussed below.

### **Scheduler**

In the original implementation of Jungle Monkey, a server would transfer any file requested immediately. If the server received a thousand requests, it would immediately start a thousand transfers. Clearly this is unrealistic – the server must limit the amount of bandwidth used. To do this, we’ve chosen to use a bandwidth scheduler (or admission controller) to schedule transfers based on bandwidth. The scheduler is part of JM. This section describes the algorithm used and a more complex algorithm that was not used.

When a request for a file (or stream) is received, the server checks to see if the file has been scheduled for transfer. If not, it uses the bandwidth scheduler to determine a transfer time. The bandwidth scheduler maintains a list of critical points, each which corresponds to the beginning or end

of one or more transfers. Each critical point is represented by a structure that contains the list of files being transferred at that time and the amount of bandwidth used.

To schedule a file, the scheduler transverses the list of critical points looking for one that has sufficient bandwidth available. When it finds one, it checks all critical points following it that occur during the transfer. If enough bandwidth is available, the file is scheduled and the critical points updated. We are guaranteed to be able to schedule the file (assuming the bandwidth required does not exceed the total bandwidth available) because the last critical point uses no bandwidth since it corresponds with the end of the transfer of the last file.

A more complex scheduler was designed and implemented, but not used. This scheduler also scheduled bandwidth used by files and streams requested by the user. If there was not enough bandwidth available for a requested file or stream, the scheduler would degrade the amount of bandwidth used for file sends. This is possible because files do not have a minimum bandwidth requirement. For example, if the user requested an audio stream while it was sending several files, the bandwidth of the files was lowered until enough bandwidth was available for the stream. The scheduler lowered the bandwidth of the least requested file first.

One difficulty encountered (and the reason this scheduler was not used) is that if a file's bandwidth is lowered, then it's transfer time is extended. This would increase the bandwidth of the following critical points. In order to guarantee that the maximum bandwidth is not exceeded either 1) we must allow transfer start times to be adjusted or 2) we must allow file transfer bandwidth to be arbitrarily small. The first solution would contradict our original goal of providing the user with transfer start time guarantees. The second solution would make it impossible for a client to detect if a server failed.

## **Load balancing**

Load balancing was added to use the network efficiently in the presence of multiple servers. The difficulty in many load balancing schemes is that the server (or the CPU) requires some knowledge (or estimate) of the state or capabilities of other servers. For scalability and simplicity reasons, we'd like to be able to balance the load without requiring a server to have any knowledge of any of the servers. Moreover, we want to do so efficiently – that is, our solution should not delay the transfer of a file or require significant communication. We also assume that when a server announces a transfer, it cannot later cancel it.

First we will examine two naive approaches. The first is the original implementation; the second an optimization to the original implementation that is similar to the solution used. In the original implementation, when a server received a request it would immediately schedule a transfer. Each server would send a message to the channel announcing the transfer and then start sending the file on another channel. Thus, if there were a million servers and one client made a request, there would be a million transfers but only one would be useful.

The second approach is to optimize this by having the server suppress a transfer if it receives a transfer announcement from another server. The problem with this is that since a server sends a transfer announcement immediately after receiving a request, most servers will send announcements almost simultaneously. This is unlikely to help servers that are close and receive requests at

the same time. This is exactly the sort of situation where we'd want to reduce the number of transfers so that we would lower the chance of congestion.

The approach finally used is similar to the second approach. The key is instead of sending the transfer announcement immediately, send it after a delay based on the time the transfer will take place. If a server can perform the transfer immediately, it sends the announcement immediately, otherwise it waits a few seconds before sending the announcement. If a server suppresses its own transfer announcement because of this, most likely it is because another server can transfer the file sooner. Thus we achieve effective and efficient load balancing. This works for the same reason VTCSMA works – the channel like is a shared bus and the transfer time a priority.

In our implementation, if the server can transfer the file in the next 30 seconds, it sends an announcement at a random time during the next 10 seconds. Otherwise, it sends an announcement at a random time after 10 seconds and before 20 seconds. Thus if the file can be transferred, the user will know when the file will be transferred within 20 seconds.

## **Evaluation**

The AOD components of JM were tested as they were implemented. Flags and variables were set to limit the amount of bandwidth available and the packet drop rate. As (and if) JM becomes more popular, more rigorous measurements of the effectiveness of the modifications will be made.

## **Related Work**

multicast Internet Radio [mIR] is an AOD-like application that streams MP3s using RTP on top of IP multicast. There are a limited number of songs a server can send at once. Users vote for songs they'd like to here and the song with the most votes is sent first.

There are many radio-like audio applications. liveCaster [live] uses the same technology as mIR, but does not allow the user to choose what is played next. Netstreamer [NS] is similar but uses a lower quality audio format. Shoutcast [SC] streams MP3s using HTTP on top of TCP. It limits the amount of bandwidth used by limiting the number of client connections.

RealAudio's RealSystem G2 uses a proprietary format to provide streamed audio. It can vary the bitrate based on bandwidth available and network conditions. G2 also supports IP multicast.

## **Conclusion**

We have designed and implemented an AOD system in Jungle Monkey. Users can request high-quality audio streams from one of multiple servers and immediately receive a scheduling and QOS guarantee from a server. Multiple servers can balance their load efficiently and effectively.

## References

- [Agg] Aggarwal, C., J. Wolf, and P. Yu. On optimal batching policies for video-on-demand storage servers. Proc. of IEEE ICMS, Japan, June 1996. pp. 253-258.
- [Cla] Clark, D., D. Tennenhouse. Architectural considerations for a new generation of protocols. Proc of SIGCOMM, 1990. pp. 201-208.
- [Dan] Dan, A., D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. Proc. of 2nd ACM Multimedia Conference, October 1994. pp. 15-24.
- [Fin] Finlayson, Ross. A more loss-tolerant RTP payload format for MP3 audio. <http://www.live.com/rtp-mp3.txt>
- [Fraunhofer] MPEG Layer-3 Info homepage. <http://www.iis.fhg.de/amm/techinf/layer3/index.html>
- [Live] Live.Com homepage. <http://www.live.com>
- [Lycos] Lycos MP3 Search Engine. <http://mp3.lycos.com/>
- [Rol] Parviainen, Roland. multicast Internet Radio. // <http://www.cdt.luth.se/~rolle/mIR/>
- [Moi] Radio Moi homepage. <http://www.musicmusicmusic.com/>
- [MP3.com] MP3.com homepage. <http://www.mp3.com>
- [mpg123] mpg123 homepage. <http://dorifer.heim3.tu-clausthal.de/~olli/mpg123/>
- [NS] Netstreamer homepage. <http://flits102-126.flits.rug.nl/~rolf/NetStreamer.html>
- [RA] RealAudio. <http://www.realaudio.com/>
- [Riz] Rizzo, Luigi. Effective erasure codes for reliable computer communication protocols. ACM Computer Communication Review, 27(2), April 1997. pp. 24-36.
- [SC] ShoutCast. <http://www.shoutcast.com/>
- [Sch] Schulzrinne, H., S. Casner, R. Frederick, V. Jacobson. RTP: A transport protocol for real-time applications. IETF, RFC 1889, 1996.
- [Spin] Spinner. <http://www.spinner.com/>
- [Sta] Stankovic, J., Ramamritham, K., Cheng, S. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. IEEE Transactions on Computers, 34(12), December 1998. pp. 1130-1143.