Audio Streams Merging Over ALMI

Christopher J. Dunkle, Zhen Zhang, Sherlia Y. Shi, Zongming Fei

Department of Computer Science
University of Kentucky
301 Rose Street, 2nd floor
Lexington, KY 40506-0495, U.S.A
cjdunk0@cslab.uky.edu, {zhen, sherlia, fei}@netlab.uky.edu

Abstract

A variety of video/audio applications based on multicast have been developed to facilitate group communications over the Internet. One of the critical issues for these applications is how to keep the bandwidth consumption low while ensure acceptable performance. The applications that allow multiple senders make this problem even more challenging, because different nodes may send data simultaneously. In this paper, we focus on a multi-sender audio conferencing application designed on top of ALMI, an Application Level Multicast Infrastructure. We present a method to improve the efficiency of the application by combining multiple audio streams from different senders into a single stream, which is then transmitted to neighboring nodes along the multicast tree. An evaluation based on a JAVA implementation of the application demonstrates the improvement brought by the method.

1. Introduction

Today's Internet is a collection of a large number of end hosts connected to each other via a variety of networks. The simplest form of data transfer over the Internet consists of one-to-one communication (or unicast), but many of today's emerging applications have a requirement that goes beyond the standard one-to-one communication. Applications such as video conferencing, corporate communications, distance learning, distribution of software, stock quotes, and news need to support one-to-many and many-to-many communications. One of the proposals was native IP multicast, which duplicates packets at each router along paths from the source to each destination [2][6]. Unfortunately, IP multicast is not widely deployed in the Internet for various reasons [5]. We need to resort to other approaches to support group communications. Application layer multicast [7][8] has been studied in recent years as an alternative approach to IP Multicast. It creates a virtual overlay network topology that is built on the underlying TCP/IP network for data transfer, and lets end hosts perform the packet duplication rather than routers. This may require more overall bandwidth than IP multicast because duplicate packets may travel the same physical links (Figure 1), but it provides a simple and inexpensive alternative of providing multicast-like activity on top of the already existing IP without any modifications to any routers. This also allows for more flexibility in the control of how multicast packets are forwarded, and how the overlay network topology is constructed.

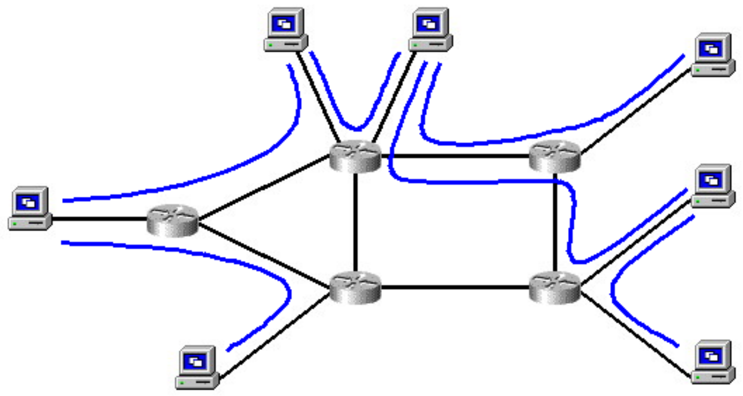

Figure 1. An overlay topology for Application Layer Multicast

One of the simple multicast applications is audio conferencing, in which a sender could transmit audio data to multiple receivers simultaneously. In this paper, we are particularly interested in the audio conferencing application that allows multiple senders to transfer audio data at the same time. This can happen when multiple participants at an audio conference talk at the same time. Another scenario is that multiple musicians play different instruments at different locations and want to see the collective effects. The simple solution is to treat each sender independently and the audio stream from each sender is multicast to all receivers. The problem with the approach is that the bandwidth usage will be proportional to the number of senders. Those receivers with low bandwidth connections to the Internet will not be able to receive all the streams, or have to receive them with poor quality. In this paper, we propose an audio merging technique to improve the efficiency of the application by combining multiple audio streams from different senders into a single stream, which is then transmitted to neighboring nodes along the multicast tree. The implementation can deal with digital audio encoded in PCM and μ-law formats. Audio data are transmitted using real-time transport protocol (RTP) [4] to enable the correct merging at the intermediate nodes. We design and implement the application on the top of an existing implementation of application layer multicast called Application Level Multicast Infrastructure (ALMI) [1].

The application is designed so that the merging module can be toggled on and off, allowing us to analyze the performance differences between merging the audio at intermediate nodes and simply forwarding streams without merging. Experiment results show that if multiple senders are trying to multicast audio data simultaneously, our method will reduce the multicast traffic greatly and lead to a more efficient utilization of network bandwidth.

The rest of the paper is organized as follows. Section 2 reviews the background and related work. Our method of audio streams merging over ALMI is presented in section 3. Section 4 describes the implementation. Section 5 analyzes the performance. Finally, the paper is concluded in section 6.

2. Background and Related Work

ALMI is a middleware of application level multicast in JAVA with many-to-many semantics. A special node called Controller is responsible for creating multicast trees. When a node joins a multicast session, it contacts the Controller corresponding to the multicast session it is interested in. The Controller creates a minimum spanning tree based on application specific metrics. Each node in the tree communicates directly only with its neighbors (parent or children nodes) along the spanning tree. Update messages are forwarded to each node from the Controller as hosts join or leave. A Controller node may be capable of handling multiple sessions at the same time. Also the Controller is not part of the spanning tree, and the IP address and the port of the Controller needs to be received though out-of-band methods.

The basic application interface (API) of ALMI is a very simple set of methods. There are methods to join and leave a multicast session as well as methods to send and receive data. When a node receives data, it will automatically send to all its neighbors, except the node from which the data are received. Similarly, the receive method simply returns the packet of data at the head of a queue where incoming data are stored.

Before the surfacing of application level multicast, early implementations of audio conferencing are usually based on IP multicast. These tools include Robust Audio Tool (RAT) [10] and vat [11]. Unfortunately, since IP multicast is not widely enabled on the Internet, their applications are restricted to those sites connected by MBONE [3].

3. Multi-sender Audio Conferencing over ALMI with Audio Streams Merging

In this section we present the audio streams merging method in multi-sender audio conferencing applications. In these applications, multiple senders will be present in the multicast tree and propagate their audio data simultaneously. Each end host that receives the audio data has to be equipped with output hardware that permits receiving multiple audio streams at the same time.

With this type of application in mind, we will show that the merging technique can improve the efficiency and reduce the bandwidth usage. At each intermediate node in the multicast tree, multiple audio streams will be merged, if possible, before re-distributed to all other neighbors. As an example (Figure 2), suppose Node-3 has 3 neighbors, and is receiving audio streams from Node-1 and Node-2. These two streams are duplicated and re-sent to Node-4. What we observed is that two audio streams sent to Node-4 are very likely to contain data that will be played out at Node-4 at the same time. It is possible to merge those two audio streams into a single stream before re-sending them to Node-4. By doing this, the amount of bandwidth used is decreased to approximately half of bandwidth used in the original scheme.

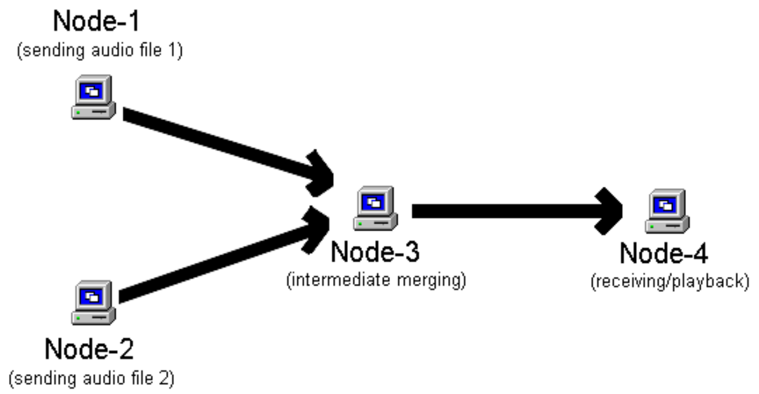

Figure 2. Example Topology where Audio Merging Takes Place

3.1. Audio Merging

Before we proceed, we would like to show that it is possible and practical to merge multiple audio streams into one stream, and that the amount of bandwidth required is decreased after merging.

Digital audio can be recorded from an analog signal by using an Analog-to-Digital Converter. This converter samples the value of the amplitude of the analog signal's waveform at very small intervals and encodes the value by means of Pulse Code Modulation (PCM). The number of samples taken per second is known as the sampling rate, and as an example, CD quality audio has a sampling rate of 44.1 kHz. Widely used sampling rates are 44.1 kHz, 22.05 kHz and 8 kHz. Also, multiple channels are allowed in the format, and generally 1 channel consists of mono audio and 2 channels consist of stereo audio. PCM data can be either signed or unsigned.

A lossy compressed audio format encoding known as μ-law is generally used for audio data with a sampling rate of 8 kHz and a bit-depth of 16-bits. The compression method allows 16-bit data to be stored as 8-bit data, and is useful for the transfer of data since only half the amount of bandwidth is required. In our implementation, we will focus on 16-bit mono audio data with a sampling rate of 8 kHz. Audio data that are transferred by using 8-bit μ-law encoding can be converted to signed PCM audio data for audio merging operations and for audio playback at end hosts.

Fortunately, merging PCM audio data is based on the fact that, in the real world, an analog waveform can be combined with another waveform to produce a resulting waveform by simply adding the two waveforms together. Amplitudes of opposite signs cause the waveform to cancel each other out at that position, while amplitudes of the same sign cause the amplitude of the resulting waveform to have larger amplitude. Similarly, this can be applied to digital audio. All sampled values of two (or more) sets of audio data can be added together at corresponding positions. The resulting values are seen as the sample value of the merged audio at that position (Figure 3). After adding the values together, it is possible that a sample value is beyond the bounds 16-bit signed PCM data could represent (Generally, 16-bit signed PCM data has a minimum of -32768 and a maximum of +32767). In this case, the sampled value needs to be truncated to be equal to the minimum or maximum value accordingly. The quality of the merged stream is acceptable and different streams can be distinguished unambiguously during the playback.

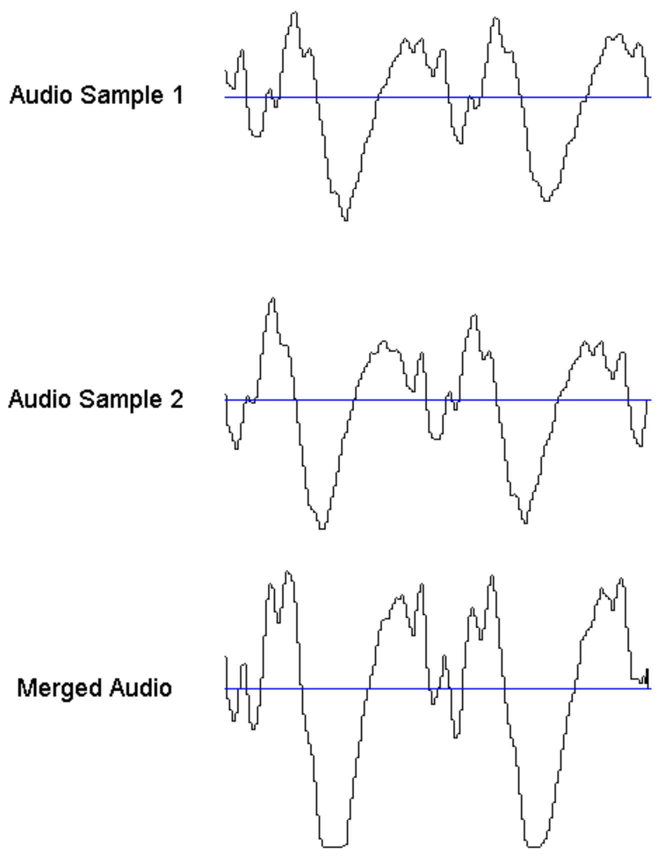

Figure 3. Merging of Two Sets of PCM Audio Data with Truncation

3.2. Timing of Audio Samples

In order to merge audio in the proper corresponding positions, we need a method of labeling the timing relationship of two audio streams. Since two different audio streams are completely independent of one another, we cannot simply flag pieces of audio data with sequence numbers, as there is no way to easily synchronize the sequence numbers used. We know that two pieces of audio data should be played at the same time if they were created or recorded at the same point of time. Thus, the easiest way to know if two pieces of audio data correspond to the same moment time is to associate a piece of audio data with a timestamp that marks when the beginning of that piece of audio data was created or recorded. When two audio streams are received at a particular node, we can compare the timestamps of the two pieces of audio and see if they overlap. If so, those two pieces of audio data could be merged. It is important to note that it is possible for them to be merged even if the timestamps are not equal. A second piece of audio data might be marked to begin at some point in the middle of the first piece of audio. In this case, the prior part of the first piece of audio data remains the same, the middle contains merged data from both audio streams, and then the remainder of the merged data contains the rest of the second piece of the audio data. Though this case will cause the required bandwidth usage to be more than that of a single stream, it is still more efficient than sending two completely independent audio streams.

To record each piece of audio data with a timestamp, we use the real-time transport protocol (RTP). An RTP header contains the timestamp when an audio packet is created according to local machine time. RTP allows for a synchronization source that would be used to ensure that all packets coming from different senders would have properly corresponding timestamps. Our application makes the assumption that all machines active in the multicast group have synchronized their local machine clocks through some out-of-band method.

4. Implementation

We take care of what data are forwarded to which node during distribution along the multicast tree. Each set of data forwarded to the neighbors depends on where the data are coming from. It is important to note that a stream only needs to be duplicated to all neighbors except the one where the data are received. If a neighbor sends a node some data, it is assumed that the sub-tree past that neighbor has already received or will receive the data. With the same example in Figure 2, Node-3 has three neighbors Node-1, Node-2 and Node-4. It may be receiving two streams, stream A from Node-1 and stream B from Node-2. Because stream A is received from Node-1, it only needs to be duplicated to Node-2 and Node-4. Similarly stream B received from Node-2 only needs to be duplicated to Node-1 and Node-4. In this example, Node-1 and Node-2 only receive one stream, so merging is not necessary. But Node-4 receives data from both streams A and B; so merging can be used to reduce the bandwidth usage. This example shows that all neighbors might have a different set of outgoing data, and this needs to be accounted for in an implementation of our application. Therefore, we need to keep track of which neighbors are present, and forward data to each neighbor as needed. The only exception to this is when audio data originates from the local end host, and the data would then need to be forwarded to all neighbors.

Another problem we need to take care of before sending data for playback is that audio data may be not fully merged by intermediate nodes along the multicast tree. In Figure 4, we illustrate the problem where merged audio data packets do not match due to a slight difference in the timestamp, which creates an offset in the merging. This offset would then introduce a new type of audio playback jitter with any audio data that immediately follows that merged audio data. Therefore, our implementation will first check if any more audio data merging is possible with following packets according to the timestamp, and then deliver the audio data for playback.

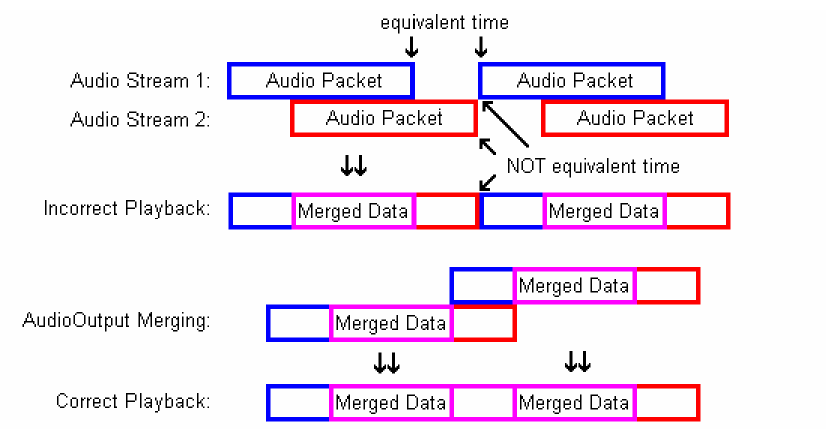

Figure 4. Received Audio Packets not fully merged requiring additional merging for playback

We now give some more details of the implementation in JAVA as shown in Figure 5. A group member begins by contacting the ALMI controller to find its place in the multicast spanning tree. It initializes and starts a thread called AudioMulticastThread that is responsible for forwarding and receiving audio packets. An audio packet is first converted to μ-law format, then prepended with an RTP header, encapsulated as an ALMI packet (AlmiPacket), and finally sent to all neighbors along the multicast tree. When a node receives a packet, an object named DirectedAlmiMember determines which neighbor the packet came from and passes it to AudioMulticastThread. This thread automatically sends the packet to the local application and puts copies of the packet into an outgoing queue for each neighbor, except the neighbor where the packet is received. AudioMulticastThread cycles through all queues for each neighbor, sorts all the packets by timestamp, and then checks each packet with the packet following it in the list to see if the timestamps overlap. If so, it merges those two packets and replaces the original ones. It ensures that all possible audio merging be appropriately carried out, and then forwards all of those packets to that neighbor. On receiving an audio packet, our application will insert the packet into a sorted queue according to the timestamp. Data will be extracted and merged again as necessary and are ready for playback.

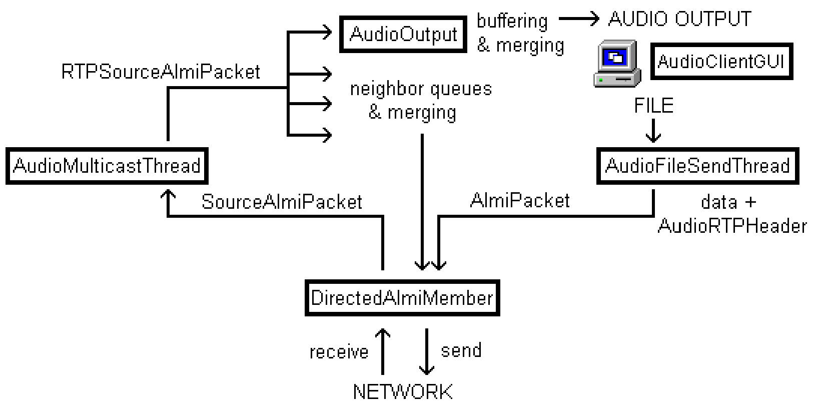

Figure 5. Basic Flow of the Application

5. Performance Analysis

The motivation behind this work is to reduce the bandwidth used by merging multiple audio streams into a single stream whenever possible. To show that the method can reduce the amount of data sent between nodes, we need to track the amount of data being sent and received over each link in the multicast tree. Since the topology of the ALMI overlay network is a minimum spanning tree, there are no loops present and there is no concern about data inadvertently crossing a link more than once.

A reasonable way to show that this application merges audio and reduces the bandwidth is to set up the multicast tree in a given topology, and test the amount of data being sent from a number of known sources. We will also want to keep the same topology and measure the amount of data being sent and received in the case when audio is not being merged. The comparison between the amount of data sent and received in both cases will be able to show us how efficient this application is at reducing data transfer due to merging of multiple audio streams.

For our performance analysis, we will use a setup of four different nodes as shown in Figure 2 above. Node-1 and Node-2 will be sending audio data from a different sound file, each transferring the data as a different stream. Node-3 in the middle will merge the incoming data as necessary. After merging data whenever possible, it will then forward the merged data to Node-4 designated as the receiving node which performs the local application playback of the audio data.

During our tests, Node-1 sends audio data from a 60 second sound clip stored in 8-bit μ-law 8 kHz mono format (*.AU) of approximately 469 KB. Node-2 sends audio data from a 48.7 second sound clip in 16-bit signed PCM 8 kHz mono format (*.WAV) of approximately 762KB. All data transferred between nodes is in μ-law format, so the 16-bit WAV file is converted to μ-law before data transfer. Thus it takes approximately half the size of the input file, about 381KB, during data transfer through the application.

For our base case test, we want to see how many bytes are sent to and received from each node when merging is not present. Table 1(a) shows the results for this base case test for each node's transfer amount. The 469KB audio file results in 526KB of data to be sent from Node-1 to Node-3. The extra data can be accounted for due to ALMI headers and RTP headers. The 762KB audio file (381KB of μ-law audio data) results in 427KB of data to be sent from Node-2 to Node-3. Node-3 thus receives 526KB + 469KB = 953KB of data and then has to forward the data to all other nodes, except where the data are received. The 526KB of data generated from the first audio file needs to be forwarded to Node-2 and Node-4 and the 427KB of data generated from the second audio file needs to be forwarded to Node-1 and Node-4. Thus, Node-3 needs to send out 526KB + 526KB + 427KB + 427KB = 1906KB. And since Node-4 is receiving the exact data from the two audio streams, it receives

526KB + 427KB = 953KB of data, since no merging is taking place at Node-3. Also, Node-4 has no other neighbors to forward the data to, and is not generating any data, so it sends zero bytes.

	KB Sent	KB Recv
Node-1	to Node-3: 526	427
Node-2	to Node-3: 427	526
Node-3	to Node-1: 427	953
	to Node-2: 526	
	to Node-4: 953	
Node-4	0	953

(a) without merging

	KB Sent	KB Recv
Node-1	to Node-3: 526	427
Node-2	to Node-3: 427	526
Node-3	to Node-1: 427	953
	to Node-2: 526	
	to Node-4: 580	
Node-4	0	580

(b) with merging

Table 1. Data sent and received from each node

Next, we examine the performance when audio merging takes effect. In our setup, merging will only take place at Node-3 because this is the only node that would be capable of forwarding data. All the other nodes either send or receive only since they only have one neighbor. Since the application does not guarantee when packets are received at a given node and different instances of merging could produce different results, we decided to run through this test a number of times to come up with an average amount of data transfer reduction. Table 1(b) shows the average amount of data sent and received from each node with the same topology as above. As we can see, the transfers between Node-1 and Node-2 through Node-3 do not change. This is because no merging could take place in these cases. All the data coming from Node-1 needs to be forwarded to Node-2, and nothing else, and the same is true in the opposite direction. Node-3 also receives the same amount of data because it is only receiving from Node-1 and Node-2. We will focus on the amount of data sent from Node-3 and received by Node-4. It is significantly reduced due to the audio merging. Since one audio file is larger than the other, the best-case scenario is when total merging is performed and the total data sent after merging would be equal to the larger of the two streams, and in this case is 526KB. The table shows that the average amount of data sent with merging is 580KB, which is only 54KB, or about 10% more than the best case. Also, from Table 1(a), sending data without merging would have required that 953KB of data be sent. This shows the application can reduce the amount of bandwidth used to about 60% when merging of audio data is enabled is for these files.

These tests have shown that the application can reduce the amount of data transferred between nodes when audio data merging is used. In this case, approximately a 40% reduction of data transfer was achieved. The amount of data reduction depends on how much data are being sent from each stream, and how much of the data overlap. It should be clear that a larger number of streams present at any given time would cause an even greater reduction due to more data being merged.

6. Conclusion

In this paper we have described an audio conferencing application that sends multiple audio streams over ALMI. In an attempt to decrease the bandwidth usage between nodes in the multicast spanning tree, we presented a method of merging multiple audio streams at intermediate nodes prior to forwarding the audio data to other nodes. We have presented a design, an implementation and a performance analysis for such an application that performs successful audio streams merging at intermediate nodes, along with proper playback of the merged audio streams at end hosts.

7. Reference

[1] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel, “ALMI: An Application Level Multicast Infrastructure”, In 3rd USENIX Symposium on Internet Technologies and Systems (USITS’01), San Francisco, CA, March 2001.
[2] F. Baker, “Distance Vector Multicast Routing Protocol – DVMRP”, RFC 1812, June 1995.
[3] H. Erikson, “MBONE: The Multicast Backbone”, Communication of the ACM, 54-60, August 1994.
[4] H. Schulzrinne, “RTP: A Transport Protocol for Real-Time Applications”, RFC 1889, January 1996.
[5] K. Almeroth, “The Evolution of Multicast: From the MBone to Inter-Domain Multicast to Internet2 Deployment”, IEEE Network Special Issue on Multicasting, January/February 2000.
[6] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. G. Liu, L. Wei, “The PIM architecture for wide-area multicast routing”, IEEE/ACM Transactions on Networking, vol. 4, no. 2, pp153-162, April 1996.
[7] Sherlia Shi and Jonathan S. Turner, “Routing in Overlay Multicast Networks”, IEEE INFOCOM, New York City, June 2002.
[8] Y. hua. Chu, S. G. Rao, and H. Zhang, “A Case for End System Multicast”, In Proc of ACM Sigmetrics, 1-12, June 2000.
[9] Y.hua Chu, S. G. Rao, S. Seshan, and H. Zhang. “Enabling Conferencing Applications on the Internet Using an Overlay Multicast Architecture”, In Proc. ACM SIGCOMM 2001, San Diago, CA, August 2001.
[10] Robust Audio Tool, http://www-mice.cs.ucl.ac.uk/multimedia/software/rat/
[11] LBNL, Audio Conferencing Tool (vat), http://www-nrg.ee.lbl.gov/vat/