AUDIO MULTICAST OVER APPLICATION LEVEL MULTICAST INFRASTRUCTURE

Masters Project Report
Feb, 2004

Submitted by
Srikrishna Putta

Advisor
Dr. Zongming Fei
Assistant Professor
Dept. of Computer Science
University of Kentucky
ACKNOWLEDGEMENTS

I sincerely thank Dr. Zongming Fei for his invaluable guidance and support through the course of my project. I also thank Dr. Manivannan and Dr. Jun Zhang for serving on my defense committee. My sincere thanks for the Department of computer science and University of Kentucky for providing me the resources for completing my project.
Table of Contents:

Overview

1. Introduction .................................................................4
2. Definitions .................................................................5
3. Related Work ...............................................................5
4. Requirements ...............................................................6
5. Application Level Multicast Infrastructure (ALMI) ...............7
6. Overview of Linux Sound Programming ............................8
7. Design
   7.1 Architecture of Application level multicast infrastructure
       .................................9
   7.2 Tree Construction ...................................................10
   7.3 Neighbor Management .............................................14
   7.4 Incremental Tree Update ..........................................15
   7.5 Data Transfer – RTP ..............................................16
   7.6 Audio Design .........................................................18
   7.7 Silence Detection ...................................................19
   7.8 GUI .................................................................19
8. Implementation details ..................................................20
   8.1 Application Layer Multicast
       8.1.1 Data structures .............................................23
       8.1.2 Functions .....................................................24
   8.2 Audio Conferencing
       8.2.1 Functions .....................................................28
9. Conclusion and Future work ...........................................28
10. Learning Outcome .......................................................28
11. References ..............................................................30

APPENDIX A - System Requirements ..................................31
1. **Introduction**

The IP multicast model allows scalable, efficient multiparty communication. However, it needs a fairly elaborate control support from network routers, membership management and multicast routing protocols. The deployment of IGMP and routing protocols requires substantial infrastructure modifications and complex modifications to IP router’s software.

Although people have deployed IP Multicast on the core routers and switches in the Internet recently, IP Multicast is still not widely available after being proposed over ten years. Also a large number of applications like audio/video conferencing usually contain a small number of group members and the groups are often created and destroyed relatively dynamically. There may be a very large number of such groups active at a given time. In such an environment, the benefits of IP multicast in terms of bandwidth efficiency and scalability are outweighed by the control complexity associated with group setup and maintenance cost.

Therefore due to the increasing number of such applications and a lack of proper deployment of IP multicast in all IP-based networks, there has been a trend of moving multicast support from the network layer into the application layer. There have been many studies to design, implement and evaluate various protocols for application layer multicast. An application layer multicast protocol builds an application-level network that connects members in a multicast group through unicast connections. Members in the multicast group then use this network for the delivery of multicast content.

There are many such application layer protocols like End system Multicast [4], Yallcast [5] etc. But as these protocols are fully distributed, they may still cause excessive control overheads and incur reliability problems, which are the same problems as in the current multicast protocols. A centralized control protocol such as Application level Multicast Infrastructure (ALMI), with careful design of redundancy can simplify the matter greatly and provide a more reliable mechanism to prevent tree partitions and routing loops. Though the centralized controller would constitute a single point of failure, we can overcome this by maintaining multiple back-up controllers.

Audio conferencing applications mainly use two techniques: Unicast Server and Multicast. Unicast Server refers to networking in which a server accepts connections from all the participants in the conference and sends the packets originated by one of those participants to all of the rest. Many of the current audio conferencing applications operate in this fashion. The main disadvantage is that all the receivers receive a separate data packet, which rapidly uses up the bandwidth. In multicast, the advantage is that only a single copy of data is sent by the source.
The most popular audio conferencing tools like RAT (robust audio tool), LBL’s VAT (Visual audio tool) require that all the systems support IP Multicast, and ideally, your network should be connected to the IP Multicast Backbone (MBONE).

Thus I implemented an application layer multicast protocol ALMI and developed an application for audio multicast that will use this middleware instead of IP multicast for creating a multicast network.

2. Definitions

This section describes some of the terms used in the document.

Controller
This is the ALMI centralized controller which handles member registrations and maintains the multicast tree.

Members
These are the members in the current multicast group.

Neighbors
Neighbor of a member is a member of the group to which it has to monitor the unicast path performance from itself. The parent and children may or may not be in its neighbor set.

Parent
Each member is assigned a parent by the controller. This parent child designation doesn’t indicate the data flow. It is used only for avoiding loops in the tree.

3. Related Work

This implementation of the ALMI protocol is almost same as described in [1] with some minor changes to suit the application needs. The only change is the incremental tree update. Two other similar application layer protocols that are very similar to ALMI are Yallcast [5] and End system multicast [4]. Yallcast uses a rendezvous host to bootstrap group members into a multicast tree. It creates a shared multicast tree using a distributed routing protocol. Overall, Yallcast envisions the deployment of IP multicast into small and disjunct network islands and provides a rudimentary architecture for global multicast. End system multicast is much more similar to ALMI. In End system multicast, members are self-organized into a multicast tree using a distance-vector multicast routing like protocol and creates source-based multicast trees. It requires members to periodically broadcast refresh messages to avoid tree partitions.
**Vat (Lawrence Berkeley national Lab.)** is a real-time, multi-party, multimedia application for audio conferencing over the Internet. Another application that provides audio conferencing support is RAT(Robust Audio Tool). RAT features a range of different rate and quality codec’s, receiver based loss concealment to mask packet losses, and sender based channel coding in the form of redundant audio transmission. But to make use of these conferencing tools, your system must support IP Multicast.

Application layer audio multicast is significantly different from the above audio tools as it doesn’t rely on IP multicast. It uses application layer multicast ALMI which provides accelerated deployment, simplified configuration and better access control.

4. **Requirements**

The users of application layer audio multicast should be able to communicate via voice in an efficient manner. The audio should be multicast using application layer multicast protocol which will be able to overcome many of the problems associated with IP multicast.

An important audio requirement will be to provide a good speech quality. Also some of the users may be interested in just listening to the ongoing audio conference. The system should be designed in such a way to allow users to participate in the conference in either way. The system should also make efficient use of network resources like bandwidth.

Middleware: The application layer multicast protocol should be implemented as a middleware which can be used by any application. It should require very slight or no modification for use by other applications.

Error Handling: The system should be able to handle any errors that arise during run time such as improper member exits, member failures. It should be designed to handle packet losses and member failures efficiently without affecting the session.

Scalability: The system should be able to handle a moderate group size without any problem.

Protocol Compliance: The system must be compatible with existing Internet multimedia protocols, and should be designed with little or no modification to the current infrastructure.

Ease of use: The application should be very easy to use and the multicast middleware should provide a framework which can be used for any application.
5. **Application Layer Multicast Infrastructure (ALMI)**

Application Layer Multicast Infrastructure was designed to support multicast groups of relatively small size with many to many semantics. The members of the multicast group are connected via a virtual multicast tree, which consists of unicast connections between end hosts and is formed as a minimum spanning tree using delay as metric.

An ALMI session consists of a controller and group members. Controller is a program located at a place which is easily accessible by all the group members. All the members are organized into a multicast tree. All the links in the tree are unicast connections through which the data is sent to all the group members. All the control messages are unicast between each member and the controller.

The Controller handles the member registration and maintains the multicast tree. It takes care that the multicast tree doesn’t split into sub trees due to node or network failures and maintains the connectivity when members join or leave the group.

When a member receives data from one of its peers, it will forward the data on all the unicast links other than the one on which it received the data packet. Data reaches all the members in this manner. Also each member periodically measures the delay from a subset of other group members. It will then send these delay measurements to the controller. The controller periodically uses all these delay measurements from group members and constructs a minimal spanning tree.

Thus in the above manner the controller manifests itself only in the control path but it doesn’t obstruct the high data rate transmissions among the group members.

Initially the member joins the multicast group by sending a JOIN message. The controller returns a member Id as well as the address of its parent. The member then sends a GRAFT message to its parent. The data transfer between two members is done using UDP as the streaming audio generates a huge amount of data and we don’t want retransmissions, reliability etc.

If the parent leaves or fails, the member will send a rejoin message to the controller. The controller will assign a new parent which has a low delay for this member by looking at its delay graph. It will also take care that the tree doesn’t split into two sub trees. Also periodically the controller calculates the MST, its cost and it will compare the cost of the current tree and this new MST. If this difference is greater than a given threshold, the controller will change the current
tree. This can be done by changing the whole tree or by updating the tree in an incremental manner which will be described in the following sections.

Application layer multicast Infrastructure (ALMI) offers an accelerated deployment, simplified configuration and better access control at the cost of a very small additional traffic load. Since it is implemented in the user space, it allows more flexibility in customizing aspects like flow control, scheduling, error recovery, security on an application specific basis.

6. Overview of Linux Sound Programming

An audio interface is a device that allows a computer to receive and to send audio data from/to the outside world. Inside of the computer, audio data is represented as a stream of bits, just like any other kind of data. However, the audio interface may send and receive audio as either an analog signal (a time-varying voltage) or as a digital signal (some stream of bits). In either case, the set of bits that the computer uses to represent a particular sound will need to be transformed before it is delivered to the outside world, and likewise, the external signal received by the interface will need to be transformed before it is useful to the computer. These two transformations are the raison d'être of the audio interface.

The Linux environment has good sound support which makes sound programming easy. As in the case of other devices, the Linux OS provides a device file, namely /dev/dsp, for accessing the sound device. So reading sound data from and writing sound data to the sound device becomes a matter of simply opening the sound device file by using the open instruction, and then issuing the read and write commands to read from and write to the sound device respectively.

The /dev/dsp device uses PCM encoding. The most fundamental parameter is sampling rate which limits the highest frequency that can be stored. It is well known (Nyquist's Sampling Theorem) that the highest frequency that can be stored in sampled signal is at most 1/2 of the sampling frequency. For example, 8 kHz sampling rate permits recording of signal in which the highest frequency is less than 4 kHz.

Quality has price. Number of bytes required to store an audio sequence depends on sampling rate, number of channels and sampling resolution. For example, just 8000 bytes of memory is required to store one second of sound using 8 kHz/8 bits/mono, but 48 kHz/16bit/stereo takes 192 kilobytes. A 64 kbps ISDN channel is required to transfer an 8kHz/8bit/mono audio stream and about 1.5 Mbps is required for DAT quality (48kHz/16bit/stereo). On the other hand, it is possible to store just 5.46 seconds of sound to a megabyte of memory when using 48kHz/16bit/stereo sampling. With 8kHz/8bits/mono it is possible to store 131 seconds of sound using the same amount of memory. These parameters correspond to a telephone quality speech.

Further the huge size of audio data can be cut down by detecting silence and removing it from the data packets being transferred. This can be done by
calculating energy thresholds for silence in the current user environment by considering the background noise levels. This can be done by recording silence for a few seconds which includes any background noise and calculate the peak energy present in these few seconds of data. We keep this as a threshold for detecting silence. If the peak energy in a sample is less than this threshold, we can assume that this sample is silence and discard it.

Also we use Real time transport protocol (RTP) specified in RFC 1889 which provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video. Those services include payload type identification, sequence numbering, time stamping. RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees, but relies on lower-layer services to do so. It does not guarantee delivery or prevent out-of-order delivery, nor does it assume that the underlying network is reliable and delivers packets in sequence. The sequence numbers included in RTP allow the receiver to reconstruct the sender's packet sequence, but sequence numbers might also be used to determine the proper location of a packet, for example in video decoding, without necessarily decoding packets in sequence.

Summary

In the above two sections, we have seen some advantages offered by application layer multicast over the IP multicast and how audio can be characterized in Linux.

The main aim of the project is to implement the application layer multicast protocol as a middleware and develop an efficient audio conferencing tool using the middleware developed.

7. DESIGN

7.1. Architecture of Application Layer Multicast Infrastructure

The basic idea behind the architecture of ALMI is that all the members are organized into a multicast tree by using a centralized controller. A link in the multicast tree represents a unicast connection between two members. All the data in any session will be sent through these unicast connections. All the control messages are unicast between each member and the controller. Since we are dealing with audio data, we don’t worry about retransmissions, reliability etc. Thus we are using UDP connection for the data transfer between members and TCP for control messages.

The Control protocol has various kinds of message formats for this infrastructure. The different messages are explained in the following section along with their function.
7.2 Tree Construction

Each member registers to the controller by sending the register message. The controller assigns a parent and randomly generates a subset of current members and sends them as the neighbors. The number of neighbors is currently defined to be at most 4 or may be defined depending on the size of the group. Also the parent assignment is done so that the degree of each member is limited to 4. The Controller also returns a unique member id for each member.

i) Initial Registration message:

It is assumed that initially the controllers address and the port number are announced through online or offline schemes. A new member will send a JOIN message which is a 4 byte “JOIN” packet. The controller responds to this join message by sending a packet of the following format.

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member ID</td>
<td></td>
</tr>
<tr>
<td>Parent IP</td>
<td></td>
</tr>
<tr>
<td>Number of Neighbors</td>
<td></td>
</tr>
<tr>
<td>Neighbor 1 address</td>
<td></td>
</tr>
<tr>
<td>Neighbor 2 address</td>
<td></td>
</tr>
<tr>
<td>Neighbor 3 address</td>
<td></td>
</tr>
<tr>
<td>Neighbor 4 address</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
</tbody>
</table>

For the first member joining the group, the parent address will be 0 and only 8 bytes are sent. The packet size depends on the number of neighbors. By default the number of neighbors is set to 4 which can be changed depending on the size of group. Also the degree of each node in the tree is restricted to 4.

ii) GRAFT message:

The member after receiving its parents address will now send a GRAFT message to the parent and check if the parent accepts the connection.

If the parent doesn’t allow the connection because it is down, then the member will send a new REJOIN message to the controller and receive its new parent and neighbor in the same manner as in above. Then it will try to connect to its new parent.
Thus the multicast tree is formed in which the unicast connection between the members will be used for data transfer while all the control messages are sent through the unicast messages between the controller and the members.

Each member runs a ping server on port 10000 which will receive any ping packets from members and replies back with a pong. Each member will periodically send a ping message to all of its neighbors and receives a pong message back. They will measure the delays and send a neighbor monitor report to the controller. The controller will store all these delay measurements in a graph. It will also periodically calculate a minimal spanning tree of all these nodes. It will then compare the current cost and the best possible and see if the difference is greater than a threshold percentage of the current cost.

iii) **PING:**

All the members will listen on port 10000 for any ping messages. The purpose of this ping server is that if it receives any ping packet it will send back a pong message with the same message ID to the sender. The message ID can be used to differentiate with stale ping messages. The ping packet format is as below:

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PING</td>
<td></td>
</tr>
<tr>
<td>Message ID</td>
<td></td>
</tr>
<tr>
<td>Member ID</td>
<td></td>
</tr>
<tr>
<td>ALMI</td>
<td></td>
</tr>
<tr>
<td>MESG</td>
<td></td>
</tr>
</tbody>
</table>

iv) **PONG:**

The ping server receives the ping messages and replies back with a PONG message. This message will contain the same message Id received in the ping packet. The structure of the pong message is as below:
v) **Neighbor monitor Report:**

All the members will measure the delays from their neighbors periodically using the PING messages. Member will store the timestamp when sending a ping packet and calculate the delay after it receives a pong message. They will then send this packet to the Controller. The structure of such a packet is as below:

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PONG</td>
<td></td>
</tr>
<tr>
<td>Message ID</td>
<td></td>
</tr>
<tr>
<td>Member ID</td>
<td></td>
</tr>
<tr>
<td>ALMI</td>
<td></td>
</tr>
<tr>
<td>MESG</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member ID</td>
<td></td>
</tr>
<tr>
<td>Number of Neighbors</td>
<td></td>
</tr>
<tr>
<td>Neighbor 1</td>
<td></td>
</tr>
<tr>
<td>Delay 1</td>
<td></td>
</tr>
<tr>
<td>Neighbor 2</td>
<td></td>
</tr>
<tr>
<td>Delay 2</td>
<td></td>
</tr>
<tr>
<td>............</td>
<td></td>
</tr>
<tr>
<td>............</td>
<td></td>
</tr>
</tbody>
</table>

There are two ways in which a member may need a new parent. Either the current parent should have left or the controller should have found a new tree and wants the member to change its parent. In either case, the member will close its connection to the parent and sends a rejoin message to the controller. The Controller will now return a new parent with the lowest delay to this member. The controller periodically calculates the new MST and its cost.

If the difference in the cost of current tree and the MST is greater than a given threshold, the controller will initiate a change in the tree. This can be done in two ways. One is completely changing the current tree and the other is an incremental update. Changing the complete tree will generate a lot of issues like
loss of data etc. Incremental Update is a way in which we just keep updating a few costly edges in the tree. We will see that in the following section. In this case of changing the tree dynamically, the controller sends a PUSH message to the member which will then close its connection to its current parent.

**vi) REJOIN:**

Whenever a member finds that its parent has closed the connection, it will send a REJOIN message to the controller and receives a new parent and neighbor set. After receiving the new parent, neighbor set, it behaves just like the initial joining. The message it receives from the controller will be in the following format:

<table>
<thead>
<tr>
<th>New Parent IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Neighbors</td>
</tr>
<tr>
<td>Neighbor 1 address</td>
</tr>
<tr>
<td>Neighbor 2 address</td>
</tr>
<tr>
<td>Neighbor 3 address</td>
</tr>
<tr>
<td>Neighbor 4 address</td>
</tr>
<tr>
<td>….</td>
</tr>
</tbody>
</table>

**vii) PUSH:**

When the controller finds a new Minimal Spanning Tree which satisfies the threshold requirements, the Controller will send each of the members a 4-byte PUSH message followed by the new parent, neighbor information.

<table>
<thead>
<tr>
<th>PUSH</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Parent IP</td>
</tr>
<tr>
<td>Number of Neighbors</td>
</tr>
<tr>
<td>Neighbor 1 address</td>
</tr>
<tr>
<td>Neighbor 2 address</td>
</tr>
<tr>
<td>Neighbor 3 address</td>
</tr>
<tr>
<td>Neighbor 4 address</td>
</tr>
<tr>
<td>….</td>
</tr>
</tbody>
</table>

| …. |
The member will close its connection to the current parent and then use the received information and connect to the new parent and also update its neighbor set.

7.3 Neighbor Management

All the members periodically calculate their delays from their neighbors and report them to the controller. The controller maintains a graph with all these delays. If there is no delay measurement between a member and another, it will assume the delay is infinity. Then periodically, it will calculate the minimum spanning tree for this graph and the cost of the MST.

**Minimal Spanning Tree:**
The minimal spanning tree is calculated using Prim’s algorithm. The first vertex is taken as the starting vertex. Find the nearest neighbor which doesn’t make a closed loop and add the edge to the visited edges list. Keep doing this until all nodes in the graph are visited. The cost of this tree is calculated using the delay measurements made by the members. A very high delay is assumed for edges in the graph without a delay measurement.

The controller calculates the difference between the cost of this tree and the current tree. If it is greater than a given threshold, the current tree will be updated. This is done by sending a PUSH message to the member and assigning it a new parent with the lowest delay.

The periodic time delay for calculating the Minimal spanning tree can be set depending on the application. Since we don’t want to lose data while tree is being updated, we can set a high tree calculation delay.

Whenever a member fails or leaves, the controller updates its graph. The children of this member will be assigned a new parent in such a way that the tree doesn’t split into two sub trees. Also the structure of the tree below this member will remain the same. The new parent is randomly picked up and tested that this new parent doesn’t have a path from the member in the current graph.
Suppose node B fails in the tree as in the above figure, member D will now send a rejoin message to controller. The Controller will randomly pick a new parent for D. This new parent will be chosen in such a way that no path exists between these two members. A path is said to exist between two nodes if either of them is reachable from the other. So in the above example, it may return C or F or G but not H or I. The sub tree D, H, I will remain the same.

The root node of the tree doesn’t have any neighbors to monitor. The neighbor set of each member is limited to 4.

7.4 Incremental Tree Update

The Controller periodically calculates the minimal spanning tree and decides if there is any update to be done to the current tree. The method for updating the tree is more application specific. Since we are dealing with audio conferencing, complete tree update may not be a good choice as there will be many issues like missing packets etc. The incremental tree update mechanism is more suitable in such a situation.

In this method, whenever the difference between the cost of the current tree and the best possible MST is greater than the threshold, the controller will pick up an expensive edge from the current tree and send a PUSH message to the children on
that link. When the children return to the controller for a new parent, the controller will issue a new parent with the lowest delay from this member.

It can repeat this process of removing the expensive edges until the difference between the cost of tree after removing a link and the possible minimal spanning tree falls below the threshold. If the periodic tree calculation is very frequent i.e., the time delay for calculating the new MST and compare it to the current tree is low, then we can update a couple of expensive edges instead of updating the tree until the threshold condition holds.

7.5 Data Transfer

The data transfer in this architecture is done by each member forwarding the data packets on all edges connected to it other than the edge on which it received the data packet.

The programs start an UDP Data server thread which takes care of transferring the data. This runs on the port 9997. The function of this server is to receive any data packet from one of its peers and forward it out on all other edges. This also checks the source of the packet. The data packet contains member Id and a sequence number. All the members store the highest sequence number received from each of its peer’s. This is done to overcome any reordering of data.

Each data packet may typically contain audio chunks of 10msec or 20msec or 40msec as chosen at the program startup. It follows Real-Time Transport Protocol (RTP). Each packet is preceded by a RTP header. The RTP header indicates what type of audio encoding (such as PCM, ADPCM or LPC) is contained in each packet so that senders can change the encoding during a conference, for example, to accommodate a new participant that is connected through a low-bandwidth link or react to indications of network congestion.

The Internet, like other packet networks, occasionally loses and reorders packets and delays them by variable amounts of time. To cope with these impairments, the RTP header contains timing information and a sequence number that allow the receivers to reconstruct the timing produced by the source, so that in this example, chunks of audio are contiguously played out the speaker every 20 ms. This timing reconstruction is performed separately for each source of RTP packets in the conference. The sequence number can also be used by the receiver to estimate how many packets are being lost.
The RTP header has the above format. The fields are defined as below:

i) Version (V): 2 bits
   This field identifies the version of RTP.

ii) Padding (P): 1 bit
    If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload.

iii) Extension (X): 1 bit
    If the extension bit is set, the fixed header is followed by exactly one header extension.

iv) CSRC count (CC): 4 bits
    The CSRC count contains the number of CSRC identifiers that follow the fixed header.

v) Marker (M): 1 bit
    The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream.

vi) Payload type (PT): 7 bits
    This field identifies the format of the RTP payload and determines its interpretation by the application. A profile specifies a default static mapping of payload type codes to payload formats.

vii) Sequence number: 16 bits
    The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.
vii) Timestamp: 32 bits
   The timestamp reflects the sampling instant of the first octet in the RTP data packet.

viii) SSRC: 32 bits
   The SSRC field identifies the synchronization source.

ix) CSRC list: 0 to 15 items, 32 bits each
   The CSRC list identifies the contributing sources for the payload contained in this packet. The number of identifiers is given by the CC field.

x) The payload data’s size is determined by the parameters specified at the program startup.

Since we don’t use any mixer, the CSRC count will be zero. Also we use the ALMI member ID as the synchronization source identifier.

Also the audio data is buffered at the receiver to avoid any breaks while playing due to delays caused by the tree update etc. The data packet can be modified by removing RTP header and create a different format to suit any application. In this current application, each member writes the payload to the audio device after setting up the device. Then it will forward the same packet through all the outgoing edges other than the one on which it received the packet.

7.6 Audio Design

Digital audio is the most commonly used method to represent sound inside a computer. In this method sound is stored as a sequence of samples taken from the audio signal using constant time intervals. A sample represents volume of the signal at the moment when it was measured. In uncompressed digital audio each sample requires one or more bytes of storage. Number of bytes required depends on number of channels (mono, stereo) and sample format (8 or 16 bits, mu-Law, etc.). The length of this interval determines the sampling rate. Normally used sampling rates are between 8 kHz (telephone quality) and 48 kHz (DAT tapes).

Sampling introduces some error. Two factors are key in determining how well the sampled signal represents the original. Sampling rate is the number of samples made per unit of time (usually expressed as samples per second or Hertz). A low sampling rate will provide a less accurate representation of the analog signal. Sample size is the range of values used to represent each sample, usually expressed in bits. The larger the sample size, the more accurate the digitized signal will be. For example, the sampling rates supported by a device can be from 4000 to 44000 samples per second. The sample size can be 8 bits or 16 bits.

The parameters that are used in this program are a sampling rate of 8000 samples per second, mono and 8 bit unsigned data. These are chosen as almost every audio device shall support these features when proper drivers are installed. These parameters also correspond to a telephone quality speech.
Typical audio conferences sample the audio device every 10, 20, 40 or 60 ms in order to keep latency low. At the above parameters, one second of audio is stored in 8000 bytes. By default, the device is sampled every 10msecs to keep the latency low. But users can choose any of the above four intervals. This data is processed to find if this 10msec chunk corresponds to silence or speech by calculating the energy in this packet.

The program starts a thread which will perform the audio recording and creates an RTP header for each packet and sends it out on the multicast tree.

7.7 Silence Detection

The program calculates the default energy threshold by recording a few seconds of silence at the beginning of the program. This will be used as the threshold for determining if a packet comprises of silence or speech. This silence detection will reduce the amount of data to be sent across the network.

Based on the silence threshold setup, we decide if a packet needs to be sent across the network. By default, every 10ms data is taken and the energy is computed. In case we sample the device in a different time interval like 20ms or 40ms, there may be more than one 10msec sample. The energy in each 10ms sample is calculated and the highest energy is returned.

7.8 GUI

This will also provide a simple basic three button GUI as below to start /stop the audio recording and exit the program. The user can press the record button whenever he wants to start recording or he can just be an audience to the ongoing conference.

![Record Stop Exit]

The stop button is provided so that the user can stop recording and just hear what is going on. The exit button should be used for completely closing the program.

In future, the graphical user interface will be changed so that it can also allow the user to have controls like volume, microphone control etc.
8. Implementation Details

The application layer audio multicast is implemented using Berkeley Unix sockets API.

The controller starts with initializing all the data structures. It will create threads to receive neighbor delay measurement reports and periodically calculate the minimal spanning tree ($DelayReports ()$, $PeriodicTree ()$). It will bind to the port specified and will keep listening for any member requests. It will keep checking if there is any new member request or if there is any request or connection termination from an already existing group member. It will use the function $HandleTCPClient ()$ to handle different requests from new members. It uses the function $HandleCtrlRequest ()$ for handling requests for a new parent or if one of the existing member exits. These are the major functions called and remaining functions are all called from these.

The member starts by calling the initialize function to initialize the audio device and its sampling parameters. It then creates threads for Data transfer, Ping Server and to start the GUI to start/stop recording. The functions called are $DataTransfer ()$, $PingServer ()$ and $Readdata ()$, respectively. It then sends a JOIN message to the controller and receives its parent and neighbor set. It starts a thread to periodically measure delay from its neighbors using the function $DelayMeasure ()$. These are the major functions called and remaining functions are called from these.

The functional flow diagrams below show the flow of the Controller and member programs.

*Using ALMI as a middleware:*

One of the main aims is to implement ALMI as a middleware package that can be used by any application. This can be achieved by simple modifications to the existing functions. The only two major changes are reading the input data to be sent and handling the received data at each member. Depending on the application, the user can change the $Readdata ()$ function in the member program to read and send its own data instead of the audio data. Instead of reading data from microphone, one needs to send the data from the new application. Similarly, the user should change the $DataTransfer ()$ function to handle the data being received, i.e., consider the size and type of data. Instead of buffering data and playing the data, one needs to specify a way to handle this data.

Functional Flow Diagrams for Controller and members are illustrated below:
Controller
Members
Described below are the important functions and data structures and their use.

8.1 Application Layer Multicast

8.1.1 Data Structures

**Controller:**

i) 
```c
typedef struct Member {
    Unsigned long int cliID;
    Unsigned long sinaddr;
    int parent;
} MemberInfo;
```

MemberInfo Group [MAXMEMBERS];

This array of structures is used to store all the member details. The \( i^{\text{th}} \) structure has the information of the \( i^{\text{th}} \) member in the group. The unsigned long cliID is used to store the member Id of a member. The unsigned long sinaddr is used to store the IP address of a member. The parent index is stored in the unsigned long parent.

ii) `int *membersocks;`

This integer array is used to store all the sockets used to communicate with the group members.

iii) `int graph [ ] [ ];`

This is used to store the current tree. A zero means that there is no edge between the two members. A one means that those two members form an edge in the tree.

iv) `int Cost [ ] [ ];`

This is used to store the delay measurements from members. This is used as the cost matrix for minimal spanning tree calculation.

v) `int min_span_tree [ ] [ ];`

This is used to store the periodically calculated minimal spanning tree.

**Members:**

i)
```c
typedef struct Sockets {
    int peerSock;
    Unsigned long peerAddress [ ];
} SockList;
```

SockList PeerInfo [MAXPEERS];

This array of structures is used to store the information about each member to which it has a connection. `peerSock` is used to store the socket for
communicating with the peer. \textit{peerAddress} is used to store the IP address of the peer. The first record corresponds to the Controller and the second record contains the parent information.

iii) \textit{Unsigned long int \texttt{neigh} [ ]}:  
This is used to store all the neighbors addresses to which the member should send a ping message and measure the delay.

8.1.2 \textbf{Functions}

\textit{Controller:}

i) \textit{initialize():}

This function initializes all the major data structures used in the program like initial graph, member addresses etc.

ii) \textit{getmemindex():}

This function returns the index of the given member. The input is the member address. It returns the index of this member by referring to the member address array.

iii) \textit{saveReports():}

The neighbor delay measurement report sent by a member is parsed out and all the delays between a member and its neighbors are stored in the corresponding positions of the cost matrix. The input for this function is the neighbor report and the address of the member who sent it. It will find the index of the member who sent this report and the indices of the neighbors in the report and stores the delays in the corresponding positions.

iv) \textit{PeriodicTree():}

This is the function called by the thread created to periodically calculate the minimal spanning tree. Depending on the delay specified for calculating the minimal spanning tree, this function will lock the mutex to avoid any concurrent changes to the current graph. Then it will calculate the best possible minimal spanning tree from the current graph and cost matrix. It will compare the cost of the current tree with this new tree and depending on the threshold, it will decide whether or not to initiate the tree update. It will use the incremental tree update method as described above and sends push messages to members on expensive edges.
v) **DelayReports():**

This function is called by the thread created in the main program to start an UDP server on port 9997 to receive the neighbor delay measurement reports sent by members in the multicast session. The function of this server is to receive the reports sent by the members and call the function saveReport to save the delays between respective members.

vi) **HandleTCPClient():**

This function handles all the members sending the join message. It verifies the received join message and discards the message if it’s wrong and closes the connection. If the message is correct, it issues a unique client Id and stores the members address and socket in the corresponding lists. It then calls the function to allocate the parent and neighbors to this member. It returns when the member has been issued a parent and neighbor set.

vii) **AllocateParentNeighbours():**

This is called by the HandleTCPClient function to allocate the parent and neighbor set for a new member. This function will call the corresponding joinok function depending on the current group size.

viii) **makeJoinOK():**

This function sends the new member a parent address and a set of addresses for neighbors. It will return a zero as the parent when this member is the only member in the group. Otherwise it will randomly pick a parent whose degree is less than 4. It will update all the data structures like parentof [], graph [] [], Cost [] [] etc. This uses the srand (time) as the seed for the Random () function to pick up a random parent, neighbors.

ix) **HandleCtrlRequest():**

This function is called up when there is some request on one of its member sockets. It handles the requests from current members. The only possibilities to deal with are that the member should have closed its connection or it should be requesting a new parent. It verifies this and acts accordingly. In the first case, it will close the member socket and update all the data structures, reduces the total number of members in the group.
x) *AllocatenewParent():*

This function is called by the HandlectrlRequest function, when it receives a request for a new parent from an already existing member of the group. The function will check if there are no other members in the group or else it will try to pick up a parent for this member by looking at the cost matrix and selecting a parent whose degree is less than 4 with the lowest delay. It will also return a new neighbor set. If that is not possible, it will randomly pick up a parent as it does for a new member.

xi) *RearrangeSockList():*

This function is used whenever all the data structures need to be updated. This may happen when a member leaves the session or dies. This will remove such member’s socket, address, entry in the graph, cost matrix etc.

xii) *Spantree():*

This function calculates the minimal spanning tree depending on the cost matrix. It uses prim’s algorithm to find the MST. It returns the cost of the best possible minimal spanning tree and the number of edges in this tree. This will be used by the periodic tree calculation function to decide if the current tree should be updated.

*Members:*

i) *DelayMeasure():*

This function is called by the thread created in the main program to do the delay measurements and send the report to the controller. This will lock a mutex to deny any concurrent updates being made to the neighbor list. This will periodically send a ping packet to each neighbor in the neighbor set and measure the delay from the neighbor. If a neighbor doesn’t reply with a pong message after a specific time delay, it will report the delay as infinity. Then it will send this monitoring report to the Controller.

ii) *ReadData():*

This function is called by the thread created in the main program to read audio from the microphone and detect if the packet contains silence or speech. It will then forward this data out to all members that are connected to it. This thread can be started or stopped depending on the audio requirement. If a member just wants to
listen to other members, he can turn this off by not creating the thread that calls this function.

iii) DataTransfer():

This function is called by the thread created in the main program to take care of the data packets coming from members in the group.

This function will store all the arriving packets in buffer which can store 100 milliseconds of data. Then plays the buffer when it is full. This buffering will avoid any breaks in the speech due to network delays.

iv) PINGSERVER():

This function creates an udp ping server on port 10000. The purpose of this server is to reply back a pong message to any ping packet it receives from any member. This is useful for calculating the delay by a member.

v) CreateTCPServerSocket():

This creates a socket and listens on that port for any incoming requests from members. The messages it will handle is the GRAFT message sent by a member.

vi) HandleNewConnect():

The member accepts the graft message received by a member if its current degree is less than 4. Otherwise it will close the connection. It will add the new member to its peer list and increments the number of members it is connected to.

vii) HandleCtrlRequest():

This function is called when the member has some data on the socket. The possibility is that the member should have closed its connection and thus it will update the peer socket list.

viii) HandleParentFailure():

This function is called when there is a request to change the parent from the controller or when the current parent leaves or dies. It will close the parent socket and request the controller for a new parent. It will then receive the new parent and neighbor set. It will connect to the new parent and add it to the peer list.
8.2 Audio Conferencing

8.2.1 Functions

i) initialize():

This function initializes the sound device with the required parameters. The sampling rate will be set to 8000 bytes per second. The number of channels will be set to mono and the size of sample will be set to 8 bits. The format of the data will be unsigned 8 bit. The program will exit if the audio device cannot work using any of the above specified parameters.

ii) Energy_Silence():

This function records some silence and calculates the energy level in these silence packet. It will calculate the thresholds for silence data.

iii) Energy_Audio():

This function will calculate the energy of the data in the audio buffer. Energy is measured in 10ms samples. Since there may be more than one 10ms sample, it will return the maximum 10ms energy calculation found.

9. Conclusions and Future Work

I was able to successfully implement the application layer multicast infrastructure (ALMI) and develop an application for multicasting audio. The application layer multicast infrastructure implementation can be used to develop any multicast application by adding suitable code. This is more like a middleware which can be used by any application.

In future, I will address issues like reducing the audio data size using techniques like encoding, compression etc. Also I would like to develop a friendlier graphical user interface for the audio application with advanced features to support any audio device and sampling parameters. We need to address issues that arise like the playback problem if each member has a device that supports a different set of parameters.

10. Learning Outcome

In this project, I was introduced to the concepts of multicast and audio programming. I learned the entire development cycle of a complex system – design, development, testing and documentation. I also learned the intricacies involved in the
implementation and development of a protocol. I gained a good understanding of the concepts of Linux audio programming. In general, this project improved my programming style and my understanding of various network protocols.
11. References


APPENDIX A – System Requirements

This section describes the software and hardware requirements for running this tool.

**Hardware Requirements:**

1. System Architecture: Intel PC architecture (Intel Pentium 200Mhz or greater recommended)
2. Memory: 128MB
3. Disk space: 30MB
4. Audio Hardware:
   a. Microphone, Speakers
   b. Sound Card supporting a sampling rate of 8000Hz, 8bit, mono full-duplex communication. E.g. SoundBlaster 128/512 etc

**Software Requirements:**

1. Linux Kernel 2.4.x.x.
2. GNU C libraries Glibc.2.x (gcc Version 2.95) Library
3. Berkeley Sockets API.
4. Sound card driver that support the specified sampling parameters.

For Clients using GUI:

5. GNU project C++ Compiler (g++)
6. glut - OpenGL Utility Tool kit
7. glui_v2_1_beta package.

**Network Connection:**

A TCP/IP network connection to the internet.