AudioPeer: A Collaborative Distributed Audio Chat System*

Roger Zimmermann, Beomjoo Seo, Leslie S. Liu, Rahul S. Hampole and Brent Nash
Integrated Media Systems Center
University of Southern California
Los Angeles, California 90089
[rzimmerm, bseo, shihuali, hampole, bnash]@usc.edu

Abstract

Educational tools that utilize the Internet to reach off-campus students are becoming more popular and many educational institutions are exploring their use. We report on a multiuser audio chat system that is using a multicast peer topology and is based on a new audio streaming protocol called YimaCast. *The tool is designed to foster collaboration and interactive learning between students, teaching assistants and instructors. The decentralized nature of the audio chat system avoids bottlenecks and allows it to scale to large groups of participants.*

We describe the system architecture, its multiple components and how it integrates with the existing distance education infrastructure at our university. We include some preliminary results from our prototype system that demonstrate the feasibility and practicalilty of our approach.

1 Introduction

The expanding capabilities of the Internet to handle digital media streams is enabling new applications and transforming existing applications in many areas. One of the fields that is profoundly affected is higher education. Traditionally, students have attended classes in lecture halls on college and university campuses. The next step was distance education, enabled via video satellite links. More recently, new communications media have broadened the potential audience and allowed anybody with a broadband Internet connection to potentially receive audio, video and slide presentations on their computers.

However, with the basic lecture distribution technology now being available, many educational institutions are grappling with the impact of this technological shift in education. Many pedagogical and policy issues must be resolved in addition to the technological challenges. The University of Southern California's Distance Education Network (DEN) is actively engaging in learning and technological issues to provide both on- and off-campus students with tools that enhance their learning experience. There is evidence from learning and psychology research that a simple one-way communication (i.e., lecture broadcast) is providing less of a learning experience for a student than actually attending an on-campus class would provide. Consequently, we have been engaged with DEN to provide collaboration tools that would make the learning experience more interactive. Our first project is a multiuser audio chat room system that is designed to allow groups of students to discuss assignments, allow teaching assistants to conduct lab sessions, and enable student questions and feedback during lectures.

The multiuser audio chat system involves numerous technical challenges that need to be addressed to build such an application. The number of participants in a chat session may be several dozens, with each student needing to hear and possibly talk to any other person in the session. Additionally, the end-to-end audio latency needs to be kept sufficiently low such that natural interaction is possible. Hence, we aim for our system to be scalable, practical (e.g., work with different types of network connections), integratable into the DEN infrastructure, and extensible with new features (e.g., speaker recognition).

In this study we report on the design and implementation of our AudioPeer chat room that is built on our overlay network multi-cast protocol called *YimaCast*. We have chosen a peer-to-peer (P2P) architecture to allow the chat room system to scale to a large user base without requiring massive resources on a central server system. We present design choices and implementation details of our system and provide some initial experimental results. The contributions of this report are in its description of multiple components that integrate into a fully working implementation. An initial field test of AudioPeer with a pilot class was conducted during the fall 2003 semester.

The rest of this report is organized as follows. Section 2 details the system design and components. Preliminary experimental results are presented in Section 3 while Section 4 surveys the related work. Finally, future extensions and enhancements are described in Section 5.

2 System Design and Components

Our system aims to provide an efficient audio chat application with low audio latency. One of the primary concerns is an efficient interconnection topology and architecture. An immediate first approach would be to connect each participant to a central server that merges incoming audio streams and then distributes the final mixed result to every listener that is connected. The advantage of such a star layout is that the sessions can be centrally managed and the

*This research has been funded in part by equipment gifts from Intel and Hewlett-Packard, unrestricted cash grants from the Lord Foundation and by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, Cooperative Agreement No. EEC-9529152.

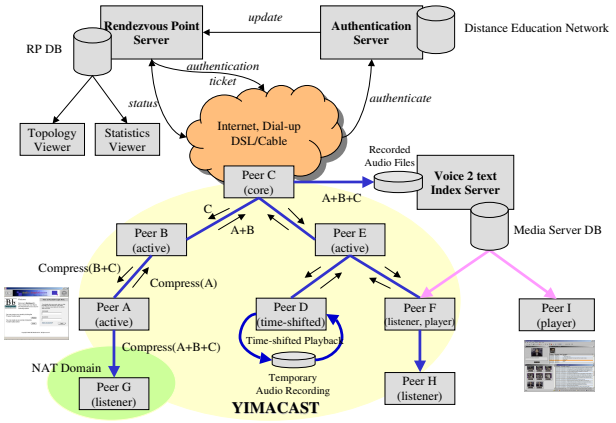

Figure 1. Overall architecture of AudioPeer with YimaCast.

delay for the sound streams to be relayed by the server depends mostly on the distance of the users from the server. The disadvantages of this architecture are that the central server requires a large amount of resources (for example memory and network bandwidth) that is proportional to the number of participants. Furthermore, the server can easily become a bottleneck and also is a single point of failure. Therefore, we adopted a distributed peer architecture where a newly joining user may be connecting to one of her peers who is already participating in an ongoing audio chat session. We call this low bandwidth consuming overlay network *YimaCast*.

One of the challenges with a distributed architecture is that the end-to-end audio latency may be more variable. From existing research we know that for an interactive conversation the delay from the microphone input through the transmission to the audio speaker output should not exceed 150 to 250 milliseconds for a natural conversation. Our audio chat system is designed to work within these limits and to dynamically adjust peer connectivity to optimize the audio transmission. Audio mixing is performed at each multicast member node to reduce the network bandwidth significantly. As a result, many simultaneous audio chat sessions can be supported. While YimaCast manages the multicast tree within a session, multi-session management is provided at the system's level. Our software suite consists of four components as shown in Figure 1: multiple *AudioPeers*, a *rendezvous point* (RP) *server*, an *authentication server*, and a *voice2text indexing service*.

AudioPeer denotes the integrated collection of modules that allow a user to attend audio conferences. It includes an embedded web browser, an application-level multicast connection manager, and an audio manager. From the embedded web browser the user can contact the authentication server and log onto the system. An interface to query and retrieve pre-recorded audio files is also provided. The connection manager handles the application-level connections with other remote AudioPeers through the YimaCast application-level shared multicast protocol. The audio manager, described in Section 2.2, captures audio samples and plays them continuously. It also mixes incoming audio packets sent from neighboring AudioPeers connected through the YimaCast delivery path, subtracts the mixed stream, and forwards the result to the designated destination nodes.

An AudioPeer can take on one of five different functional roles: active peer, time-shifted peer, passive peer (or listener), recorder, and player. An active peer participates in online discussions. It requires low end-to-end latency with other active peers. A time-shifted peer may suspend the play-out of the current discussion temporarily. In the meantime, it automatically records the incoming audio packets into a file, which can be reproduced when the user requests a resume operation. To catch up with the current on-going session, it may skip some audio packets, such as silence. A listener is a passive user who mostly listens to the current discussion and speaks infrequently. It requires less tight delay bounds, enabling higher audio quality. Usually, active peers are located near the Yimacast core node, while listeners are attached to the leaves of the tree. A recorder, a special case of a listener, receives audio packets and stores them in an audio file. Finally, a player renders pre-recorded audio content stored on the voice2text indexing server.

The distance education network (DEN) at our university provides all registered students with electronically available educational services. The web-based *authentication server* maintains class information, registered user information, and recorded lecture materials. Thus, any user of our system is authenticated through the DEN system. After verifying a user, the server then forwards the login information to the rendezvous point server.

The *rendezvous point (RP) server* is the bootstrap node that enables an authenticated node to join ongoing sessions. To this end, it stores information about users, currently available sessions, and other peers. For administrative purposes it also provides a statistics viewer and an YimaCast topology visualizer. Once an AudioPeer is authenticated, it can freely join and leave sessions at any time without contacting the RP server because YimaCast allows decentralized tree migration by design.

The *voice2text indexing service* allows users to perform a keyword search and retrieve the matching audio fragments through a web interface. Functionally it consists of an audio recorder and an indexing server. The recorder is connected to one of the AudioPeers in every live session and stores the audio packets in a file. The indexing server then extracts keywords and associated audio fragments from these files (currently an off-line process). We plan to use three audio processing plug-ins: speech recognition, speaker identification, and audio classification. Speech recognition generates keyword indices by comparing speech in an audio signal to words in the speech recognition dictionary. Speaker identification identifies speakers in an audio signal by comparing voices in an audio sig-

nal to those of speakers in a speaker identification dictionary. Finally, audio classification recognizes specific audio types such as thunder, applause, or laughter, and stores time stamped information for classification based queries.

2.1 YimaCast Protocol

YimaCast is an application-level multicast protocol designed to serve as a reliable audio streaming platform that provides minimum overall end-to-end delay among all peer nodes. Aiming at high reliability and low latency, the YimaCast protocol dynamically maintains a shared multicast tree among all peer nodes. For space reasons we must restrict our presentation of the YimaCast protocol to a relatively high-level description.

2.1.1 Tree Construction and Maintenance

YimaCast builds and maintains the multicast tree by confining the operation of the AudioPeer client to the well-defined states shown in Figure 2.

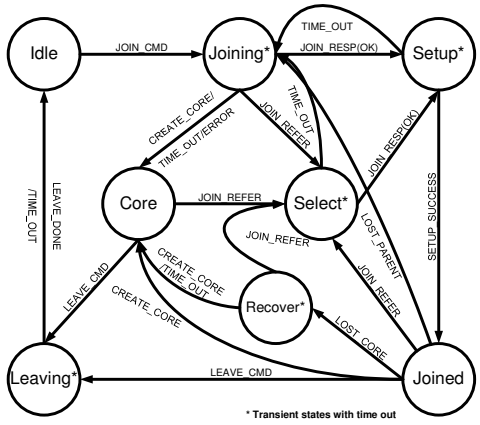

Figure 2. AudioPeer state diagram.

Tree construction is initiated with a connection to the RP server, which replies with information needed for the AudioPeer client to set up an audio connection with the exiting peer nodes. The bootstrap phase is the only time when the YimaCast protocol needs to contact a central server; after that, all tree maintenance is performed in a distributed manner. In order to achieve the minimum end-to-end delay, we must to solve the famous Steiner Tree problem, which is known to be $\mathcal{NP}$-complete [3]. Fortunately, since all the nodes in our tree are multicast members, the Steiner Tree problem is reduced to the minimum spanning tree (MST) problem, which can be solved in polynomial time. There exist several centralized and distributed algorithms to solve the MST problem, e.g., Prim's algorithm [5] and the approach proposed by Gallager et al. [7]. However, these algorithms need to rebuild the complete tree every time when there is a node join or leave operation. They are not suitable for our application which operates in a very dynamic environment. Hence, during the join process, YimaCast uses a heuristic Shortest Path Tree (SPT) algorithm, which attaches the new node to the nearest known nodes in the multicast tree. The design is to achieve minimum tree delay cost while concurrently minimizing the service interruptions to the existing nodes when modifying the tree.

YimaCast is also designed to handle the dynamics of a distributed environment. At run time, various errors can occur, e.g., the failure to establish a connection with the parent node, the loss of a connection during operation, the loss of the core node, etc. By incorporating a multi-layered error handling policy, YimaCast can repair most of these errors without asking for help from the RP server. Our experiments also show that most of the errors are recoverable in this distributed manner.

2.1.2 QoS-based Tree Optimization

YimaCast addresses two QoS issues related to audio conferencing: the end-to-end delay and playback hiccups. The end-to-end delay denotes the latency between speakers and listeners and the goal is to minimize this delay. Playback hiccups are caused by the variable delivery time of packets over a standard TCP/IP network. Data buffering at the playback side can help to smooth out jitters, but at the price of increasing the end-to-end delay. To solve the conflicting goals between these two QoS issues, YimaCast introduces a dynamic tree optimization algorithm that depends on the participants' individual QoS demands for the audio chat room.

A user may be in either a listening or a speaking mode. Naturally, users who speak more frequently require shorter end-to-end latency because too much delay between speakers will make the conversation uncomfortable. At the same time, users who are mostly listening may tolerate a longer delay. In this case, the focus should be on minimizing the playback hiccups to obtain a better listening experience.

Dynamically reducing the end-to-end delay among speakers is achieved by clustering the speaker nodes. This is carried out by continuously monitoring the behavior of the user and – if a user speaks frequently – gradually migrating her toward the core. On the other hand, if a user keeps silent for extended times during a chat session, the client will increase the audio playback buffer to reduce audio hiccups. With this algorithm, active peers move closer to the core while passive peers are migrated to the edges of the tree, hence the QoS requirements of both groups are met. It is very noteworthy to mention here that the selection of the core node does not significantly affect the performance of the optimization. The reason is that the core node is only used as the generic reference direction in which active peers move together. Once clustered, the delay between the speakers is optimized. Therefore, the optimization result is relatively independent of the core position.

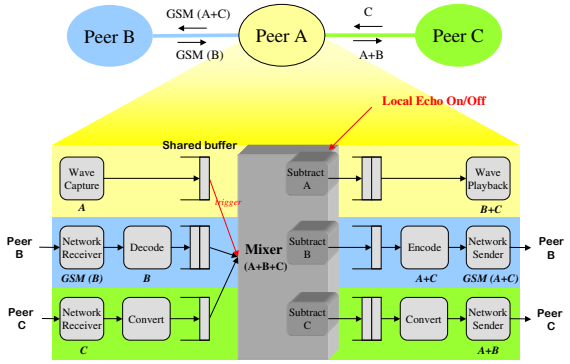

Figure 3. Audio mixer module.

2.2 Software-Based Audio Mixing

The audio mixing algorithm focuses on minimizing the network utilization for our audio conferencing application. Unlike in video conferencing, it is possible to aggregate the uncompressed audio sources through simple arithmetic calculations, preserving the original audio bandwidth.

Table 1 lists the currently supported audio media types and their characteristics: high-quality, low latency (PCM stereo); medium-quality, low latency (PCM mono); low-quality, low latency (GSM.610); and high-quality, high latency (MPEG-1 Layer 3). Among the audio codecs available, GSM.610 is reported to have a small compression delay and a tolerable audio quality. The high-quality high latency audio format is useful because some AudioPeers, participating as listeners or recorders may be connected via a low bandwidth network.

Each peer node is equipped with an audio mixing module that relays the incoming audio from remote nodes to the outgoing connections. Our design allows heterogeneous audio transmissions and each conference participant can create a specific quality of audio samples, such as PCM stereo sound and GSM mono sound. We use a software-based audio mixing algorithm called *decode-mix-encode* [12]. A linear mixing algorithm requires all the input audio bitstreams to be uncompressed for simple arithmetic additions and subtractions. Thus, all incoming encoded bitstreams are decoded into their uncompressed form, and the resulting uncompressed bitstreams are merged to a mixed bitstream. This stream is later used when constructing the outgoing streams for each respective remote nodes.

To illustrate, consider an example with three peer nodes, *A*, *B*, and *C*, connected as shown in Figure 3 (*A* is connected to *B* and *C*). Peer *A* locally captures uncompressed audio samples with a bandwidth of 1.5 Mbps; two unidirectional links between *A* and *B* are constructed to transfer compressed audio packets, say, as 13 Kbps GSM streams. The links between *A* and *C* use uncompressed PCM audio transmissions at 64 Kbps. *A* receives the audio packets β and γ respectively from *B* and *C*. It also generates newly captured audio packets α periodically. With this information the audio mixer performs the following steps.

1. Transcode incoming audio packets into linearly uncompressed audio packets. Uncompressed audio is stored in memory for future subtraction.
2. Mix all uncompressed incoming packets by adding arithmetically, $\phi = (\alpha + \beta + \gamma)$.
3. Subtract the original audio from the mixed stream. For local playback, subtract the original audio packets α from ϕ, resulting in $(\beta + \gamma)$. For *B*, subtract β from ϕ, resulting in $(\alpha+\gamma)$. For *C*, subtract γ from ϕ, resulting in $(\alpha + \beta)$.
4. Encode the subtracted audio bitstreams in the form of the network connection supported audio format. The local playback module, *A*, does not need any encoding. The outgoing audio bitstream, $(\alpha+\gamma)$, for *B*, needs encoding from a 1.5 Mbps PCM bitstream to a 13 Kbps GSM bitstream. For *C*, $(\alpha + \beta)$ needs transcoding from a 1.5 Mbps PCM bitstream to a 64 Kbps PCM bitstream.
5. Packetize the audio bitstreams and send the packets to the remote peer nodes, respectively.

Simple addition, while preserving the volume level throughout a chat session, may cause integer overflows when multiple talkspurts are added simultaneously. Another approach, dividing the original talkspurt by the number of participants or by the number of active talkspurts before adding it to the mixed stream, prevents this overrun problem. However, it requires intelligent floor control mechanism to detect the active talkspurts and generates additional exchange overhead of control messages.

Our implementation uses an augmented version of the simple addition algorithm. It detects overruns before the addition step and lowers the volume level of the audio sources. The mixer is implemented as a single thread with real-time priority. It is blocked until the local capture module sends a signal to wake it up. Immediately, it collects the uncompressed incoming audio samples, aggregates them and subtracts the original data. This mechanism guarantees a continuous hiccup-free audio transmission to the remote nodes. One slight drawback is that it may increase overall end-to-end delay because of queuing delays at the mixing module.

3 Experimental Evaluation

We present some preliminary results from experiments measuring the end-to-end audio delay between two AudioPeer clients in a LAN/WAN environment and identify the primary causes of any delays. We performed our experiments on a Windows platform. Note that all results are subject to improvements due to changes in audio I/O drivers, operating system support, and specialized hardware.

The end-to-end audio delay can be modeled as the summation of the capture interval, the play-out delay, and any network latency. With our experiments, we identified the two primary components of the end-to-end audio delay: the capture interval and minimum play-out delay. The

Supported audio media types				
Compression Type	uncompressed		compressed	
Audio Format	PCM	PCM	GSM.610	MPEG Layer3
Bits per Sample	8	16	8	16
Channels	mono	stereo	mono	stereo
Sampling Rate	8 KHz/16 KHz	48 KHz	8 KHz	48 KHz
Delivery Rate	64 Kbps/128 Kbps	1.536 Mbps	13 Kbps	56 Kbps
Usage Method	LAN	LAN	dial-up modem	cable modem, DSL
Latency Requirement	low	low	low	high
Audio Quality	medium	high	low	high

Table 1. Audio types supported.

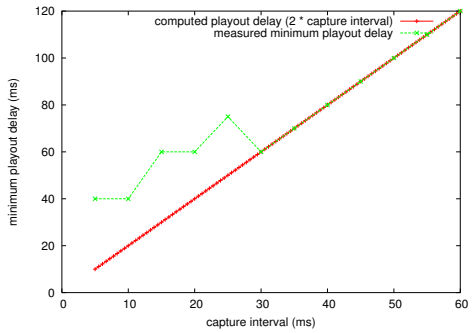

Figure 4. Minimum play-out delay per capture interval.

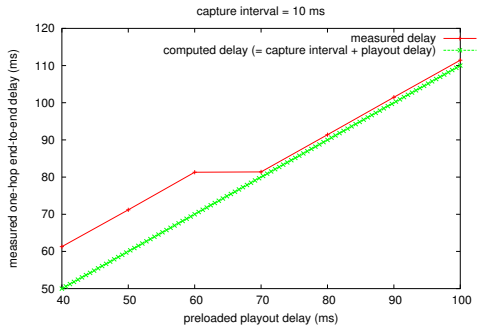

Figure 5. Measured one-hop end-to-end delay in a LAN environment.

capture interval, in milli-seconds (ms), is the time period between successive callback functions to collect the audio samples from the audio driver. If the capture interval is 20 ms, a callback function is invoked every 20 ms to transmit the captured audio samples from the driver to the application. The *minimum play-out delay*, also represented in milli-seconds, is the time used to pre-load the audio samples for smooth audio play-out and to compensate for network jitters and slightly irregular capture intervals.

Experimental setup. The Windows MME (MultiMedia Extension) API was used for waveform capture and playback. To precisely measure the end-to-end audio delay, we used an audio split cable. The two inputs of the cable were connected to the original audio source and the receiving AudioPeer. The output was recorded on another machine and the maximum delay offset between the two inputs was computed using cross-correlation in MATLAB. We repeated this experiment ten times with the same configuration to reduce the statistical errors.

Figure 4 shows the minimally required play-out delay as a function of the capture interval with no audio dropouts. We set the play-out delay as a multiple of the capture interval. We observe that (1) the minimum play-out delay increases as the capture interval increases and (2) the CPU is more heavily loaded as the capture interval decreases. If the capture interval rises above 30 ms, the play-out delay linearly increases by a factor of two. However, the minimum play-out delay levels off at 40 ms.

Figure 5 shows the measured end-to-end latency when the capture interval is 10 ms with varying play-out delays. In a LAN environment the network transmission delay was less than 1 ms and the capture interval and play-out delay would be expected to dominate the end-to-end delay. Our expectation is confirmed in Figure 5 where the play-out delay is greater than or equal to 70 ms. The small distance between the two curves is caused by the network delay and system overhead. For play-out delays in the range of 40 - 60 ms an additional 10 ms delay is introduced whose cause still under investigation.

From these two figures we find that the optimal capture interval and the play-out delay are 10 ms and 40 ms, respectively. Thus, the minimum one-hop audio delay is 60 ms in a LAN environment. Accordingly, any multi-hop end-to-end audio delay can be roughly represented as *minimum one-hop audio delay (= 60 ms) + end-to-end network delay*. Note that placing mixing modules at intermediate relay nodes may add additional processing delays.

Next, we also measured the end-to-end audio delay in a WAN environment between the USC campus in Los Angeles and Information Sciences Institute in Arlington, Virginia. We placed a loop-back module at the east coast site to receive audio packets and reflect them back to the sender. The average round trip time (RTT) was measured at 70 ms with a small variance. The end-to-end audio delay in the WAN environment exactly matches that of the LAN environment plus the one-way network transmission delay.

These encouraging results show the feasibility of a medium-sized application-level audio chat service in a commodity Windows environment. Similar results have also been confirmed by other research groups [9, 10].

4 Related Work

Our system integrates the following components to create an efficient, easy-to-use, scalable live audio chat service: web-based session management, shared tree based application-level multicast delivery, live recording and retrieval, speech-to-text translation service, and audio mixing. There exists related work in the audio mixing and multicast area, but to the best of our knowledge no system with an integrated feature set like ours has been reported.

Hierarchical audio mixing architectures [13] place all participants at the leaf nodes and locate the mixers at non-leaf nodes. A mixer relays the audio coming from its children to the parent after merging the streams. A root mixer broadcasts the mixed audio to all participants through the IP multicast network. Several commercial products such as the *ClickToMeet* conference product [1], formerly CUseeMe, construct an efficient delivery path (full-mesh or multicast) among the mixers. Because most of the commercial products are integrated with video conferencing applications, they focus on video delivery and cannot fully take advantage of audio mixing. In end-system mixing architectures [12], one of the participants functions as the mixer. Additionally, most of the above cited systems do not consider how to build efficient delivery paths among mixers and participants.

Application-level multicast protocols providing many-to-many connectivity can be classified as mesh based, tree based, cluster based and distributed hash table (DHT) based architectures. An example of a mesh based architecture is Narada [8], where each node constructs a single-source multicast tree from the mesh structure. Due to its centralized nature, Narada does not scale well. Tree based architectures include Yoid [6], HMTP [14] and our proposed YimaCast. NICE [2] was developed as a hierarchical architecture that combines nodes into clusters, then selects representative parents among these clusters to form the next higher level of clusters, and so on. DHT based architectures use hashing mechanisms to generate node identifiers such that nodes close to each other logically have similar node identifiers. Subsequently a multicast tree is created on top of this substrate. Pastry/Scribe [4] and CAN/CAN Multicast [11] are based on this concept. Even though these systems are designed to support a large number of users, all the previous application-level multicast protocols fail to recognize the potential for constructing a more efficient topology if the aggregation of audio packets along the delivery path is allowed.

5 Conclusions and Future Directions

We have presented our design and implementation of a multiuser audio chat system using the peer-to-peer based YimaCast protocol. Preliminary results are encouraging and we are planning further real world tests and deployment in the coming months. We also intend to extend our work in several directions.

First, silence detection can improve the audio mixing process by removing the noise floor that accompanies with every open microphone. Second, our adaptive QoS algorithm that migrates nodes based on their active speaker status requires further investigation and performance tests.

References

[1] Click to meet, http://www.fvc.com/.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast, 2002.

[3] M. R. Carey and D. S. Johnson. Computers and intractability. *W.H. Freeman Co. New York*, 1979.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[5] T. Cormen, C. Leiserson, and R. Rivest. Introduction to algorithms. *MIT Press*, 1997.

[6] P. Francis. Yoid: Your own internet distribution.

[7] R. Gallager, P. Humblet, and P.Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, pages 66–77, January 1983.

[8] Y. hua Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *ACM SIGCOMM 2001*, San Diago, CA, Aug. 2001. ACM.

[9] K. Koguchi, W. Jiang, and H. Schulzrinne. QoS measurement of VoIP end-points. In *IEICE Group meeting on Network Systems*, December 2002.

[10] K. MacMillan, M. Droettboom, and I. Fujinaga. Audio Latency Measurements of Desktop Operating Systems. In *International Computer Music Conference (ICMC'01)*, September 2001.

[11] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Third International Workshop Networked Group Comm.*, November 2001.

[12] K. Singh, G. Nair, and H. Schulzrinne. Centralized Conferencing using SIP. In *Internet Telephony Workshop*, April 2001.

[13] H. Vin, P. Rangan, and S. Ramanathan. Hierarchical conferencing architectures for inter-group multimedia collaboration, 1991.

[14] B. Zhang, S. Jamin, and L. Zhang. Host Multicast: A Framework for Delivering Multicast to End Users. In *Proceedings of IEEE Infocom*, New York, June 23-27, 2002.