SuiteSound: A System for Distributed Collaborative Multimedia[*]

John Riedl
Vahid Mashayekhi
James Schnepf
Mark Claypool
Dan Frankowski [†]

April 28, 1993

Keywords: Teleconferencing, Collaborative Systems, Multimedia, Distributed Systems, Silence Deletion, Silence Detection, Continuous Media, Digital Audio, Ethernet

Abstract

Multimedia has the same potential to revolutionize human-computer interfaces that bitmapped workstations have realized over the last decade by providing a more familiar environment to the users. Achieving this potential requires the development of programming environments with integrated support for multimedia. *SuiteSound* is one such environment. SuiteSound is built in the *Suite* object-based system on a conventional UNIX operating system. SuiteSound objects incorporate multimedia by creating *flows* and *filters*. Flows are streams of multimedia data moving through a sequence of objects. They bridge the gap between objects representing the state of an entity at a discrete point in time and space and continuous media such as live audio or video. Filters are intermediate objects between the source and destination of a flow. They take a flow as input, perform one of several operations such as multiplex-in, multiplex-out, gain control, or silence deletion on it, and send the resulting flow to its destination. In effect, they provide a virtual device interface for the application programmer that is uniform and independent of any physical device. This paper describes the design and implementation of SuiteSound on the Sun SparcStation. We perform experiments to determine the network and CPU load of the sound tool, detail experiences using the SuiteSound environment and applications, and suggest future work.

1 Introduction

Computer Supported Collaborative Work (CSCW) is the study of methods for enhancing cooperation among computer users by providing tools that explicitly support user interaction and sharing of information [1, 2]. Suite is an object-based system with powerful tools for developing distributed collaborative applications [3, 4]. This paper describes SuiteSound, a tool for manipulating multimedia flows, such as digital audio, in Suite.

SuiteSound supports development of collaborative applications using multimedia by integrating flexible, easy-to-use digital audio with a flexible, easy-to-use object-based system. SuiteSound applications readily combine live or recorded audio with editable representations of user objects. This paper describes the design, implementation, and early experience with SuiteSound, along with some sample applications.

[*]We gratefully acknowledge the support of the National Science Foundation (grant number $NSF/IRI - 9015442$) and the research funds of the Graduate School of the University of Minnesota.

[†]Department of Computer Science, University of Minnesota, Minneapolis, MN (612)-624-7372

1.1 Previous Work

There are many applications for multimedia in collaborative applications. Multimedia electronic mail systems, such as Diamond [5], ease communication between participants, especially those who are not used to using computers. Teleconferencing systems improve synchronous interaction, when traveling is not desirable. Distributed education is also possible, with students at remote sites seeing video images of the instructor, and participating in the class through sound transmitted back to the classroom. This section briefly describes several of the successful multimedia environments for collaborative work.

The Etherphone system supports locally distributed computing environments with multiple workstations and multiple networks [6]. In the Etherphone system, each workstation is associated with an Etherphone that digitizes, packetizes, and encrypts the voice and transmits it over the Ethernet. The voice manager in the Etherphone system provides facilities for recording, editing, and playing stored voice.

Collaborative multimedia has already proven useful for distributed decision making. For instance, the Mermaid system is designed for collaborative decision-making with multiple remote participants [7]. Data, voice, and video are transmitted using N-ISDN channels with 64Kpbs per channel. The voice and data use one B-channel and video the other. Users can talk freely, with voices mixed and transmitted to all users simultaneously. In Mermaid, synchronization, routing, and distribution are done separately for voice, video, and data. This allows all three to run at high speeds, but makes the synchronization of the media more difficult.

Researchers are also focusing on developing sharable audio servers for supporting multimedia applications. Angebarnndt et. al describes a client-server model that uses an audio server for supporting shared access to audio hardware, introduces a protocol that provides applications with a device-independent interface to the audio capabilities of the workstation (virtual devices), and provides primitives for synchronizing the various media components [8]. Terek and Pasquale describe an extension to the X Window System [1] to support audio [9]. The modified X11 server accepts audio requests from across the network and executes them on Sun SparcStations.

1.2 Overview

Our work extends previous research in four ways. First, SuiteSound provides an object-oriented interface for building applications that integrate multimedia with traditional objects, promoting simplicity and re-usability. Second, our work supports multimedia flows with multiple sources and multiple sinks, and different ways to merge or split the flows. Third, we are investigating methods of concurrency control that allow end-users to effectively control multiple multimedia streams at their workstations. Fourth, we are performing experiments to measure the costs of different implementation strategies for multimedia systems on conventional operating systems.

[1]The X Window System is a network transparent window system developed at MIT.

Section 2 describes issues in incorporating multimedia in an object-based system on a conventional operating system, and describes our prototype implementation. Many of these issues are independent of the particular system we have chosen for our implementation. Section 3 introduces three applications: *Teleconf*, *Annotator*, and *CSI*. Section 4 describes a number of experiments aimed at measuring the network and CPU load of Teleconf, compares the performance of two silence deletion algorithms, presents formulae for measuring work done at each local workstation, estimates the percentage of dropped sound bytes, and shows a breakdown of sound packet sizes. Section 5 summarizes the lessons learned in working with SuiteSound. Finally, section 6 presents our conclusions and suggests directions for future work based on our experiences

.

2 Design and Implementation of SuiteSound

This section presents an object-based design for continuous data passing through discrete objects, introduces Suite, an object-based system for building distributed applications, presents the implementation of SuiteSound, explains the recording and playing operations in SuiteSound, and gives an example of a filter built into the sound tool for removing silence.

Our goal is to extend Suite with support for continuous multimedia, such as live sound, live video, or animations. We are particularly interested in support for collaborative applications, so our focus is not on database techniques for storing multimedia, but on active techniques for processing the data. Specifically, we intend to develop a system that supports simple specification of multimedia processing and easy integration with other collaborative applications. The continuity and real-time requirements of multimedia processing can make applications difficult to write. We want infrastructure to handle the complexity, while allowing programmers the flexibility to create the applications they need.

2.1 Continuous Media in Discrete Objects

Object-based systems are being used for many tasks because of the benefits of improved encapsulation [10]. However, objects are inherently discrete, conceptually containing the representation of the state of a system at a fixed point in space and time. Continuous media, such as voice and video, flow across time, with no convenient boundaries for object representation. Objects can be used to represent discrete parts of the flow, but do not readily represent the entire flow.

We propose breaking continuous media into chunks. Rather than using individual objects to represent these chunks, individual objects represent points in space past which the chunks must flow over time. In our design, continuous media is processed through *flows* and *filters*. Flows represent connections between objects through which continuous media can move from a source to a destination. Filters are objects that take flows as input, process the flow, and produce a modified flow as output. Filters provide a virtual device interface on top of physical audio devices. Filters modify the functionality of the virtual device, but provide the same interface, so additional filters can be applied to the modified flow. A filter performs an operation on a flow traveling from a source to a destination. Filters can combine flows or create new flows as part of an operation. Possible filters include **`multiplex-in`**, **`multiplex-out`**, and **`discard silent periods`**.

Flows carry data from the output of one virtual device to the input of another. Programmers build applications by instantiating filters and joining them with flows, much as UNIX programmers create new tools from UNIX utilities by joining old tools with pipes. Once a flow is established, it moves automatically through the filter. The permanent state of the object represents a brief interlude (usually 1/4 to 1 second) in the continuous flow of the media.

Flows are constructed as a series of filter objects linked together, with a source at one end and a sink at the other. In most applications either the source or the sink is chosen to provide the real-time clock rate for the flow. The designated clock source issues callbacks to the next link in the flow, delivering data if the clock source is also a data source, and requesting data if the clock source is a data sink. These method invocations from an object to an application filter are named *callbacks* because the method to be invoked is passed to another object, which "calls-back" the original object. Callbacks are similar to Swift's *upcalls* [11] in that they are invocations from a lower layer of a protocol stack to a higher layer. Within an application, callbacks are used to pass the data from one filter to the next in the flow, until it arrives at the sink.

2.2 Suite: Distributed, Persistent Objects

SuiteSound is being developed in the Suite environment. Prasun Dewan and his research team designed and developed Suite, a multi-user interface generator, at Purdue University [12]. They implemented a Suite prototype on top of UNIX [2], TCP/IP, NFS [3], and X [3]. Suite supports remote procedure calls (RPC), active persistent data, and management of "direct manipulation" user interfaces. The Suite object model is an extension of UNIX, allowing distributed, shared, protected and persistent objects.

2.3 Sound on the Sun SparcStation

The Sun SparcStation has a physical audio device that plays and records at 8000 bytes per second. Each byte represents a single 13 bit Pulse Code Modulated (PCM) sample from the physical device, compressed to 8 bits through μ-law companding. SunOS 4.1.1 provides a library of functions for operating on the μ-law data.

SuiteSound is compatible with any interface based on the general framework of a device to which sound blocks can be read or written, allowing one SuiteSound interface to be implemented on many physical devices.

[2]UNIX is a trademark of AT&T.

[3]Network File System is a trademark of Sun MicroSystems.

2.4 SuiteSound

SuiteSound uses a separate object to manage the record and play components of the sound device. These objects interact with the sound device, and present the filter virtual device abstraction to filter objects. The sound objects deliver or receive sound in two modes: synchronous or asynchronous. In synchronous mode, the sound object buffers sound and delivers or plays it when requested by its successor filter in the flow. Synchronous mode is useful when some other filter in the flow is delivering data at a physical clock speed. In asynchronous mode, the sound object uses a timer to periodically deliver or request sound data. When the timer expires, the sound object issues a callback to the object that initiated the flow. In asynchronous mode, a filter provides data at a real time rate.

Filters in a flow operate in push or pull mode. A filter pushes data by requesting that its predecessor operate asynchronously, and its successor operate synchronously. In response to each message from its predecessor, the filter pushes the data to the waiting successor. A filter pulls data by requesting that its predecessor operate synchronously and its successor operate asynchronously. In response to each message from the successor, the filter requests more data from its predecessor to send to the successor. A filter can change from synchronous to asynchronous operation by just changing the calls that connect it to its neighboring filters. Usually all filters in a flow operate in the same mode.

2.5 Silence Deletion

Silence deletion can be thought of as a filter accepting flows at one end, removing silence from, and sending them to the destination. Application developers access silence deletion through the filter library.

The problem of silence detection and deletion has been studied in various domains, including natural languages, multimedia systems, telephone systems, and network systems. Watanabe explores the development of a machine that adapts its conversational speed to the particular speech pattern of a speaker by detecting silence [13]. Kashorda and Jones show that muting of transmitters during periods of silence increases the voice capacity of a microcellular cordless telephone system [14]. The MAGNET system uses adaptive protocols for dynamically controlling the network load requirements by detecting silence in speech or lack of motion in video [15]. These projects demonstrate the feasibility of silence deletion and detection. Our project aims at reducing the network traffic experienced in unfiltered Free mode by silence removal.

There are many possible algorithms for determining non-silent periods. Most are based on watching the linear audio signal for sections of pre-determined length with non-zero energy. Since most energy comes from voiced speech, this may do poorly for sibilants or fricatives at the beginning or end of a word [16]. Our algorithms are based on measures of energy, but do not seem to have such problems, perhaps because of relatively large chunk size we used.

Our silence deletion algorithms do not prefilter any frequencies, so the audio signal has background noise. For this reason, instead of looking for non-zero energy, the algorithms look for energy above a certain threshold. The algorithms are somewhat sensitive to the choice of threshold, but noise levels are low enough that speech may readily be distinguished from silence.

We present two algorithms for detection and deletion of silence both based on measures of energy:

Differential - This algorithm finds a byte nonsilent if it is sufficiently distant from the previous byte as defined by some minimum threshold. Distance is the absolute value of the difference between the linear values of two bytes. The idea is that if an audio signal has energy from voiced speech, it will have much larger oscillations than a silent signal.

HAM - This algorithm finds a chunk of bytes nonsilent if the average energy is above a minimum threshold. The chunk size used for our experiments is 4 ms (32 bytes). The average energy is the sum of the absolute values of the linear values of the bytes in a chunk [16]. The idea is that if an audio signal has energy from voiced speech, the average energy of a chunk will be much higher than that of a chunk from a silent signal.

HAM is less sensitive than the Differential method to small spikes in the audio signal since it averages over a chunk of bytes. In order that these algorithms not delete silence from between words or phrases, each detects 400 ms of silence before starting to discard bytes or chunks. After 400 ms, bytes or chunks are discarded until nonsilence is again detected.

2.6 Floor Control

Floor control in a collaborative system comprises a set of policies to control access to the set of shared objects. Floor control policies usually permit or deny access to all the objects in the collaboration simultaneously, conceptually controlling the entire "floor" at once. More general concurrency control policies separate the objects into domains that can be accessed individually [17]. Floor control can be important in collaborative multimedia to structure the interaction and reduce bandwidth. SuiteSound is designed to support arbitrary floor control. In this paper we discuss two floor control policies: first-in, first-out (FIFO), in which participants are queued in FIFO order, and *free*, in which there is no floor control.

3 Applications

This section describes the design and implementation of three SuiteSound applications: Teleconf, Annotator, and CSI.

3.1 Teleconf

Teleconf is an audio teleconferencing tool built on SuiteSound. Teleconf has objects that manage floor control and sound delivery. These objects interface with SuiteSound objects to play and record the sound.

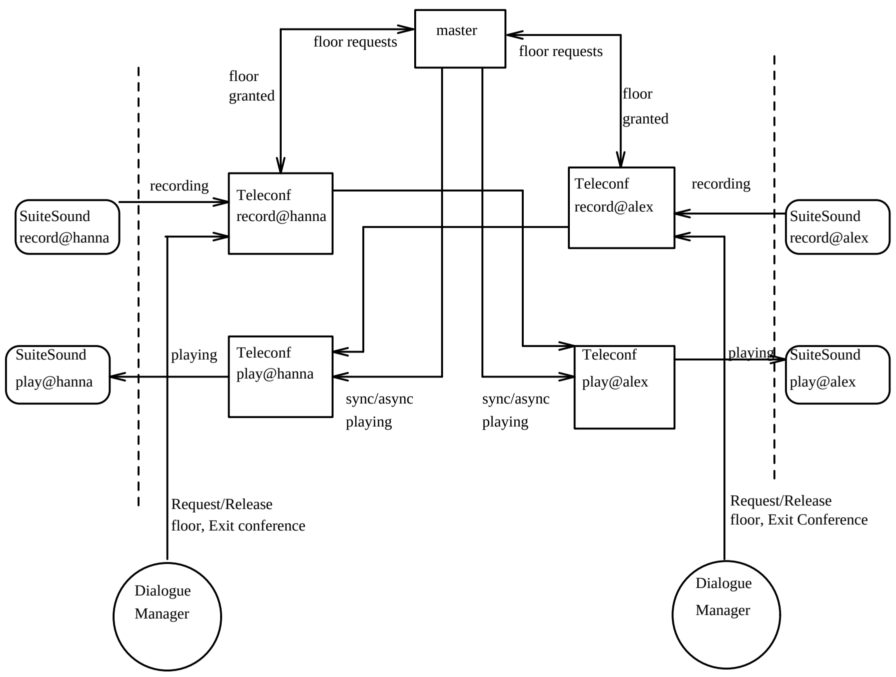

Figure 1: Object interconnections for the Teleconf teleconferencing tool

Figure 1 depicts the interconnection among SuiteSound objects during a two-person teleconference. Each workstation runs two SuiteSound processes that manage the record and play capabilities of the audio device. In addition, each user has two Teleconf objects, one for routing sound to other users, and one for receiving sound from other users. A Master object manages floor control, not processing any sound itself, but informing the Teleconf objects when they should transmit or receive sound.

Teleconf currently supports two floor control modes: FIFO and no floor control (Free). In FIFO mode, participants queue to obtain permission to transmit sound. Only one station transmits sound at any given time. In Free mode, each station can transmit sound to every other station. Thus, the Teleconf objects have multiple incoming audio data streams. They convert the data to linear form and add any other data received since they last called their SuiteSound objects.

3.2 Annotator

Annotator is a single-user application built on SuiteSound for recording and playing back voice annotations. The user enters voice entries into a variable sized array. The array stores a voice-entry-id, a brief description string and the approximate length of the annotation. Voice annotations are persistent; user's can modify and replay them as needed. Annotator supports operations that change the content of the sound annotation. These include: `Record, Concatenate`, and `Replace.` Annotator also supports operations that do not change the content of the sound annotation. These include: `Play`, `Loop (replay)`, `Sort`, and `Length`.

The CSI tool (see section 3.3) relies heavily on the Annotator.

3.3 Collaborative Software Inspection

The Collaborative Software Inspection tool (CSI) supports software inspection in a distributed, collaborative environment [18]. Software inspection activities can be thought of as asynchronous and synchronous. Preliminary results suggest, that given the proper audio support, distributed collaborative software inspection can be as effective as traditional software inspection.

The asynchronous activities are those that can be done separately by the participants. These are the distribution of target material and the creation of the correlated fault list. Reviewers view the lines of target material through the Browser object and annotate the lines through a sound or textual annotator. The annotations help the producer create the correlated fault list. The software inspectors use the fault list during the synchronous part of the inspection.

The synchronous activities are those that require all participants to work at the same time. These are the discussion of the correlated faults, the agreement on the faults, the recording of the action items, and the determining the status of the inspection. Software inspectors use Teleconf in conjunction with CSI to support verbal interactions. CSI displays the recorded material on the all of the participants' screens. The recorder enters action items into an action list visible to all participants. The recorder also maintains information on the attendees, status of the meeting and the final decision made concerning the target material being inspected. CSI automatically records system activity such as loading of files and objects, updating of data structures in objects, and unloading of objects for future statistical studies.

4 Experiments

This section describes several experiments in Teleconf. Our primary goals are to assess Teleconf performance under Free mode and under two different silence deletion algorithms. Performance is based on network and CPU loads.

4.1 Goals of Experiments

The goals of our experiments in Free mode are to:

1. Assess the feasibility of teleconferencing on Ethernets for different number of participants.
2. Compare the network load of Free mode with and without silence deletion.
3. Compare the two silence deletion algorithms in terms of network and CPU loads.

4.2 Methodology

Our experiments involved four Computer Science graduate students at the University of Minnesota. These students engaged in conversations using Teleconf on a 10 Mbps Ethernet and Sun IPX Workstations. The students received Teleconf training prior to conducting the experiment. We held ten sessions, varying the number of participants and the method of silence deletion used.

We used **`etherfind`** to observe the network load generated by Teleconf. **`Etherfind`** watches packets between any specified pair of machines and keeps track of the number of bytes sent or received in 1/100 second intervals. The **`etherfind`** output for each packet contains a timestamp, number of bytes in the packet, protocol, source workstation, destination workstation and the sending and receiving ports.

To measure CPU load, we followed the method described in Lazowska et. al.'s paper [19]. We ran a process in the background that incremented a counter, and compared speed with and without Teleconf. The counter was run in conjunction with Teleconf, supporting 1, 2, 3, and 4-person conversations, for the same time interval. We repeated the runs five times in each case. The average ratio between the final result of the counter values, with and without Teleconf running, is a coarse estimate of the CPU that Teleconf uses and how much processing resources are left for other applications.

We used a computer lab that was lightly-loaded during vacation. We were the only users of the computers used in the Teleconf sessions, and network traffic from other users was extremely light.

We centered the discussions on proposed revisions of this paper. Discussions were seven to eight minutes in length. We based our results on five minutes from the middle part of the trace files, to evaluate the stable part of the conversation.

We performed experiments in unfiltered Free mode, Free mode with a HAM silence deletion filter, and Free mode with a differential silence deletion filter. We used combinations of 2, 3 and 4 participants.

The dependent variable is the network load, in terms of the number of bytes per second (average, peak and standard deviation), and number of packets per second (average, peak and standard deviation).

4.3 Results

The following tables and figures show the results of our experiments:

Function	**None**	**1-person**	**2-person**	**3-person**	**4-person**
Average	196301085	178768415	157235246	142793487	127771036
STD	1510579.74	4331539.91	2013402.57	768557.559	3172823.03
% CPU Left	1	0.9106848	0.8009902	0.72742077	0.65089317
% CPU Used	0	0.0893152	0.1990098	0.27257923	0.34910683

Table 1: Counter values without Teleconf and with 1, 2, 3, and 4 participants in Free mode.

Method	**Function**	**2-person**	**3-person**	**4- person**
Unfiltered	Mean	17361.21	51847.01	102310.62
	Std Dev	4308.96	9678.85	18501.79
	Peak	33614.00	83044.00	151542.00
differential	Mean	7922.64	15371.11	23847.19
	Std Dev	3375.19	9323.52	12350.46
	Peak	18883.00	45574.00	83829.00
HAM	Mean	7437.39	12984.68	20864.92
	Std Dev	2943.44	8137.69	12748.07
	Peak	17094.00	38836.00	78256.00

Table 2: Network load in bytes per second in Free mode unfiltered and with HAM and differential silence deletion filters for 2, 3 and 4 person conversations.

CPU Usage - Table 1 shows the counter values without and with Teleconf running with 1, 2, 3, and 4 participants in Free mode.

Results of unfiltered Free, differential, and HAM - Table 2 gives a summary of the raw data we obtained during our experiments.

4-person conversations - Figures 2, 3, 4 show the total number of bytes in one-second intervals for a 4-person conversation in unfiltered free mode and with the HAM and differential silence deletion filters.

Mean load in bytes - Figure 5 shows the mean network load in bytes for the unfiltered Free, and with differential, and HAM for 2, 3 and 4 person conversations.

4.4 Analysis

This section analyzes the results given in the previous section.

CPU Usage: Figure 6 shows the projection of CPU usage for 10 participants based on the experimental results for 1, 2, 3, and 4 participants in unfiltered Free mode. CPU costs become prohibitive after 7 or 8 participants, not allowing any other applications to run. Preliminary profiling indicates silence deletion may reduce total CPU load.

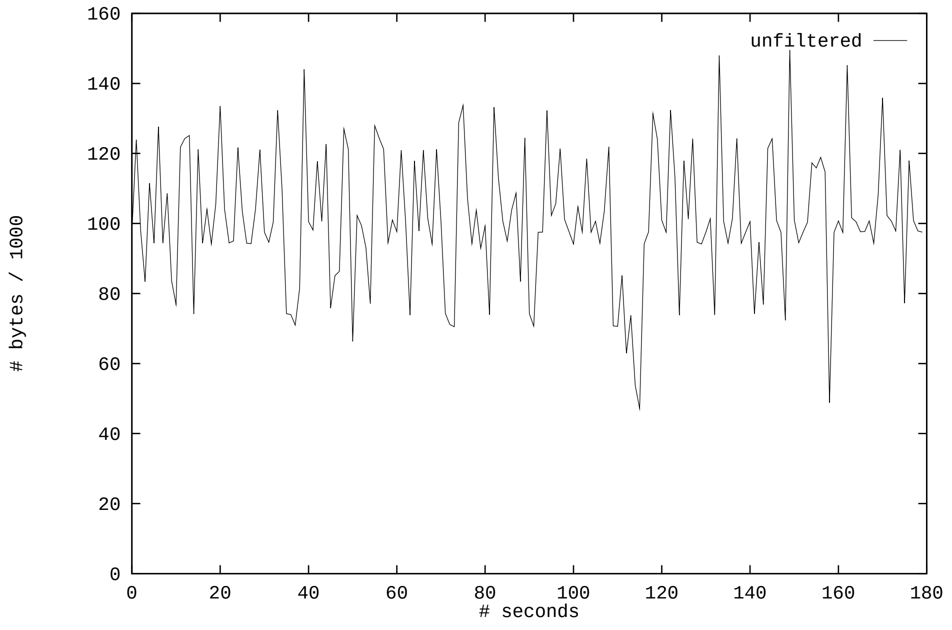

Figure 2: Number of bytes per second in a 4-person conversation in unfiltered Free

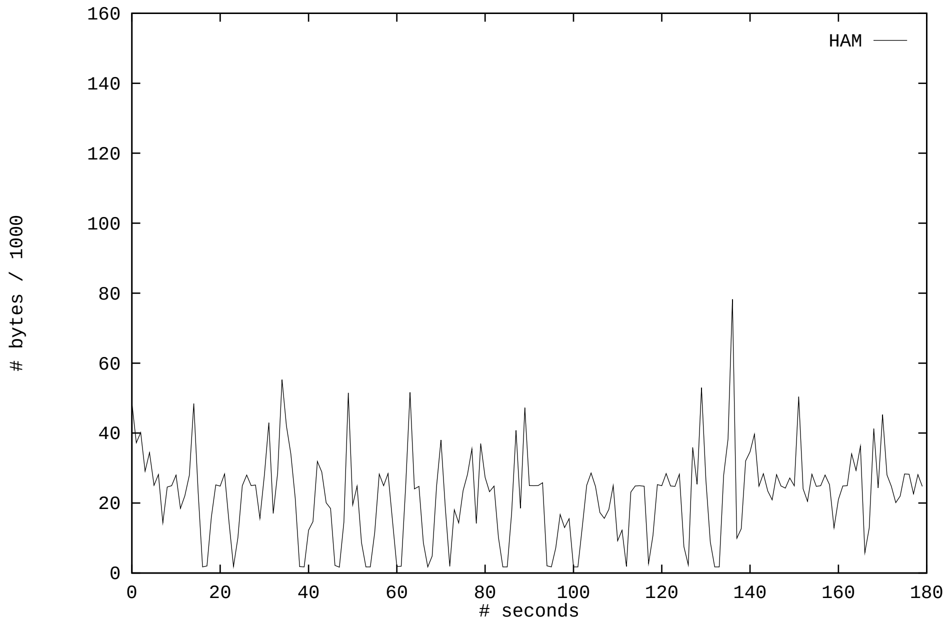

Figure 3: Number of bytes per second in a 4-person conversation in Free mode using HAM

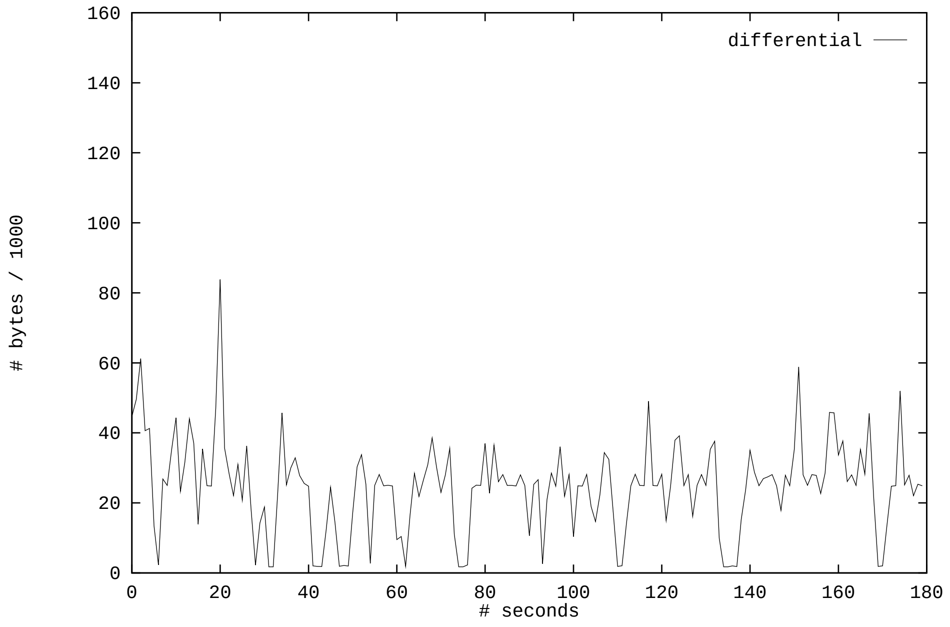

Figure 4: Number of bytes per second in a 4-person conversation in Free mode using differential

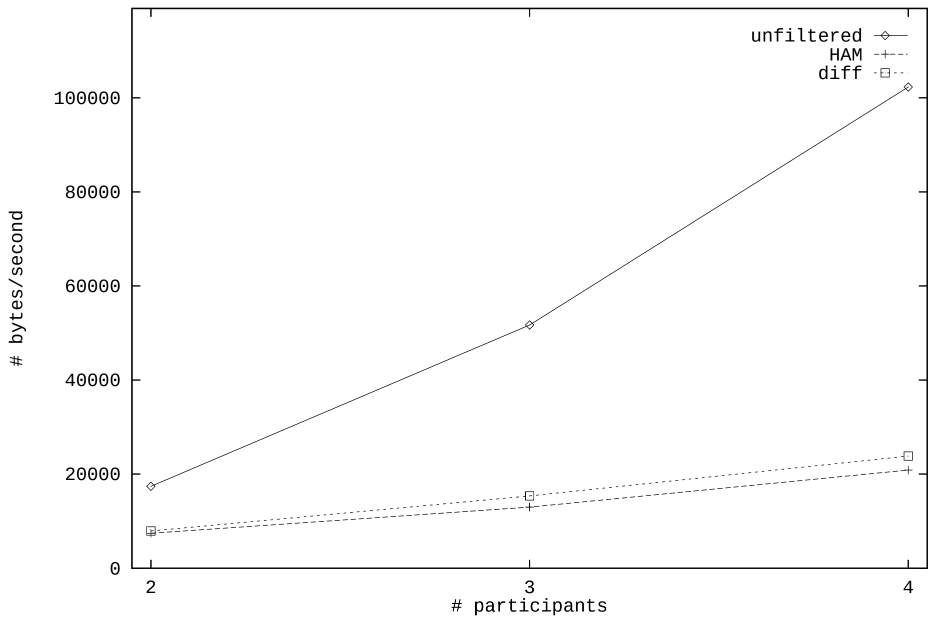

Figure 5: Mean network load in bytes per second for unfiltered, HAM and differential with 2, 3 and 4 participants

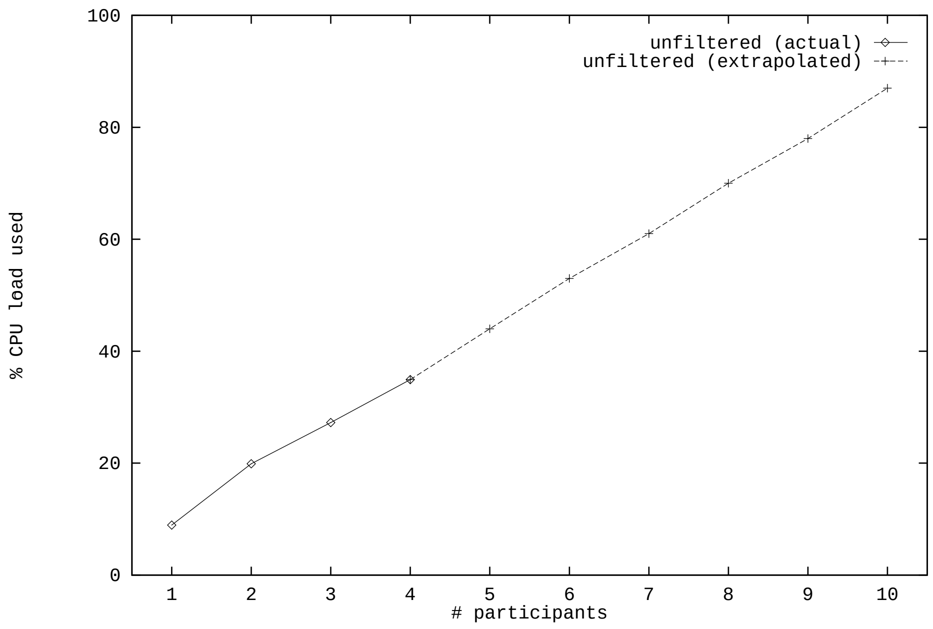

Figure 6: Extrapolation of the percentage of CPU used by Teleconf to 10 participants

Comparison of Unfiltered Free, differential, and HAM:

1. Figures 2, 3, 4, and Table 3 show there is far more variability in the filtered results. We expect this since when no one is speaking, the load generated is much smaller, and when speech is present, the load will be very close to the unfiltered Free mode.

2. The mean bytes for 2, 3 and 4 conversants show the silence deletion algorithm significantly reduces network traffic. The difference increases as the number of participants increases as shown in figure 5. We expect this since typical conversation has only one speaker at a time; the others are listening and generating silence. In the unfiltered Free mode, increasing the number of participants increases the amount of silence that is sent over the net.

3. In the 4-person conversations, the peak load for a 1 second interval was 151 kbytes/sec for the unfiltered Free mode, 78 kbytes/second for HAM and 84 kbytes/second for differential. The peak load for HAM and differential is close to half that of the unfiltered Free mode.

Extrapolated network load: Since the load in unfiltered Free mode is constant for each participant, the total offered load is dependent on the number of participants. The sound generated by each of n participants is sent to $n-1$ other participants. The total network offered load is: $LOAD(n) = k \times (n) \times (n-1)$ where k is the amount of load placed on the network by one person sending messages and getting acknowledged. However, in a typical conversation there is only one person talking at a time. With a silence deletion filter,

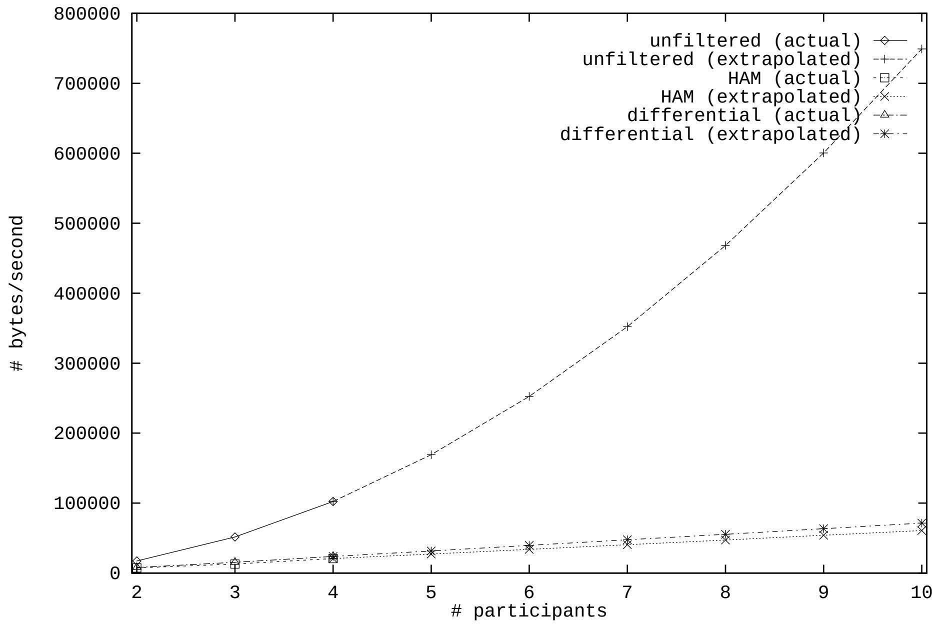

Figure 7: Extrapolation of network offered load to 10 participants

	No Multicast	**Multicast**
unfiltered	$LOAD(n) = k \times (n) \times (n-1)$	$LOAD(n) = k \times (n) \times (1)$
filtered	$LOAD(n) = k_1 \times (1) \times (n-1)$	$LOAD(n) = k_1 \times (1) \times (1)$

Table 3: Derived formulae for network offered load

we are effectively reducing the load to O(n); at any point in time, only one person is putting load on the network. The total network offered load is then: $LOAD(n) = k_1 \times (1) \times (n-1)$ where k_1 is a reduced load put on the net by the participant who is currently talking. In practice a real Ethernet will not be able to deliver this offered load, because of increased contention, but the projection suggests upper bounds on the number of participants in a teleconference.

For $n = 2, 3, 4$ solving for k with our observed load, we consistently obtain a value of about 8600 bytes/second. Solving for k_1 in the case of HAM and differential, we obtained values of 7000 and 7900 respectively.

This result allows us to extrapolate the network offered load to conferences with more than 4 participants. Figure 7 shows our estimate of the increase in network load with more participants. The unfiltered Free load increases quadratically. The results show that the silence detection and deletion filter reduces the network load from O(n^2) for unfiltered Free to O(n) for the filtered conversations.

The use of silence deletion filters is compatible with hardware multicasting. Our projections of the load with and without silence deletion for multicast and uni-cast are shown in table 3.

Lost Data: We investigated whether the audio device was keeping up with the task of reading from the microphone. The SuiteSound record object reads the data from the audio device every 500 milliseconds (twice per second), reading approximately 4096 bytes of data. SuiteSound buffers this data until the Teleconf record object uses a "callback" routine. If the sound device overflows the buffer, ome of data may be over-written. How much data is lost this way?

Per minute of conversation, the actual bytes stored in the buffer averages to $467K$ bytes, while the total bytes that should have been read averages to $490K$ bytes. This is a difference of $23K$ bytes or approximately 4.5 percent of the total. We find this percentage of loss to be acceptable.

5 Lessons Learned

This section describes a number of lessons we have learned from using a teleconferencing tool in a workstation environment. The study of these lessons motivates the future work described in section 6.

Floor Control: First-in, first-out (FIFO) floor control was found to be too restrictive. No floor control (Free mode) used much more CPU and network resources. Silence deletion reduced the resource usage in Free mode to almost the same as strict FIFO floor control.

Delay: Delays in transmitting the speech were noticeable, but acceptably small on a local-area network.

User interface: Because Teleconf is often used in support of other applications, feedback to the user should not distract from the other applications. We found audio feedback more helpful than visual feedback.

Collision detection: In Free mode the speech of two speakers speaking simultaneously is merged. It was often possible to identify the speakers in merged speech, which was an advantage for later floor control negotiation.

Use of Headsets: We used headphones and lapel microphones for all silence deletion. Using speakers created feedback into the microphones. In pilot tests we have used headsets with built-in microphones, and these were also effective. Lower quality microphones were significantly noisier, making silence deletion less effective.

Limitations of the Supporting Environment: The UNIX process model does not currently support multiple threads of control within an address space. This limitation means that execution of methods within an object can interfere with the execution of other methods. The most serious problem we discovered as a result of this limitation is that flows with a cycle in them will sometimes fill up the network streams used by Suite for communications, resulting in deadlock. To avoid this problem, our applications do not construct cyclic flows. For instance, the Teleconf play and record objects are separate to avoid creating a false cycle.

None of these problems would exist in an operating system that supports multiple threads of control in each address space [20, 21].

Silence Deletion: There is a substantial amount of audio data that does not need to be sent. The silence deletion algorithms remove a significant portion of the flow of sound without affecting the quality. These algorithms reduce network load substantially, and appear to also reduce CPU load.

6 Conclusions and Future Work

SuiteSound has proven a convenient tool for building object-based applications that include audio. The mechanism provided has made it relatively easy to build audio applications, and has proven flexible enough to support every application we have attempted. The flow model is convenient, especially since applications can initiate a flow and then continue with their other work while the flow passes through asynchronously. Suite has been a good platform for building collaborative applications, and the SuiteSound extension enhances these applications.

Our new design for SuiteSounds provides high-level specification of flows based on a library of filters. Filters can be dynamically arranged into processes to reduce communication overhead. We expect an implementation of the new design to improve performance and make flows easier for programmers to create and modify.

Our experiments show the CPU and network load of live audio using SuiteSound under various conditions. Our results indicate that (1) Unfiltered Free mode imposes a high CPU and network load; (2) The use of silence deletion filters is an important factor in reducing network and CPU load; (3) A controlled floor model, such as FIFO, greatly improves network traffic by reducing the number of links that each workstation has to support; (4) The differential method for silence deletion reduces an average conversation to 0.44 percent of its original size, while HAM reduces an average conversation to 0.34 percent of its original size.

We suggest a rich set of problems for future work, based on our experience with SuiteSound.

Sound as a First-Class Object: Integrate sound in a collaborative infrastructure as a first-class object. Programmers should build applications using multimedia in the same way they build applications using conventional media. Users should have the same support for access control, concurrency control, undo/redo, and coupling for all media types.

Graphical Specification: Build a visual programming language for flow specification [22]. Programmers could construct flows by drawing connections among filters from an iconic toolkit. Running flows could show sound moving among the filters visually.

Quality of Service: Develop a quality of service specification for the flow model. Our experiments have shown the importance of low latency in maintaining effective audio communications. Measure the improvements in audio quality over wide-area networks using the quality of service specification.

Experiments: Develop an objective measure of speech quality that reflects the trade-offs in workstation-supported audio. Perform experiments to study the CPU and network tradeoffs in various floor control policies, using silence deletion, compression, and encryption. Repeat the experiment for wide-area networks, using synchronization techniques as necessary.

References

[1] C. Ellis, S. Gibbs, and G. Rein. Groupware. *Communications of the ACM*, January 1991.

[2] J.M. Tazelaar. In-depth groupware. *Byte magazine*, December 1988.

[3] P. Dewan and E. Vasilik. An object model for conventional operating systems. *Usenix Computing Systems*, December 1990.

[4] P. Dewan and R. Choudhary. Flexible user interface coupling in a collaborative system. *Proceedings of the ACM CHI's 91 Conference*, April 1991.

[5] H.C. Forsdick and R.H. Thomas. The design of Diamond: A distributed multimedia document system. Technical report, TR number 5402, Bolt Beranek and Newman Inc., October 1982.

[6] D.B. Terry and D.C. Swinehart. Managing stored voice in the Etherphone system. *ACM Transactions on Computer Systems*, February 1988.

[7] S. Sakata. Development and evaluation of an in-house multimedia desktop coneference. *IEEE Journal on Selected Areas in Communications*, April 1990.

[8] Susan Angebarnndt, Richard Hyde, Daphne Luong, Nagendra Siravara, and Chris Schmandt. Integrating audio and telephony in a distributed workstation environment. In *Proceedings of the Summer 1991 USENIX Conference*, pages 419–435, Nashville, TN, 1991.

[9] Robert Terek and Joseph Pasquale. Experiences with audio conferencing using the X, Window system, UNIX, and TCP/IP. In *Proceedings of the Summer 1991 USENIX Conference*, pages 405–418, Nashville, TN, 1991.

[10] P. Wegner. Dimensions of object-based language design. In *OOPSLA Proceedings*, October 1987.

[11] D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM symposium on operating systems principles*, December 1985.

[12] P. Dewan. A guide to Suite. Technical report, SERC-TR-60-P, Software Engineering Research Center at Purdue Univesity, February 1990.

[13] T. Watanabe. Adaptation of machine conversational speed to speaker utterance speed in human-machine communication. *IEEE Transactions on Systems, Man and Cybernetics*, 20(2):502–507, March 1990.

[14] A. Kashorda and E.V. Jones. A spectrum efficient technique for cordless telephone access to ISDN. In *Fifth International Conference on Mobile Radio and Personal Communications*, pages 15–19, London, UK, 1989.

[15] C. Gan and R.W. Donaldson. Adaptive silence deletion for speech storage and voice mail applications. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 36(6):924–927, June 1988.

[16] L. R. Rabiner and M. R. Sambur. An algorithm for determining the endpoints of isolated utterances. Technical Report 2, Bell lab, Feb 1975.

[17] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. In *Proceedings fo the ACM SIGMOD '89 Conference in Groupware systems*, May 1989.

[18] Janet Drake, Vahid Mashayekhi, John Riedl, and Wei-Tek Tsai. A distributed collaborative software inspection tool: Design, prototype, and early trial. *IEEE Software*, 1993. To appear.

[19] E. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[20] David. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *TOCS*, 3(2):77–107, May 1985.

[21] Richard F. Rashid. Threads of a new system. *UNIX Review*, 4(8):37–49, August 1986.

[22] Mashito Hirakawa Tadao Ichikawa. Iconic programming: Where to go? *IEEE Software*, pages 63–68, November 1990.