An Architecture for Distributing Processing on Realtime Data Streams

by
Angela S. Hinrichs

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Science in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY June 1995

© Angela S. Hinrichs, MCMXCV. All rights reserved. The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author.

Department of Electrical Engineering and Computer Science
May 19, 1995

Certified by .

Christopher M. Schmandt
Principal Research Scientist, MIT Media Laboratory
Thesis Supervisor

Certified by .

Daniel C. Swinehart
Principal Scientist, Xerox Palo Alto Research Center
Thesis Supervisor

Accepted by .

Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses
An Architecture for Distributing Processing on Realtime Data Streams

by

Angela S. Hinrichs

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 1995, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Science in Electrical Engineering and Computer Science

Abstract

Emerging local area networks can reliably transfer hundreds of Megabits/sec., permitting the direct connection of audio and video devices to the network instead of the workstation. The network can thus serve as a very flexible data interconnect between the devices and processing intermediaries. By distributing processing on the network, we can extend the range of available processing functions beyond what is available from digital hardware packaged with the devices. A client-server architecture for providing realtime Digital Signal Processing (DSP) functions to audio applications is presented. Audio devices (and files) are the ultimate sources and sinks of data streams; DSP servers receive, process, and send the data to/from devices and other servers. DSP servers must offer generally useful functions and data transport options, and add as little delay and delay variance to the data stream as possible. The server design presented in the thesis allows clients to specify processing tasks by creating objects representing the functions, and creating connections between those objects; it provides a set of audio DSP functions that covers the needs of basic applications; it makes it possible to add new functions to a server without changing the code for any existing functions; and it offers client programmers control over the timing trade-off of introducing additional delay and/or delay variance for the sake of keeping late-arriving data in the stream. An audio DSP server and some simple clients were built and evaluated for the thesis.

Thesis Supervisor: Christopher M. Schmandt
Title: Principal Research Scientist, MIT Media Laboratory

Thesis Supervisor: Daniel C. Swinehart
Title: Principal Scientist, Xerox Palo Alto Research Center
Acknowledgments

Dan Swinehart was my advisor at Xerox PARC, where the work was done. His suggestions and guidance while I was there and his comments while I was writing the thesis back at MIT have been invaluable. A large part of what I learned from the project came directly from our conversations.

Chris Schmandt was my advisor at MIT. His comments on the thesis greatly contributed to the clarity and completeness of the document. His willingness to read drafts in the final weeks, when he had four other Master’s theses to supervise, is much appreciated. I’d also like to thank him for hiring me when I was a freshman who, as he gently pointed out, “really didn’t know how to program.” That job was where I learned how to program and got my first glimpse of computer science in action.

Steve Ward let me be a TA for 6.004, my favorite class at MIT, and gave me office space and the use of a workstation this spring while I was writing the thesis.

My parents, Joel Jr. and Elaine, have supported me financially and emotionally through MIT and the years before. My sister Berkeley and brothers Chris, Tom, and Joe3 were also central figures in my upbringing. My only regret about leaving home for MIT is that I’ve missed much of theirs. It’s meant the world to me that we’ve stayed close across the geographical separation.

The past five years have been years of learning, growing, and plain old enjoying thanks to the incredible friends I met at MIT. Sara and Anne have been my second family since freshman year. Others too numerous to type out before I have to turn in this stack of sheets! have also been defining figures in my MIT experience.

Danielle, in ways that nobody else could, has helped me live and love and grow and she’s gotten me through writing this Master’s thesis.
Contents

1 Introduction ................................................. 9

2 Background ................................................. 12
   2.1 Workstation-based Audio Servers ...................... 12
   2.2 Audio and Video Devices on the Network .............. 15
   2.3 Distributing Audio Digital Signal Processing ............ 16
   2.4 Timing Issues for Realtime Data ....................... 19
   2.5 Example Applications of Realtime DSP ................. 20
      2.5.1 Sound Effects for Shared Virtual Spaces .......... 21
      2.5.2 Echo Cancellation ................................ 21
      2.5.3 Audio Timescale-changing Operations ............... 22
   2.6 Goals of the Thesis .................................. 23

3 Architecture of the Server ................................. 24
   3.1 Implementation Environment ............................ 24
   3.2 Interface Abstraction .................................. 26
   3.3 The ILU Control Interface Implementation ............... 28
   3.4 The Server Core Implementation ........................ 30
      3.4.1 Broker .......................................... 30
      3.4.2 InPort and OutPort Objects ........................ 31
      3.4.3 Mechanism for Passing Data between Function Objects .... 34
   3.5 Implementation of Function Objects ..................... 37
   3.6 "Utility" Function Objects ............................ 40
4 Clients and Supporting Software 42
4.1 Network Sources and Sinks of Audio 42
4.2 A Tcl Extension for Clients 43
4.3 Test Clients 44

5 Results and Observations 47
5.1 Function Objects Implemented 47
5.2 Control over Output Timing 48
5.3 Constructing Clients 51
5.4 New Function Objects from Old Code 52
5.5 Future Work 52

6 Conclusion 55

A ILU Interface Descriptions for Server Objects 57
A.1 DSPCore.isl 57
A.2 RTP.isl 59
A.3 Fmt.isl 60
A.4 Rate.isl 62
A.5 Mixer.isl 63
A.6 TC.isl 64
A.7 Big.isl 65
List of Figures

2-1 Workstation-based Audio and Video .................. 13
2-2 Networked Audio and Video ............................ 15
2-3 Server Control and Data Paths ......................... 18
2-4 Server(s) Adding Room Effects in a Shared Virtual Space .... 21

3-1 Server Interface Model ................................. 27
3-2 ILU Interface Description for Rate Conversion Object .......... 29
3-3 PQ (packet queue) and Packet data structures ................. 36
3-4 Example processing module main loop routine ............... 39

4-1 Example test client setup (time compression) .............. 44
4-2 Excerpts from example test client code .................. 46
List of Tables

3.1 Hook routines for InPort and OutPort state-changing calls . . . . . 32

4.1 Device control (devctl) packet formats . . . . . . . . . . . . . . . . 43
Chapter 1

Introduction

The speed and capacity of computers and networks have increased dramatically in recent years, making possible more and better applications of digital audio and video. Digital Signal Processing (DSP) is the body of transformations of signals performed by manipulating digital representations of the signals. Examples of DSP operations include compression/decompression, encryption, rate conversion, filtering, and speech/image recognition. Sometimes it is desirable or necessary to perform the processing in real time, i.e. to process the data as fast as it is being produced and consumed. For example, mixing of audio or video streams for live teleconferencing must be performed in real time if it is to be usable.

Demands on bandwidth of communication channels and speed of processing are high for realtime DSP. This is especially true for video, where data rates are generally an order of magnitude higher than for audio even with compression. Special hardware has typically been used to perform DSP operations where speed has been an issue. Furthermore, since the bandwidth required to move data around in real time is high, the data have typically been confined to special devices or to workstation buses. However, processing power per dollar has increased exponentially with time, and emerging network technologies are capable of carrying multiple streams of realtime audio and video data, so it is becoming feasible to send these media over the network and process them on a different machine from the machine where the data is digitized. The flexibility of systems that involve distributed processing
on realtime audio and video data can be increased by offering processing functions in servers whose interfaces are modular, open, and easy-to-use, provided that the servers add very little delay. These servers could run on high-powered, expensive machines, providing a shared resource.

Digital audio and video devices\(^1\) are typically designed to send or receive data to or from a workstation’s I/O bus. The workstation and its audio and video devices are a mostly closed system, even when the workstation is on a network, because current networks have trouble supporting multiple streams of realtime high-quality audio and video data. Processing of data from these devices in real time is almost always performed on the workstation to which the audio or video device is attached, or on a separate DSP processor that interfaces to the workstation’s I/O bus, especially in the case of video.

Fortunately, affordable Local Area Networks (LANs) with aggregate bandwidths in the hundreds of megabits per second range will be available within the next few years, lifting the network capacity restriction. Considering past trends, it is reasonable to expect that people will find uses for the bandwidth, and that realtime audio and video data will soon be common traffic on the LAN. When that is the case, it will no longer be necessary to tether devices to workstations nor to process the data only on the device’s workstation.

If devices connect directly to the LAN and processing of data is necessary after the data is sent over the network by the devices, then processing must be distributed. Applications can be structured so that the processing takes place always on the workstation which is running the application program requiring the processing. On the other hand, the processing could be performed by a server process on some machine that might or might not be the workstation running the application program. The delays involved in both cases should be the same if the application workstation or server is on the same LAN as the devices: LAN delays for sending data between the devices and processing machine and the delay associated with the processing itself. The second way of structuring applications, as clients of a processing server,

\(^1\)microphones, speakers and cameras with digitizers, CD players, etc.
has several advantages over the first way if the processing servers can provide all of the processing functions needed by applications. Structuring applications as clients can simplify the task of writing applications and porting them to different machines and operating systems. It can also make it possible to share the processing power of on expensive machine among a group of users.

This thesis explores issues of distributing processing on realtime data streams, i.e. reducing the complexity of the application programs by performing processing operations on server machines and structuring applications of realtime signal processing such that they use the server(s) for data processing. A server for audio DSP functions was built to demonstrate and develop the motivating ideas. The work was done at the Xerox Palo Alto Research Center (PARC) in the Computer Science Laboratory. The interface to clients is fairly simple and straightforward to use for tasks that involve continuous processing on steady streams of audio data. Clients can set various parameters to control buffering and timing of the processing tasks that the server creates for them. The server is extensible instead of limited to a predetermined set of processing capabilities.

The next chapter will discuss background material and related work. Chapter 3 describes the server implemented for the thesis. Simple clients and other code written for the thesis are described in Chapter 4. Observations and suggestions for future work are discussed in Chapter 5, and Chapter 6 concludes.
Chapter 2

Background

This chapter will provide background for the thesis and discuss related work. Section 2.1 describes workstation audio device servers. Section 2.2 motivates designing audio and video devices to connect to the local area network instead of the workstation. Two approaches to structuring applications of realtime multimedia processing are contrasted in Section 2.3 and timing issues for realtime data streams are described in Section 2.4. Section 2.5 describes several applications of realtime audio DSP and Section 2.6 briefly summarizes the goals of the thesis.

2.1 Workstation-based Audio Servers

Applications that involve manipulating audio data between audio devices generally need to control certain audio device parameters and actions: volume, possibly sampling rate, starting and stopping, and source or destination of the data. Many "audio servers" have been built to offer a clean interface to audio devices attached to workstations[14, 8]. This section will provide some historical background on audio servers that have focused on controlling audio devices attached to a workstation. These audio servers’ models of functions and data connections have served as a basis for our networked processing server’s model of processing functions and connections between them.

As long as digital audio devices have been attached to workstations, people have
been writing simple programming interfaces to audio devices to hide the idiosyncrasies of particular device drivers and to allow those devices to be shared. These were written as libraries whose routines controlled the local audio device, and as audio servers that offered a networked client-server interface to the device. In addition to a simple interface, some offer models for building compound audio tasks to run within the server. Three audio servers are described here: Olivetti’s VOX[4], DEC’s XMedia[3], and NCD’s NetAudio[5].

VOX models audio devices as LAUDs (Logical Audio Devices). Each LAUD can have input and/or output ports, and ports can be “soldered” together to establish data connections between the devices represented by the ports’ LAUDs. A client can register collections of LAUDs whose ports are soldered together as a CLAUD (Composite LAUD), which can have its own data ports. After creating and registering a CLAUD, a client can ask for instances of that CLAUD, so it doesn’t have to create and connect the component LAUDs all over again. VOX was written for devices whose data inputs and outputs were generally analog, and used a digitally controlled crossbar patchboard to establish the data connections, instead of passing digital data inside the host computer. Nevertheless, the model of LAUDs with input and output data ports and CLAUDs as new LAUDs is an elegant way to describe an interconnection of components that produce and/or consume audio data.

XMedia’s model of audio devices and connections is very similar to VOX’s; it separates “virtual devices” (device-independent device interfaces) from LOUDs (still
Logical Audio Devices), which are seen as roots of hierarchical trees with virtual devices at the leaves, instead of as containers like CLAUDs. As in VOX, data connections are established between ports of virtual devices; connections are called “wires” and have data format types associated with them. Unlike VOX, XMedia was written for workstations with digital audio devices, so data connections were formed inside the workstation, and it could send or receive data to or from clients as well as devices.

NetAudio, an audio server from Network Computing Devices, also focuses on device control, but several useful processing functions are built into the server. Clients specify flows, which are tree-structured descriptions of connections between devices and intermediate processing elements. Mixing and gain control are the available processing elements. NetAudio automatically converts data formats and rates as necessary. It also has storage elements called buckets, which hold data that will be frequently accessed, and wave function generators. The authors plan to add programmable filters and radio elements for broadcasting device data on the network. Each element has a state, one of Started, Paused, or Stopped, to indicate that it is producing/consuming data, waiting for room/data to become available, or inactive, respectively. Clients can control the amount of buffering used by elements and can request notices that data or buffering is about to run out.

The NetAudio server approaches our goals for an audio processing server. Its interface to clients is simple and flexible; it allows clients to define the actions of elements in a flow when too much data or no data is available from a source; and it offers several common, useful processing operations. However, its list of processing operations is predetermined, and substantial parts of the server would have to be modified in order to add a new processing element. Also, once a flow is defined, it cannot be modified; instead it must be destroyed and redefined.
2.2 Audio and Video Devices on the Network

Most current LANs do not have enough bandwidth to support many streams of high-quality, uncompressed realtime audio data, much less video. Audio and video devices plug directly into workstations because the workstation buses can handle the high data rates. In order to share realtime video data over the network, the data must be compressed and decompressed at the workstations on either end. Since audio data requires less bandwidth, compression is not absolutely necessary for sending one or two streams of high-quality audio, but as few as 5 to 10 simultaneous streams could overload current LANs such as Ethernet, which can carry up to 10Mb/s.

High-bandwidth network technologies such as ATM (Asynchronous Transfer Mode, 155Mb/s per link) promise to lift the constraints on audio and video data traffic in the local area. When the bandwidth of the typical LAN increases to the hundreds of Megabits per second range, audio and video devices can be made to send and receive data directly to and from the LAN instead of workstation buses. As an example, MIT's Telemedia, Networks and Systems group (TNS) has devel-
oped a video capture board that connects directly to an ATM LAN; they plan to
design and build several other network-based multimedia peripherals[2, 1]. Similar
approaches have been or are being developed at Xerox PARC, Cambridge University,
and elsewhere.

This network peripheral approach will make it easier to move devices from one
place to another, to change a user’s devices, and to share the data from a device
among multiple users or programs. The devices will have enough internal processing
power to run the LAN interface, their control interface, and some basic signal
processing. However, the devices’ onboard processors will probably not have the
programming flexibility (operating system, file system, etc.) of a networked computer,
so additional processing might need to be performed by some machine that
gets the data from the LAN.

2.3 Distributing Audio Digital Signal Processing

Two approaches to structuring applications that process audio and video data are
described in this section. Both use the network to transport data between devices
and processors. The MIT TNS architecture structures applications as single pro-
cesses that receive, process and send or display data on the end user’s workstation.
The approach taken in this thesis is to run the high-level code and graphical user
interface of the application on the workstation, while processing is performed by a
server process which may be on the same machine or a different machine.

TNS has developed a gigabit per second desk/local area ATM network called
VuNet[1], a video capture device that connects directly to the VuNet[2], and VuSys-

tem, a programming system for applications that process live video[9, 10].

VuNet is a software-intensive system: complex network functions are handled by
software drivers so that the network hardware can be very simple and the software
implementation can be changed or ported to different systems without affecting the
hardware. The software-intensive approach also involves processing data in large
timeslices with large gaps in processing, on the order of 100 milliseconds. Since the
data is processed in such a bursty fashion, the ultimate sources and sinks of data must be able to send and receive data in bursts instead of more regularly timed streams.

The goal of VuSystem is to support applications that process audio and video data instead of simply directing it from one device to another. It uses VuNet’s high bandwidth to deliver video and audio data directly to the workstation on which the application runs. Processing tasks are structured as directed graphs of processing modules. Application code is split into two bodies, \textit{in-band} code and \textit{out-of-band} code. In-band code is the data processing code that runs on every video frame or audio segment. Out-of-band code is the higher-level, event-driven control code and the graphical user interface of the application. In-band code is written in C++, and out-of-band code is written in Tcl[12], extended with additional commands to control processing modules. In-band modules are linked in with the extended Tcl shell that interprets applications’ out-of-band code. Data is passed between processing modules in dynamically-typed \textit{payloads} that are also used as scheduling tokens to determine which piece of application code should be executed at any given time.

The approach of this thesis is to structure application programs such that the user interface and high-level control code run on the user’s workstation while DSP operations are performed by a networked server. Instead of requiring all users’ workstations to have the I/O bandwidth and processing power required for realtime DSP, a high-performance DSP processor could be used as a DSP server. Users could share that resource via the LAN instead of doing all processing locally. Allowing a group of users to share a DSP server would eliminate the necessity of buying DSP hardware for each of them. It would also decrease the cost of upgrading their workstation hardware: DSP code written for the server would be unaffected by their change of hardware. Alternately, every machine could run a DSP server, some with more functions available than others, so that a few complicated tasks could be spread out over a pool of machines.

Current workstations can perform many useful audio DSP functions, so one
might wonder whether pushing DSP from workstations to a server would be worth the trouble, especially since processors are getting faster and cheaper so quickly. However, there are some audio DSP functions that require more processing power than workstations are likely to have in the near future, such as echo cancellation and sound localization. Also, many processors technically capable of performing realtime DSP run operating systems that unpredictably deschedule realtime processes for as long as 100 milliseconds and don’t deliver reliable fast timers and system clocks. A delay of 100 milliseconds is too large for some realtime applications. Moreover, if multiple stages each add 100 milliseconds of delay or jitter to a signal, then the delays and delay variances accumulate across the stages and the final destination must buffer the data stream enough to ensure that the data can be still played continuously. If the jitter exceeds the added delay/buffering, then a gap in the playback of data will result. Gaps of even 10 milliseconds are audible, therefore undesirable. Special care could be taken with a DSP server to ensure realtime performance from the hardware, operating system and network interface. An audio DSP server could also serve as a model for a video DSP server, which would more likely require expensive special-purpose processing hardware.

The audio DSP server presented in this thesis makes DSP services available on the network similar to the way traditional audio servers make device control available on the network. In an environment with audio DSP servers and audio devices whose data connections are to the LAN, applications running on user’s workstations do not need to directly manipulate audio data. Instead, they can remotely control the
devices and the DSP server. If they don’t want to use the DSP server, they can
direct the devices to send/receive the data to the local host for processing.

2.4 Timing Issues for Realtime Data

A realtime data stream is simply a stream of data packets arriving at a rate that
matches the rate at which they would be produced or consumed by physical devices.
For example, a realtime data stream of audio sampled at 8kHz with one byte per
sample is composed of a sequence of packets whose data portions sum to 8kBytes
every second. It is desirable for the timing of these packets to be very regular.
Continuing the example, we might choose a packet data size of 400 bytes, so that
our stream should include 20 such packets each second. If one packet arrives each
50 milliseconds, then we can send each packet to a device such as a speaker when it
arrives, and be done with it. However, if a packet is late, then the speaker will not
have any audio to play for that time, and will produce a pop or other glitch sound.
If too many packets arrive within a single packet time, there may not be room to
store them all, so it may be necessary to lose some of the data, thus creating a glitch
in the output later on.

Delay is the amount of time from the production of data until its arrival (at the
device that will play it or process it). Packaging delay is incurred when multiple
samples are gathered into a single packet; the first sample created could be sent off
immediately, but instead it must wait for all other samples to be gathered into the
packet before the packet can be sent off. Other sources of delay are network transport
and buffering. Jitter, or delay variance, is the average absolute difference between
actual packet arrival times and perfectly regular packet arrival times, independent
of a constant delay shared by all packets in the stream.

A system component expecting a realtime input and attempting to produce a
realtime output must deal with delay variance in the timing of the input packets. A
common method used to decrease the delay variance of the outputs is to buffer up
some input packets before starting to process, and then to use a timer to drive the
production of outputs, using the buffered-up initial packets as insurance that there will be data to process, even if a subsequent input packet is more than one packet time late. Providing that the delay variance of the inputs never exceeds the initial buffering delay, the delay variance of the outputs will be that of the timer used to drive processing.

Another approach to minimizing delay and delay variance is to attempt to process data as soon as it is available. Assuming that processing takes an equal amount of time for each packet, the delay variance of the output will be the delay variance of the input. Delay variance introduced in the transport stages will add up, but the ultimate destination component can apply whatever buffering is necessary to cope with the delay variance, or it can simply lose data every now and then.

When two or more streams are used as input to a processing stage, the differential delay and delay variance between the streams must also be considered. If packets from one stream arrive consistently later than packets from another stream, then packets from the earlier stream can be buffered so that packets originating at the same times can be processed together. Alternatively, packets that arrive at about the same time could be processed together, despite the delay of one stream relative to the other. If the differences between arrival times of packets from different streams are not regular, then there is differential delay variance. To avoid losing data, there must be sufficient buffering to hold multiple packets arriving from one stream while the processing stage is still waiting for a packet to arrive at another. However, often loss of data is more desirable than a long wait for data from one stream. Different applications treat this tradeoff differently, so it is important for a processing server to allow clients to set its threshold for delay and buffering vs. loss of data.

2.5 Example Applications of Realtime DSP

Several applications of realtime audio DSP are described in this section to illustrate possible uses of our server and realtime application requirements.
2.5.1 Sound Effects for Shared Virtual Spaces

An application that provides virtual rooms for users to meet and converse could have the same "room effects" added to the audio data from each user's microphone.¹ For example, in a large virtual room some reverb could be added to the voice, and perhaps some attenuated noise from nearby rooms mixed in, etc. Each user's speaker would then get a mix of the other users' processed streams. The mixing could be done by the application program on the user's workstation or on some DSP server. In Figure 2-4, each reverb and mix bubble is a task running on some DSP server (network transport server functions providing data to and sending data from the reverb and mix objects are not shown).

2.5.2 Echo Cancellation

Consider a teleconferencing setup in which each participant has a speaker and a microphone. The speaker plays the sounds from other participants' microphones mixed into a single sound. When one participant speaks, that sound will be played at other participants' speakers a short time later, and picked up by those participants' microphones. A short time after that, the sound from those microphones will travel back to the first participant's speaker, causing an echo. There are feedback loops between all participants' microphones and speakers. This echo can be annoying at the least, and the feedback loops can cause ramping loop-frequency tones to drown

¹David Nichols of Xerox PARC CSL provided this example.
out the signal if the gain of the loops is not extremely small.

Therefore echo cancellation is very desirable for systems with a open-air microphones and speakers. It is extremely computation-intensive, however, and very sensitive to the timing of its inputs and the transfer function of the feedback paths. Any irregularity in input timing, for example if too many or too few samples from a given time period are used as as inputs, can cause the echo cancellation filter to amplify echoes. For echo cancellation to work with distributed devices and processing, the timing of the digitization and transport of the data must be very regular.

### 2.5.3 Audio Timescale-changing Operations

To timescale a signal is to linearly stretch or shrink the signal along the time axis. In other words, given a two second long audio segment, we can stretch the signal out by a factor of 1.5 so that it now plays for 3 seconds, or shrink it by a factor of 2 so that it plays for 1 second. The pitch of the sound changes linearly as well; shrinking by a factor of two along the time axis means that the pitch rises one octave.

Another way to change the timescale of a signal is to remove or duplicate redundant segments of a waveform. This method does not change the pitch of the sound. It is called “time compression” in the rest of the paper to differentiate it from the linear timescaling described in the previous paragraph. (Note that time decompression is also possible with this method.)

“Time clutching” is the use of time compression for catching up with a realtime audio stream from a late start. Imagine that an ongoing, recorded conversation started five minutes ago and a user wants to hear the conversation from the beginning but eventually catch up to the speakers. Time compression can be used to speed up the recorded sound of the conversation (without raising the pitch) until the user has heard all of the old buffered sound and can listen to the rest of the conversation in real time.

An interesting feature of any kind of processing that involves a change of timescale is that the input can be realtime or the output can be realtime, but never both. This is discussed further in Section 5.2.
2.6 Goals of the Thesis

Distributed capture, processing and playback of audio data allows great flexibility in the physical setup of audio and processing hardware. Structuring applications so that processing of realtime data is performed by networked servers instead of by the applications themselves allows flexibility of implementation of the processing and sharing of processing resources.

The main goal of the thesis is to provide a framework for processing realtime data streams as a network service in support of such a style of distributed application. The conceptual model of data streams and processing tasks must adequately describe them for the class of applications described in Section 2.5. The server's interface to clients must be straightforward to use and flexible enough to accommodate the control interfaces required by different processing modules. The server implementation must be extensible, so that new processing modules can be added to the server without changing the interface or code of any other modules. The server must add as little additional delay and delay variance to the data streams as possible.
Chapter 3

Architecture of the Server

This chapter will describe the design and implementation of the server built to test the designs developed in this thesis. Section 3.1 describes the programming environment in which the server code was written. Section 3.2 describes the main components of the client-server interface. Section 3.3 discusses the implementation of the client-server interface in more detail. The implementation of the core components of the server is described in Section 3.4. Section 3.5 gives the guidelines for implementing data processing function objects for the server framework. Section 3.6 describes a few processing function objects that are useful for constructing complex tasks and moving data around different platforms.

3.1 Implementation Environment

The server was written in ANSI C for IRIX 5.2 and ran on an SGI Indy. C was chosen to implement the server because it was immediately available, familiar to the author, efficient, and a large body of already-written signal processing software is available in C. IRIX was chosen initially because of several available operating systems (Sun Solaris, SunOS 4.1, and IRIX), it had the best realtime support: a high-priority scheduling class and reliable fast timers and interrupts.

Inter-Language Unification (ILU)[7], a multi-language object interface system, was used for the client-server interface and general structure of the server. ILU
interfaces are written in ILU’s ISL (Interface Specification Language) and can be compiled into client and server implementation stub procedures in several common programming languages, including ANSI C, C++, Common Lisp, and Modula 3, for several machine and operating system platforms. Interfaces can be defined in terms of objects with methods that can raise exceptions as well as return values, while each client or server can be implemented in any language and operating system supported by ILU. Communication between client and server is implemented by shared memory if they run on the same machine, and by one of several remote procedure call protocols if they run on different machines.

ILU clients invoke object methods by calling client stub procedures on object handle data structures. A client can obtain an object handle either as the return value of some method or by calling an ILU-generated procedure that creates an object’s handle from a universal object identifier string called the String Binding Handle.

An ILU interface can include the definitions of many data types and objects. Each interface has a name which will be referred to as Ifc in subsequent sections. Each object has a name within the interface, which will be referred to as Obj. In interface declarations other than the one in which it is defined, that object is called Ifc. Obj. This naming convention will be used widely in the next sections.

Real-Time Protocol (RTP)[15] provides end-to-end network transport functions specialized for realtime data such as audio and video data. It transfers data in a single direction, allowing for multiple destinations (multicast) if the underlying network transport protocols provide for multicasting. It is designed primarily but not exclusively for conferencing applications. RTP is accompanied by a control protocol (RTCP) for monitoring quality of service and communicating session information between multiple participants. RTP headers include fields to identify the type, source, and timing of data in the packet’s payload. The main source identifier, called the synchronization source identifier (SSRC), is a randomly generated number. Timing information is provided by the sequence field, which is incremented by 1 with each new packet, and the timestamp field, which is incremented by the
number of samples contained in each packet. A header extension bit can be set to indicate that additional fields follow the common RTP header fields.

RTP is in the process of becoming an Internet Engineering Task Force (IETF) standard for transmitting realtime data. The sample timestamp identifier field in RTP headers marks the data with single-sample resolution timing information. RTP’s support for header extensions allowed us to include additional timing information in the data streams. Therefore, we chose RTP as the means of transporting data between LAN-connected sources and sinks of audio and the server. RTP data packets are sent as UDP packets with RTP headers, extended to contain information about the type and timing of the data: a timestamp in the NTP[11] format, the sampling rate of the data in Hz, a data format identifier, the number of bits per sample, and the number of channels in the stream. The packets are sent over a standard 10Mb/s Ethernet LAN.

3.2 Interface Abstraction

This section will explain the conceptual model of processing tasks performed by the server for clients, and the client-server interface abstraction that supports it.

The server itself, processing functions running as part of the server, and data ports associated with the functions can be viewed as objects on which the client can make control calls. The client arranges for processing of streams of data in the network by asking the server to create objects representing the network transport and DSP functions it needs, and then connecting those objects’ data inputs and outputs together.

The server is represented by an object called the Broker (because it deals out processing function objects to clients). The Broker can return its list of function objects it supports, return its list of currently running functions, or take requests to create new function objects. All other interactions with functions running in the server are carried out through the function object interfaces, not the Broker.

A function object should have at least one data input port (InPort) or at least
one data output port (**OutPort**) through which it can receive or send data from or to other function objects in the same server. It must have a self-destruct call and calls that return handles to its data ports. It may provide any number of additional calls to offer the client control over its operation, such as calls to get and set parameters and calls to tell it to initialize or change state. For example, one might create an object for rate conversion and call it **RateConv**. It would have one data input and one data output for connecting to other function objects within the same server, so it would need calls to return handles to those ports: **GetInput** and **GetOutput**. We could allow clients to set input and output data sampling rates by defining calls **SetInRate** and **SetOutRate**. The processing code would use the input and output rates to figure out a conversion factor and would process data whenever it was available from its **InPort**.

**InPort** and **OutPort** objects are separate objects from their parent function objects so that connections between data ports can be standardized. Writers of function objects need to know only about port objects, not about other function
objects. In other words, connections are made by calls to port objects, instead of function objects. Both InPort and OutPort objects have calls to set data type information. InPort objects provide calls to set various buffering and timing parameters which will be discussed later. OutPort objects provide calls for making and breaking connections with InPort objects.

3.3 The ILU Control Interface Implementation

Function object interfaces are defined in the Inter-Language Unification (ILU) interface specification language, version 1.6[7]. ILU interfaces can be compiled to client and server stubs in one of several programming languages, which include ANSI C, C++, Modula 3, CommonLisp, and Fortran. ILU allows us to write object-oriented client-server interfaces, declare exceptions that can be returned from calls, and implement them in our language of choice. The server was written in C, but clients can be written in any language and operating system supported by ILU.

ILU interfaces are written in ILU’s Interface Specification Language (ISL). An interface file contains the definition of one interface, which can include many data type definitions (record, sequence, short integer, etc.), and many objects.

The client gets a Broker object by calling an ILU routine with an identifier called a String Binding Handle exported by the server, which the client can get from a file, user input, or some other name service.

The Broker, InPort, OutPort, and FunctionT objects are defined in the ILU interface DSPCore. Function objects are defined in separate ILU interface descriptions.

All function objects in our server must be subclasses of FunctionT, which has a single method Die. This allows code to be written for all function objects (even those that haven’t yet been defined), while still allowing each type of object to be specialized for its function. Figure 3-2 shows the ILU interface description of the RateConv object described in the previous section. The ILU interface name is Rate and the object name is Conv. The Rate interface imports the DSPCore interface
INTERFACE Rate IMPORTS DSPCore END;

EXCEPTION BadValue: INTEGER;

TYPE Conv = CLASS
  SUPERTYPES DSPCore.FunctionT END
  METHODS
    getInput () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort,
    SetInputRate (r: INTEGER)
      RAISES BadValue END,
    SetOutputRate (r: INTEGER)
      RAISES BadValue END
  END;

Figure 3-2: ILU Interface Description for Rate Conversion Object

and the Conv object declares DSPCore.FunctionT as a superclass. ILU 1.6 allows multiple inheritance, so function objects can be subclasses of FunctionT and other ILU object classes simultaneously.

ILU objects can have only methods in their interface definitions, not fields, so routines must be defined to get and set any data that might otherwise be a field, including a function object’s InPort and OutPort objects and algorithmic parameters. However, objects can be implemented with fields. The C implementation of each object in the server includes the definition of a structure IfcObjSpecificData, where IfcObj is the ILU interface and object names concatenated. Whenever a new instance of an object is created, that instance’s ILU structure’s data field is initialized to point to such a structure.

To access the fields of the structure, the data field of the ILU structure must be typecast to the appropriate IfcObjSpecificData. It is all too easy to incorrectly typecast the data field; for example, we could accidentally typecast the data field of an InPort object as an OutPortSpecificData pointer, and then use it as an OutPortSpecificData without getting any compiler errors.
3.4 The Server Core Implementation

The server core consists of the server main routine, the DSPCore objects (Broker, FunctionT, InPort, and OutPort) the bookkeeping for active function objects, the mechanism for passing data from one function object to another, and the programming interface to the function objects.

The main routine is very small; it initializes some operating system parameters, exports its String Binding Handle (the identifier by which ILU clients can reach it), and finally it calls ILU.C.Run so that the main thread can start handling calls from clients. Function objects’ processing threads are spawned by this thread when clients ask the Broker to create new function objects. All calls from clients to any object in the server will be handled by the main thread.

3.4.1 Broker

The Broker object has three methods: listAvailable, listActive, and createFunction. The first two are very simple. listAvailable returns an ILU sequence of strings, the list of names of function objects offered by the server. These names are in the ILU interface declaration format, Ifc. Obj. That sequence never changes during the execution of the server program, because function object modules are included in the server by compilation as opposed to dynamic linking (which would admittedly be nicer). listActive returns an ILU sequence of strings, the list of names of currently running function objects. These names are also in the ILU interface declaration format and are intended to give clients a rough indication of the server’s load, in terms of how many and what function objects are currently running. Although easy to implement in the server, this is a crude way of communicating load information. Numbers indicating the percentage of idle CPU cycles and memory usage would be more digestable by clients. This list is updated by createFunction and a routine that must be called by objects’ Die methods.

createFunction has a single argument, a string naming the requested function in the ILU interface description format, Ifc. Obj. If successful, it returns an ILU
object handle for the newly created function object. Otherwise, it raises an exception NotSupported.

createFunction searches through a linked list of structures containing function object initialization information, trying to find a match for its argument. When it finds a match, it starts up the requested function object task as follows:

- allocates space for the ILU instance identifier and IfcObjSpecificData structure.
- calls the function object’s creation call, which is generated by ILU from the object’s interface description.
- calls the function object module’s routine for initializing the IfcObjSpecificData structure.
- adds the ILU function object structure to its internal list of currently active function objects.
- spawns a thread to run the function object module’s data processing routine and requests a high scheduling priority for the thread.
- updates the external list of active function objects (for listActive).

3.4.2 InPort and OutPort Objects

InPort and OutPort objects are created in their parent function objects’ IfcObj SpecificData initialization routines. Instead of the ILU calls to create InPort and OutPort objects, the routines DSPCore.InPort.New and DSPCore.OutPort.New must be used so that the port object’s data field can be properly initialized.

Data ports are separate entities from their parent function objects so that data connections between function objects can be implemented outside of the function objects’ code. However, function objects generally need to know when the client calls an InPort or OutPort method that would change the InPort or OutPort object’s state, so the implementation of such methods (Connect etc.) includes a call to a procedure defined as part of the function object implementation, if the procedure exists. To let data ports access pointers to those procedures, which are fields of the port’s parents’ data structures, each InPortSpecificData and OutPortSpecificData contains a pointer to the parent function object. Only the
<table>
<thead>
<tr>
<th>Hook routine</th>
<th>Arguments</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>InPortSetPolicyHook</td>
<td>DSPCore_FunctionT, DSPCore_InPort, DSPCore_PolicyInfo *</td>
<td>void</td>
</tr>
<tr>
<td>InPortCheckFormats</td>
<td>DSPCore_FunctionT, DSPCore_InPort, DSPCore_OutPort</td>
<td>int</td>
</tr>
<tr>
<td>InPortConnectHook</td>
<td>DSPCore_FunctionT, DSPCore_InPort, DSPCore_OutPort</td>
<td>void</td>
</tr>
<tr>
<td>InPortDisconnectHook</td>
<td>DSPCore_FunctionT, DSPCore_InPort, DSPCore_OutPort</td>
<td>void</td>
</tr>
<tr>
<td>OutPortSetPolicyHook</td>
<td>DSPCore_FunctionT, DSPCore_OutPort, DSPCore_PolicyInfo *</td>
<td>void</td>
</tr>
<tr>
<td>OutPortCheckFormats</td>
<td>DSPCore_FunctionT, DSPCore_OutPort, DSPCore_InPort</td>
<td>int</td>
</tr>
<tr>
<td>OutPortConnectHook</td>
<td>DSPCore_FunctionT, DSPCore_OutPort, DSPCore_InPort</td>
<td>void</td>
</tr>
<tr>
<td>OutPortDisconnectHook</td>
<td>DSPCore_FunctionT, DSPCore_OutPort, DSPCore_InPort</td>
<td>void</td>
</tr>
</tbody>
</table>

Table 3.1: Hook routines for InPort and OutPort state-changing calls

The first 9 fields of the parent’s IfcObjSpecificData can be accessed, since those are the fields common to all function object IfcObjSpecificData structures. They are pointers to “hook” routines, which are called by InPort and OutPort calls when the parent function object needs to be involved. Hook routines are called with the ILU object handles of the function object to which they belong and the arguments of the InPort or OutPort call that required the hook. See Table 3.1 for a listing of the hook routines and their arguments and the rest of the section for a description of the calls that use the hooks.

InPort owns the buffering and timing policy information about the connection associated with it. The calls that make and break connections belong to OutPort.
Both have a data format structure in their `SpecificData` structures. When the client requests a connection, hook routines of the `InPort`'s parent function object and of the `OutPort`'s parent function object are called to check that both parents agree to the connection. These hooks are primarily for the parent objects to check the data formats of the port objects.

`InPort` has three methods: `GetSrc`, `GetPolicy`, and `SetPolicy`. `GetSrc` returns the `OutPort` to which the `InPort` is connected, or raises the exception `NotConnected`. `GetPolicy` and `SetPolicy` get and set a four-field record containing buffering and timing parameters; these parameters are described in Section 3.4.3. After `SetPolicy` updates the parameters, the `InPort`'s parent's `InPortSetPolicyHook` routine is called (if it is not a null pointer). If the `InPort` is connected to an `OutPort`, the `OutPort`'s parent's `OutPortSetPolicyHook` is called.

`OutPort` also has three methods: `GetDest`, `Connect`, and `Disconnect`. `GetDest` returns the `InPort` to which the `OutPort` is connected, or raises the exception `NotConnected`. `Connect` is called with the ILU object structure for the `InPort` as an argument. First `Connect` calls both ports' parents' `InPortCheckFormats` and `OutPortCheckFormats` hooks, and raises `Incompatible` if either returns 0. Otherwise it goes ahead with the connection,disconnecting the `OutPort` if it is already connected. The connection is made by setting the `OutPort` object's `OutPortSpecificData.dest` field to the `InPort` object, and setting the `InPort` object's `InPortSpecificData.src` field to the `OutPort` object. `Connect` then calls both parents' `ConnectHook` calls so that the parent objects can update their data after the change in their ports' data. `Disconnect` raises `NotConnected` if the `OutPort` is not connected. Otherwise, it calls both parents' `DisconnectHook` calls and then clears the `InPortSpecificData.src` field and the `OutPortSpecificData.dest` field.
3.4.3 Mechanism for Passing Data between Function Objects

The mechanism for passing data from one function object to another is hidden from clients; clients know only that connections are made by requesting an OutPort and InPort to connect, and that the behavior of the connection is controlled by several "policy" parameters that clients can set through a call to the InPort. This subsection will describe the policy parameters, the original audio data-passing mechanism, and the one used in the final implementation.

InPort objects have Get_ and Set_Policy calls that get and set a four-field record with parameters that control the behavior of the connection between the InPort and whatever OutPort it is connected to. The fields are

- **maxBuf**, the number of sample frames\(^1\) allowed to build up in the connection before the producer overwrites data or blocks.

- **overwrite**, a Boolean value which matters only when more than maxBuf samples build up between the producer and consumer: if true, the data is discarded; otherwise, the producer blocks until the consumer takes some data.

- **constDelay**, the number of sample frames to allow to build up initially as an insurance against jitter.

- **maxSlip**, the biggest differential delay allowed between two different input streams. If the delay is greater, than data from the slower stream will be lost.

In hindsight, maxSlip does not belong to a connection. Unlike the others, it is not implemented by the data passing mechanism. Instead it is left to function objects' processing code to pay attention to the maxSlip parameter of each InPort, since it is that code that decides whether to processes data when some input has become available.

Originally, ring buffers were used to pass data inside the server, and the above parameters had not yet been defined. Function object processing threads were awakened by timers periodically, expecting to find data in their ring buffers. The

\(^1\)A sample frame is a block of \(n\) samples, one from each channel of an \(n\)-channel audio data stream. It was chosen as the unit of these parameters because it is more precise than a floating-point time value and is independent of the number of channels of a stream.
first and last sample frame in the buffer were visible to the function objects, but they had no way of knowing if all data between the first and last samples was present. We also considered using a single thread to run the processing code for all function object tasks; the tasks would be topologically sorted so that the task on the producing side of a connection would be run before the task on the consuming side. Using threads made for a simpler and cleaner implementation.

Using ring buffers allowed function objects to process input data in variable-sized chunks. We also hoped that using a timer to drive the production of output data would help us achieve very low jitter in our outputs. However, there were several problems with this. First, operating system timers are often unreliable: instead of every 50ms, they may come 10ms apart, then 100ms. It is inefficient to give a separate timer to each function thread, and on some operating systems it is not possible. Second, since the function object knows only the time of the first and last sample contained in the buffer, if packets arrive out of order, it may think that it has the data from both packets before it actually does, and it may use garbage data as input when it should actually wait until the next timer tick. Finally, the decision of what data is present and whether or not to produce output falls on the writer of each function module. The complexity of using ring buffers to pass packets from one function object to another made it difficult to write function object processing routines.

To avoid these problems, we changed the mechanism to packet queues, relying the timing of inputs to drive processing rather than timer interrupts. This means that jitter in the inputs can show up directly in the outputs and varying processing delays can add to that jitter. However, there is no way to cut down on jitter without introducing some delay as insurance. Even with extra delay to mask jitter, if the jitter exceeds that added delay then we still must make a choice between allowing jitter in the output and losing data. Therefore, using the arrival of inputs to time the processing of data minimizes the delay added by the server, relying on the ultimate sink of data to buffer data enough to insure against jitter in the playing of the data. It also allows the use of more dependable operating system scheduling primitives.
such as waiting for data to arrive from the network or block/unblock operations, as opposed to fast, precise timers.

A packet queue is simply a record containing the policy fields described above and a pointer to a linked list of data packets. See Figure 3-3. Each packet is a linked list record containing a pointer to the actual data, the number of sample frames of data, and a sample frame sequence number.

A function object waiting for input data calls \texttt{PQ\_getPacket}, which blocks until a packet arrives on an input queue. When it processes the input and produces output data, it bundles that data into a packet (or reuses the input packet) and calls \texttt{PQ\_addPacket} on the packet and its \texttt{OutPort(s)}. If too much data has built up in the connection, depending on the value of the \texttt{overwrite} parameter of the \texttt{InPort} connected to its \texttt{OutPort}, \texttt{PQ\_addPacket} either flushes the connection and adds the packet or blocks until there is enough space available in the connection and adds the packet.

The packet queue mechanism was a better fit to the way data was actually moving around in the server, since data arrived from the network in packets and was processed in packets. Unfortunately, some pieces of processing code had an input size hard-wired into them and it was inconvenient to have to modify them so that they could handle variable-sized inputs from packet queues. Using packet queues also made it possible to cut down on much of the data copying that was necessary when using ring buffers; however, some functions require internal buffering of input
data, and some functions produce more data bytes than they consume, requiring them to allocate additional memory for the output data packet.

3.5 Implementation of Function Objects

New function objects can be written for the DSPCore server framework and compiled into a server program. Several function objects were implemented for the server as part of this thesis. Some are referred to in the next section, all are listed in Section 5.1, and Appendix A contains their ILU interface descriptions.

Each function object must be described in an ILU interface file, as an object with DSPCore.FunctionT as a superclass, from which it will inherit the method DIE. If the function object is to receive data from other function objects in the server, it will need an InPort, so its interface must include a call that returns its InPort to the client; similarly for sending data to other function objects, it must have an OutPort that the client can control. Parameters that control the operation of the processing code can also be made available to the client through additional calls in the ILU control interface. Calls can also be added to the interface to show the client the state of the function object to assist in debugging. See Figure 3-2 in Section 3.3 for an example of a function object's ILU description.

Each function object's implementation must include a structure named IfcObjSpecificData, where Ifc and Obj are the ILU interface and object names. The first nine fields of this structure must be identical to the first fields of the struct SD defined in the union GD in DSPCore-config.h, and IfcObjSpecificData must be a member of that union, so that certain fields of any function object's IfcObjSpecificData can be accessed by server core code that knows nothing else about function object code. The first eight fields are pointers to "hook" procedures that InPort and OutPort calls use to notify a function object when the state of one of its ports has changed, or to get its permission to form a connection. The ninth field carries the process/thread identifier of the function object's processing thread. After the first nine mandatory fields, the IfcObjSpecificData structure can have any fields that
the module writer requires. These must include all data associated with a function object, including its InPort and OutPort objects. All global variables from the processing code should be moved to fields of this structure, so that multiple function objects in the server can run the same code without overwriting each other's data.

An initialization routine for the IfcObjSpecificData structure, named Ifc_Obj._initializeSD, must also be implemented. It must set the hook procedure pointers to actual hook routines or to NULL. If the function object has InPorts or OutPorts associated with it, then the initialization must set the corresponding fields to the ILU objects returned by DSPCore.InPort.New and DSPCore.OutPort.New.

The data processing code is part of the function object's Ifc_Obj.mainloop routine, which executes as the thread spawned by the Broker's createFunction. This routine should perform whatever initializations are necessary; its IfcObjSpecificData, which it will get as an argument, will already be initialized, but it may have local variables to initialize as well. Then it should loop continuously, getting data from InPorts or other sources (network, file, etc.), processing the data in chunks, and outputting the data to its OutPorts or other destinations. See Figure 3-4 for the main loop routine of the rate conversion function object.

In order for the function object to be successfully integrated into a server program, the header file DSPCore-config.h must be modified to contain a few pieces of information about the object. The definition of the procedure make.TaskInitInfo must be updated to include an entry for the function object, which includes all information necessary to create a new instance of the object: its name (Ifc_Obj), the size of its IfcObjSpecificData structure, a pointer to its ILU object creation procedure, a pointer to its IfcObjSpecificData initialization procedure, and a pointer to its mainloop procedure. The function object's IfcObjSpecificData structure must be added to the GD union, which is used for typecasting to a generic record of hook routine pointers. Its ILU initialization call, Ifc.InitializeServer(), must be added to the server's main routine.
/* typedef the object's RateConvSpecificData structure */
#define RCSD ((RateConvSpecificData *)(self->data))

void Rate_Conv_mainloop (DSPCore_FunctionT self) {
    Packet *inpkt;
    void *newd;
    int outlen;

    while (1) {
        /* wait for data to arrive */
        inpkt = PQ_getPacket (RCSD->inq);

        /* output will not be the same size as input, so get allocate */
        /* a different chunk of mem for it */
        newd = malloc ((inpkt->len * RCSD->Factor + 1) * sizeof(short));

        /** do the rate conversion **/
        outlen = resample (self->data, inpkt->data, newd, inpkt->len);

        /* set up the packet with the new chunk of mem and length */
        free (inpkt->data);
        inpkt->data = newd;
        inpkt->len = outlen;

        /* pass the data on */
        /* this may block us if the connection's overwrite param is */
        /* set false and too much data has built up in the connection */
        PQ_addPacket (RCSD->outq, inpkt);
    }
}

Figure 3-4: Example processing module main loop routine
3.6 "Utility" Function Objects

The server core can stand on its own as a body of code and can be extended with any function objects that conform to the rules, but there are a few basic function objects that are necessary in order for any processing to be done, and a few others that are useful building blocks for complex tasks.

It is absolutely necessary to have function objects that can take in data from the network and send data out over the network. Processing can only take place if there are sources and sinks of data within the server. The server implemented for the thesis includes two RTP function objects: RTP.Src, which receives data from the network and serves as a source of data to other function objects in the server, and RTP.Snk, which takes data from other function objects in the server and sends it out as RTP packets. The RTP objects are not hard-coded into the design. For example, a VAT protocol source and sink could have been used. The choice of RTP did influence the design of the server, however (note the seq field of the queue packets shown in Figure 3-3).

There are several different data formats and rates used by common workstation audio devices. For example, Sun SPARC 1's and 2's use only 8kHz, 8-bit mu-law encoded data, while SGI Indy's can use linear encoded 8-, 16-, or 24-bit data at varying from 8kHz to 48kHz. Also, different pieces of DSP code are written for different data formats. Therefore, in order to support processing on data from differing platforms, format and rate conversion objects are necessary. The implementation of the server included format converters to convert between any pair of linear, mu-law, and floating point formats, and a rate conversion object that handled linear data. Rate and format conversion are such basic operations that even current audio servers such as NetAudio generally have them built in.

There are many other function objects that would be useful in forming complex tasks. An output-tee object that would pass data from a single input to multiple outputs would be necessary in order to send a data stream to multiple function object inputs in a server. A multiple-input synchronizer object could be written to
deal with differential delay complexities, so that writers of multiple-input function objects would not have to worry about it. A bit-bucket object, to store some amount of data and then play it back on demand, could also be a useful building block.
Chapter 4

Clients and Supporting Software

In order to test the server, sources and sinks of realtime audio data on the network were written for some available workstations with audio devices. An extension to the Tcl scripting language was created so that clients could be written as simple Tcl scripts, and simple clients were written to test the server's function objects.

4.1 Network Sources and Sinks of Audio

Audio data sources and sinks for the server were written for Sun SPARC with 8kHz mulaw audio devices and SGI Indys whose audio devices use linear encoded 1-, 2-, or 4-byte samples at 8kHz to 48kHz. Sources took audio data from the workstation's audio device or a file, packaged it as an RTP stream, and sent it to some destination machine and port. Sinks received data as an RTP stream and either played it on the workstation's audio device or wrote it to a file. Control commands were sent to the sources and sinks in single UDP packets following the format shown in Table 4.1. Extension commands were added to a Tcl/Tk shell for each source/sink control command (see the next section for more information on Tcl/Tk and extension commands). A script was written in the extended Tcl/Tk to provide a graphical user interface for sending commands to sources and sinks.
Commands interpreted by both sources and sinks:

<table>
<thead>
<tr>
<th>command</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>none</td>
<td>start sending or receiving data</td>
</tr>
<tr>
<td>STOP</td>
<td>none</td>
<td>stop sending or receiving data</td>
</tr>
<tr>
<td>SETDATAPORT</td>
<td>p:integer</td>
<td>set data port to p and control port to p+1</td>
</tr>
<tr>
<td>SETSSRC</td>
<td>ssr:integer</td>
<td>set RTP SSRC (identifier) to ssr</td>
</tr>
<tr>
<td>SETUSEFILE</td>
<td>0 or 1</td>
<td>if 0, use audio device; if 1, use file</td>
</tr>
<tr>
<td>SETFILE</td>
<td>name:string</td>
<td>set filename to name</td>
</tr>
</tbody>
</table>

Commands interpreted by sources only:

<table>
<thead>
<tr>
<th>command</th>
<th>arguments</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETDESTHOST</td>
<td>name:string</td>
<td>set destination host (for RTP) to name</td>
</tr>
<tr>
<td>SETDESTPORT</td>
<td>port:integer</td>
<td>set destination data port (for RTP) to port</td>
</tr>
<tr>
<td>SETPACING</td>
<td>0 or 1</td>
<td>if 0, send packets as fast as possible; if 1, pace output by reading from the audio device. (Only makes sense when sending from a file.)</td>
</tr>
</tbody>
</table>

Table 4.1: Device control (devctl) packet formats

4.2 A Tcl Extension for Clients

Tcl[12] is a scripting language for which extension commands can be fairly easily implemented in C. Tcl is interpreted instead of compiled, so code can be changed and retested without waiting for compilation. Its syntax is mostly straightforward, much like shell scripts. There are two data types, the string and the associative array (hash table); statements are of the form command arg1 arg2 ... argn. Tk, an extension of Tcl for X windows, makes it possible to create X window interfaces to Tcl programs with very few lines of code. Because Tcl is interpreted, it is not as efficient as compiled programming languages such as C. It is awkward and inefficient for numerical computations or for any computation that relies on following pointers, because Tcl does not have numerical data types or pointers. However, new commands can be implemented in C for higher performance and flexibility. For these reasons, Tcl is popular as a high-level programming interface to systems whose lower levels can be grouped into commands and implemented in C.

43
Since a Tcl extension for clients of our server enables applications to be quickly prototyped and modified, extension commands were written for each ILU client call in the Broker, InPort, OutPort, and all function object interfaces. The next section shows excerpts of a test client written in the extended Tcl.

VuSystem[9] (also described in Section 2.3) provides Tcl extension commands to control its processing modules, with the same goal of making it easy to write applications. However, VuSystem's extended Tcl shell includes the data processing code as well as the extension commands, while ours is simply a wrapper around ILU C client calls.

### 4.3 Test Clients

Each function object implemented for the server was used by one or more test clients written in the extended Tcl. An example test client (testing the time compression object) is shown in Figure 4-1. RTP objects were used by all test clients to get data in and out of the server. Format converters were often required to change the input data to the encoding expected by the function object's processing code. For example, if the original source of data was a mu-law encoding audio device and the function object was written to operate on floating point numbers, the and mu-law-to-float conversion object was needed. Rate conversion was needed if source and sink devices used different sampling rates. Rate conversion can also be used to lower the
sampling rate at which processing is performed to speed up the processing at the price of sound quality.

Figure 4-2 shows parts of the extended-Tcl client script corresponding to the test client in Figure 4-1. Each client script followed the same outline:

1. Connect to the server and get a Broker object handle.
2. Ask the server to create the processing objects and return their handles.
3. Get handles for each processing object’s InPort and OutPort objects.
4. Initialize parameters of each function object.
5. Connect InPorts and OutPorts.

Once these steps have been followed and data is available to the first function object(s) in the graph, processing begins.
# 1. connect to the broker (string binding handle comes from stdin)
set brokerID [broker_createFromSBH $brokerSBH]

# 2. get processing objects from broker
set srcID [broker_createFunction $brokerID "RTP.Src"]
set m2fID [broker_createFunction $brokerID "Fmt.Mulaw2Float"]
set tcID [broker_createFunction $brokerID "TC.Clutch"]
set f2mID [broker_createFunction $brokerID "Fmt.Float2Mulaw"]
set snkID [broker_createFunction $brokerID "RTP.Snk"]

# 3. get InPorts and OutPorts from processing objects
set srcOutID [rtpSrc_getOutput $srcID]
set m2fInID [fmtM2F_getInput $m2fID]
set m2fOutID [fmtM2F_getOutput $m2fID]
set tcInID [tc_getInput $tcID]

... 

# 4. initialize parameters
#set up RTP Src to receive from the machine klono's audio device (NPR)
rtpSrc_setSSRC $srcID 1111
rtpSrc_setHostInfo $srcID klono 0 14986 29395
rtpSrc_setDataFormat $srcID 1 1 1 8000

#set up mulaw to float converter
fmtM2F_setRate $m2fID 8000
fmtM2F_setNumChans $m2fID 1

#set up time compression object
tc_setFactor $tcID 2.0
tc_setNumChannels $tcID 1
tc_setRate $tcID 8000

... 

# 5. connect InPorts and OutPorts
outPort_connect $srcOutID $m2fInID
outPort_connect $m2fOutID $tcInID
outPort_connect $tcOutID $f2mInID
outPort_connect $f2mOutID $snkInID

Figure 4-2: Excerpts from example test client code
Chapter 5

Results and Observations

This chapter discusses the final state of the system implemented for the thesis and some lessons drawn from our experiences. Section 5.1 lists the function objects included in the server. Section 5.2 reiterates our goals of allowing clients certain controls over buffering of data and timing of processing, evaluates how each goal was met or not, and describes two additional timing concerns. Section 5.3 discusses the simple client applications built to test the server and Section 5.4 comments on the process of generating new function objects from existing code. Areas of future work are discussed in Section 5.5.

5.1 Function Objects Implemented

Six ILU interfaces, each defining one or more function objects, were written and their objects were implemented. Their ILU interface descriptions are in Appendix A. The RTP ILU interface included \texttt{Src} and \texttt{Snk} objects. \texttt{Src} received RTP data from the network and acted as a source of data inside the server. \texttt{Snk} acted as a sink of data inside the server and sent data over the network as an RTP stream. The \texttt{Fmt} interface included six data format conversion objects, one for each pair of mu-law, linear, and floating-point. The \texttt{Rate} interface defined \texttt{Conv}, a rate conversion module that operated on linear data, and \texttt{Scale}, which changed the timescale of its (linear data) inputs by resampling while declaring the outputs to have the same
sampling frequency. The **Big** interface defined a single module, **Buf**, for recording and playing back sound segments. The **Mixer** module defined two-input mixers for mu-law and linear data. The **TC** module defined a single module **Clutch**, which performed time compression on floating point data; in other words, it changed the timescale of the inputs without changing the pitch by duplicating or removing similar portions of the input. See the discussion of timescale-changing operations in Section 2.5.3 for a description of linear timescaling, implemented by **Rate.Scale**, and time compression for "time clutching", implemented by **TC.Clutch**.

### 5.2 Control over Output Timing

Initially we hoped to guarantee near-zero jitter from our server. We reasoned that if we used a reliable timer to drive processing, and buffered up enough input data to insure against the jitter of the inputs, then we could practically eliminate output jitter at the price of adding some delay. This would reduce the amount of buffering required for sink devices. We used ring buffers as the mechanism for passing data between devices so that packet sizes would be transparent to processing code; the code would simply take the appropriate amount of data from the ring buffers when told to process.

Unfortunately, we found that most workstation operating system timers are not very reliable at short periods (20-50 milliseconds), and UNIX processes can be descheduled for up to 100 milliseconds. Multiple timers (for multiple processing threads) are either not available or even less reliable. Function objects with multiple inputs had to perform complex checks on input buffers to make sure that the desired data was there, and the decision of how far one input could fall behind before it was necessary to proceed without it was made by the processing module, not the client.

We came to the realization that in most cases there was no benefit to using a timer to drive processing, because if the data is not there then the module must either wait or lose data. In most cases, especially if input streams would have good timing characteristics, we would add the least delay and jitter by using the arrival
of input packets instead of a timer to drive processing. In other words, a processing module would wait for input data before deciding whether or not to produce outputs. If some application absolutely required timer-driven outputs, then a pacing module could be used after the input-driven function objects. It could use a timer to drive processing while using the same input-driven mechanism for getting input data.

As long as processing takes the same amount of time for each input packet and begins the same amount of time after the packet's arrival, no additional jitter is induced by input-driven processing. Even if varying processing times do add jitter, there is no more benefit to de-jittering the stream within the server than to de-jittering at the final destination, except that the final destination's buffering requirements might be reduced.

Since input packets were used to drive processing, the mechanism for passing data within the server was changed to a packet queue. This greatly simplified the structure of function objects with one input and one output: instead of checking to see if the desired input samples were present in a ring buffer and deciding what to do if they weren't there, the objects simply waited for an input packet, processed it, and passed on the results. Function objects with multiple inputs still had to decide what to do when one input fell behind, but it was still simpler to assess the state of a packet queue than a ring buffer.

In order to move timing and buffering decisions out of the processing module code and into server core code, the packet queues were given four buffering and timing parameters: \texttt{constDelay}, \texttt{maxBuf}, \texttt{maxSlip}, and \texttt{overwrite}. (See Subsection 3.4.3 for a description of each parameter.) These were intended to allow clients the following controls over each link, \textit{without} requiring the function objects to contain any special code:

- set an initial delay that could be used as insurance against jitter (\texttt{constDelay})
- in a multiple-input processing module, decide the tradeoff between waiting for late-arriving packets from one stream and losing their data (\texttt{maxSlip})
- control the amount of data that can build up in a link, and whether the link should push back on the producer or flush itself when that amount of data is succeeded (\texttt{maxBuf}, \texttt{overwrite})
The first two goals were partially attained and the third was met.

Accumulating data in a link before beginning to process any of the data, thus adding delay, acts as insurance against jitter if a more regularly timed sequence of events is used to drive processing than the arrival times of the incoming packets. This more regularly timed sequence could come from a timer or another input stream. We abandoned timers because they were not reliable, so the constDelay parameter only mattered when the link was an input of a module using a more regularly timed input to drive its processing. When the link was the only input to a module, setting an initial delay did not guard against jitter because no more reliable source of timing was available.

The maxSlip parameter specified the largest amount of data that could build up in a link before processing had to proceed regardless of whether the consumer module was still waiting for data to arrive from another input. The packet queue routines did not pay any attention to the maxSlip parameter. There was only a routine to wait for data to arrive in any queue in a set. It was left to the mainloop routines of function objects with multiple inputs to enforce the maxSlip parameter, contrary to our goal of keeping timing issues hidden from processing module code.

The buffering behavior of links is not dependent on input timing or the behavior of other inputs to the consumer module, so these complications did not affect the client’s control over the maximum amount of buffering and the push-back vs. overwrite behavior of the links.

When testing the server with audio sources and sinks on the network, there was a perceptible delay between the time the user would tell the ultimate source or sink audio device to stop sending or playing data and the time the sound would cease. This was due both to buffering inside the server’s function objects and to buffering in the audio device that served as the ultimate sink. To get around this problem we could have added a flush command to the RTP-based data transport protocol and to the queues used for passing data between function objects within the server.

Some processing operations, such as timescaling and time compression, do not produce and consume data at the same rate though the nominal sampling rates of the
input and output data are the same. Therefore, such an operation can have either realtime inputs or realtime outputs, but never both. For example, a timescaling module set up to slow down an 8kHz realtime input sound by a factor of 2 would consume 8 kilosamples per second but would produce 16 kilosamples (nominally sampled at 8kHz) per second. Its output could not be played in realtime because it would take twice as long to play the output as the input. On the other hand, consider a timescaling module set up to speed up an 8kHz realtime input sound by a factor of 2. It would consume 8 kilosamples per second and produce only 4 kilosamples per second, too few samples to play in realtime.

Since timescale-changing operations can't have realtime inputs and outputs, they do not directly map onto our model of realtime data streams and processing elements, which assumes realtime inputs and outputs. In order to map a timescale-changing process and its input and output onto our model of realtime streams, we can scale our expectations of how much data should arrive or how much should be produced by the timescaling factor. This is essentially the same as treating the timescale change as a sampling rate change. We still deal with a realtime flow of data, but the amount of data on one side is larger than the amount of data on the other side. For example, consider a time compression module that receives data at twice realtime speeds from a file, compressing the timescale by a factor of 2 and outputting realtime data. Even though the input is arriving faster than realtime, we can treat it as a realtime stream that simply uses a sampling rate twice as large as the output uses, thus twice the number of samples enter the time compression module as leave it.

5.3 Constructing Clients

All clients were scripts interpreted by the extended Tcl shell described in Section 4.2, following the general setup sequence described in Section 4.3. Using Tcl with extension commands for client calls made it very easy to write and modify scripts and to create simple graphical user interfaces when desired.
The clients were extremely simple, designed to test and demonstrate the operation of server function objects. They were not compelling applications in themselves.

5.4 New Function Objects from Old Code

Existing data processing code was incorporated into function objects in the Rate, TC, Fmt and Mixer objects. Code for converting between linear and mu-law data formats and for mixing mu-law data, written by Ron Frederick of PARC, was used for the Fmt.Lin2Mulaw, Fmt.Mulaw2Lin, and Mixer.TwoMulaw objects. Time compression code used for TC.Clutch was also available at PARC. Rate conversion code originally written for a Stanford AI Lab (SAIL) project, then translated into C and modified by others, was used for Rate.Conv and for Rate.Scale, the linear timescaling object.

It was not as easy as was hoped to turn existing code into new function objects. It takes time to understand code written elsewhere well enough to identify which variables are transients and which are parameters, and which parameters should be settable by clients. Also, not all code is written to process variably sized arrays in memory, which function objects must be able to do. Instead, some code is written to process hard-coded chunks of data from an input file to an output file.

The bulk of the code for a processing module was very predictable. Each processing module needs ILU server implementations for each call defined in its interface, an initialization routine for its IfcObjSpecificData structure, and a mainloop routine. Tcl client wrappers should be defined for each new processing module as well. Much work could be saved by a simple stub-generator that parses the ILU interface file and generates templates for each of those, leaving the meaningful parts of the code to be filled in by the programmer.

5.5 Future Work

The server built for the thesis was a program that ran on a standard workstation, but for better performance the server could be rewritten to run on a workstation
with special DSP hardware whose operating system was tuned to move data between the DSP hardware and the network as fast as possible. This would involve nontrivial changes to the structure of the server, because the processing code would run on the DSP hardware while the control (ILU) code would run on the workstation's host processor. The code running on the DSP hardware would need a scheduler that would probably be unrelated to the ILU code, except when function objects were created and destroyed. Our input-timed approach would make the scheduler's task much simpler than a timer-driven approach.

Several higher-level toolkits could act as intermediaries between applications and the client-server interface to smoothe over tedious details of the system. For example, a toolkit could be provided to insert the necessary rate and format conversion objects when the application requests a connection between ports with incompatible rates or data types. Another use for a toolkit would be to hide network transport details. Applications could request a connection between some network device and some processing functions. The toolkit would find a server, set up a network connection between the device and the server, and setup up RTP and processing objects in the server.

Resource management would be a useful addition to the server; the server would know when it was about to get overloaded and would refuse requests to create new function objects. Resource discovery would allow clients to query servers not only about what function objects they support, but also about the data rates at which they can perform those functions.

The server built for the thesis includes new function object modules by compilation, so server downtime is necessary when extending the server's set of function objects. Dynamic registration and loading of function objects would remove this necessity and would make server configuration a much simpler task overall.

Finally, more work is required to truly offer clients control over the timing of processing and the decision of when to process input data. Parameters that only matter for function objects with multiple inputs, such as the relative delay parameter maxSlip, should be separated out from universally applicable parameters, such as
the buffering parameters. A mechanism for applying these parameters should be developed so that processing module code never has to interpret and implement them. Special function objects that serve as synchronizers or pacers might serve this purpose.
Chapter 6

Conclusion

This thesis has presented arguments for distributing the processing of realtime data streams by placing processing functions in networked servers, the design of an extensible framework for such servers, and experiences gained by building the server and simple client applications.

The server framework and several function objects were built. Some of the function objects were created using data processing code from external sources. Simple clients were written to test the server objects and demonstrate the server’s realtime operation.

Using the arrival of data inputs to time the production of outputs was found to be much better overall than using a timer. Added delay and jitter were generally smaller with the input-timed approach and the mechanism for passing data between function objects was easier for function objects’ processing code to use.

Although continuous processing happened smoothly, without gaps between the playing of output packets, starting and stopping were not as responsive as would be desirable. The timing and buffering parameters of the data ports allowed clients to specify the buffering behaviors of connections between function objects and gave clients some control over the timing of processing relative to the timing of input packets. However, the parameters did not completely characterize the timing of processing as we had hoped, and the mechanism for passing data between function objects did not fully remove timing concerns from the processing code of function
objects with multiple inputs.

Client scripts for testing server function objects were written with very little effort in an extended version of the Tcl scripting language. However, the clients built were extremely simple, not very compelling applications in themselves.

Although already-written processing code was successfully incorporated into the server, it was not as easy to do this as we had hoped, and debugging new function objects was complicated by the lack of protection boundaries between separate running tasks within a server.

The fact that standard workstations and LAN were used successfully for distributed processing on realtime audio streams provides a compelling demonstration of our ideas about distributing realtime DSP by means of a networked server architecture.
Appendix A

ILU Interface Descriptions for Server Objects

A.1 DSPCore.isl

INTERFACE DSPCore IMPORTS ilu END;

(** the main server interface - clients ask this for function objects ***)
EXCEPTION NotSupported;
EXCEPTION OutOfMem;
EXCEPTION OutOfTime;

TYPE Broker = OBJECT
  METHODS
    listActive () : TaskList,
    listAvailable () : FunctionList,
    createFunction (newf:FunctionID) : FunctionT
      RAISES NotSupported, OutOfMem, OutOfTime END
  END;

(** Superclass objects for all function modules - methods to return **)  (** input and output ports are now entirely up to the module writers. **)
EXCEPTION BadValue : INTEGER;
EXCEPTION NotConnected;

TYPE InPort = OBJECT
    METHODS
        GetPolicy () : PolicyInfo,
        SetPolicy (pol: PolicyInfo)
            RAISES BadValue END,
        GetSrc () : OutPort
            RAISES NotConnected END
    END;

EXCEPTION NonLocal;
EXCEPTION Incompatible;

TYPE OutPort = OBJECT
    METHODS
        Connect (inport:InPort)
            RAISES NonLocal, Incompatible END,
        Disconnect ()
            RAISES NotConnected END,
        GetDest () : InPort
            RAISES NotConnected END
    END;

(** data types used in calls **)  
TYPE PolicyInfo = RECORD
    maxBuf: INTEGER,
    constDelay: INTEGER,
    maxSlip: INTEGER,
    overwrite: BOOLEAN
    END;

TYPE FormatInfo = RECORD
    formatID: INTEGER,
    byteWidth: SHORT INTEGER,
    numChans: SHORT INTEGER,
    rate: INTEGER
    END;

TYPE TaskID = ilu.CString;
TYPE TaskList = SEQUENCE OF TaskID;

TYPE FunctionID = ilu.CString;
TYPE FunctionList = SEQUENCE OF FunctionID;
A.2 RTP.isl

INTERFACE RTP IMPORTS DSPCore END;

EXCEPTION BadValue: INTEGER;

TYPE HostName = ilu.CString;
TYPE HostID = INTEGER;
TYPE PortID = INTEGER;

TYPE HostInfo = RECORD
  Name: HostName,
  ID: HostID,
  dataport: PortID,
  ctIport: PortID
END;

TYPE RTPEntity = CLASS
METHODS
  GetSSRC () : INTEGER,
  SetSSRC (ssrc: INTEGER)
    RAISES BadValue END,
  GetHostInfo () : HostInfo,
  SetHostInfo (hinfo: HostInfo)
    RAISES BadValue END,
  GetDataFormat () : DSPCore.FormatInfo,
  SetDataFormat (fmt: DSPCore.FormatInfo)
    RAISES BadValue END
END;

TYPE Src = CLASS
SUPERTYPES DSPCore.FunctionT, RTPEntity END
METHODS
  GetOutput () : DSPCore.OutPort
END;

TYPE Snk = CLASS
SUPERTYPES DSPCore.FunctionT, RTPEntity END
METHODS
  GetInput () : DSPCore.InPort
END;

59
A.3  Fmt.isl

INTERFACE Fmt IMPORTS DSPCore END;

EXCEPTION BadValue:INTEGER;

(* each format conversion module will have 1 input, 1 output, *)
(* and variable rate and # of channels. *)
TYPE Blank = CLASS
    METHODS
        GetInput () : DSPCore.InPort,
        GetOutput () : DSPCore.OutPort,
        SetRate (rate:INTEGER)
            RAISES BadValue END,
        SetNumChannels (nc:INTEGER)
            RAISES BadValue END
    END;

(* actual conversion objects *)
(* these are subclasses of DSPCore.FunctionT and Blank, above *)
(* the ones that input or output linear data can be told to deal *)
(* with 1, 2, or 4-byte linear samples. *)

TYPE Lin2Mulaw = CLASS
    SUPERTYPES DSPCore.FunctionT, Blank END
    METHODS
        SetLinBytesPerSamp (Bps:INTEGER)
            RAISES BadValue END
    END;

TYPE Mulaw2Lin = CLASS
    SUPERTYPES DSPCore.FunctionT, Blank END
    METHODS
        SetLinBytesPerSamp (Bps:INTEGER)
            RAISES BadValue END
    END;

TYPE Float2Mulaw = CLASS
    SUPERTYPES DSPCore.FunctionT, Blank END;

TYPE Mulaw2Float = CLASS
    SUPERTYPES DSPCore.FunctionT, Blank END;

TYPE Lin2Float = CLASS
    SUPERTYPES DSPCore.FunctionT, Blank END
    METHODS
        SetLinBytesPerSamp (Bps:INTEGER)

RAISES BadValue END

END;

TYPE Float2Lin = CLASS
  SUPERTYPES DSPCore.FunctionT, Blank END
  METHODS
    SetLinBytesPerSamp (Bps:INTEGER)
      RAISES BadValue END

END;
A.4 Rate.isl

INTERFACE Rate IMPORTS DSPCore END;

EXCEPTION BadValue: INTEGER;

TYPE Conv = CLASS
  SUPERTYPES DSPCore.FunctionT END
  METHODS
    GetInput () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort,
    SetInputRate (r: INTEGER)
      RAISES BadValue END,
    SetOutputRate (r: INTEGER)
      RAISES BadValue END
  END;

TYPE Scale = CLASS
  SUPERTYPES DSPCore.FunctionT END
  METHODS
    GetInput () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort,
    SetRate (r: INTEGER)
      RAISES BadValue END,
    SetScale (s: SHORT REAL)
      RAISES BadValue END
  END;
A.5 Mixer.isl

INTERFACE Mixer IMPORTS DSPCore END;

EXCEPTION BadValue : INTEGER;

(* note: this isn't a DSPCore.FunctionT subtype - it just defines *)
(* some methods that all two-input mixers will have *)
(* (we couldn't do this before multiple inheritance in 1.6...) *)
TYPE TwoInputMixer = OBJECT
  METHODS
    Start (),
    Stop (),
    GetInputA () : DSPCore.InPort,
    GetInputB () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort
  END;

TYPE TwoLinear = OBJECT
  SUPERTYPES DSPCore.FunctionT, TwoInputMixer END
  METHODS
    SetFormat (fmt: DSPCore.FormatInfo)
      RAISES BadValue END,
    SetOffset (numSamps: INTEGER)
      RAISES BadValue END
  END;

TYPE TwoMulaw = OBJECT
  SUPERTYPES DSPCore.FunctionT, TwoInputMixer END
  METHODS
    SetOffset (numSamps: INTEGER)
      RAISES BadValue END,
    SetInGainA (gain: SHORT INTEGER)
      RAISES BadValue END,
    SetInGainB (gain: SHORT INTEGER)
      RAISES BadValue END,
    SetOutGain (gain: SHORT INTEGER)
      RAISES BadValue END
  END;
A.6 TC.isl

INTERFACE TC IMPORTS DSPCore END;

EXCEPTION BadValue;

TYPE Clutch = CLASS
  SUPERTYPES DSPCore.FunctionT END
  METHODS
    GetInput () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort,
    GetFactor () : SHORT REAL,
    SetFactor (f: SHORT REAL)
      RAISES BadValue END,
    SetNumChannels (n: INTEGER)
      RAISES BadValue END,
    SetRate (r: INTEGER)
      RAISES BadValue END,
    Flush ()
  END;
A.7 Big.isl

INTERFACE Big IMPORTS DSPCore END;

EXCEPTION BadValue: INTEGER;

TYPE Buf = CLASS
  SUPERTYPES DSPCore.FunctionT END
  METHODS
    GetInput () : DSPCore.InPort,
    GetOutput () : DSPCore.OutPort,
    Clear (),
    Recrd (),
    Play (),
    Pause (),
    Stop (),
    Loop (),
    NoLoop (),
    GetLen () : INTEGER,
    SetMaxLen (1: INTEGER)
      RAISES BadValue END
  END;
Bibliography


[13] Andrew Payne. Ak, TCL extension for AudioFile, available via ftp from crl.dec.com as /pub/misc/ak-0.7.tar.Z.

