Sprint Voice Transport Over ATM Network: Real-Time Implementation

Khoi Nguyen
David W. Petr
Joseph Evans
Victor Frost

Telecommunications & Informations Sciences Laboratory
The University of Kansas
Lawrence, KS 66045-2228

July 1995

Abstract

As ATM (Asynchronous Transfer Mode) is coming in the near future, accompanied by its high-speed transmission and switching backbone, real-time multimedia applications such as teleconferencing have found it a good match for voice and data transport. Telephone -grade speech, conventionally transported over a circuit-switch telephone network, is no exception in finding multimedia potential in ATM. In fact, using the AAL1 (ATM Adaptation Layer Type 1) for constant bit rate services, transporting 64 Kbits/s PCM voice over an ATM network has been widely supported and researched.

This report describes the real-time implementation of the neccessary interfaces when such a transportation is not only over the ATM network but also over the existing PSTN (Public Switched Telephone Network). In this case, the ATM network will serve as an intermediate network for which voice quality accessments under controlled cell loss and delay conditions can be made. In particular, there are two interfaces that must be implemented: the telephone-ATM network interface, and the ATM network-to-PSTN interface. Because of the availability of hardware interfaces, only software implementation was actually carried out; however, both hardware and software implementations are discussed in detail.

Contents

List of Figures

List of Tables

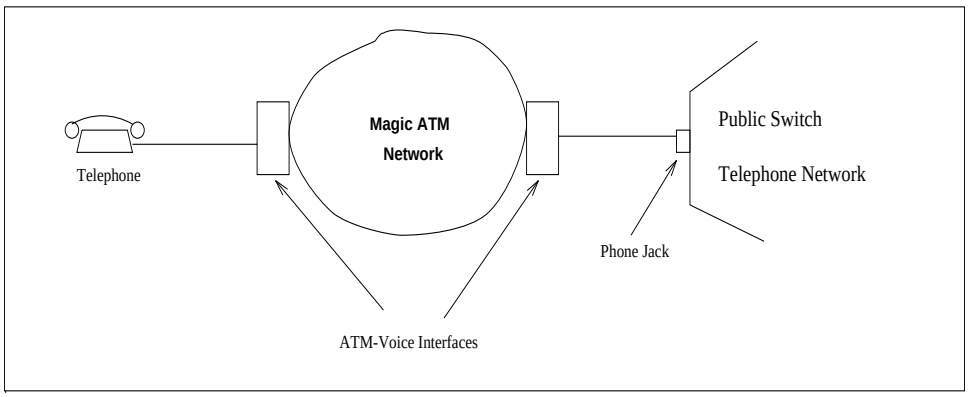

Figure 1: Voice-Over-ATM Prototype Configuration

1 Introduction:

1.1 Implementation Goals:

This technical report concerns the real-time implementation of voice on an ATM network, as part of the Voice On ATM Project funded by Sprint. In particular, it will describe in detail the implementation of a real-time demonstration system, which will provide a tool for evaluating voice over ATM networks. The implementation of the system is based on the proposed system solutions as described in [1], and is matched as closely as possible to the design proposed in [2]. This research is being coordinated with other ongoing research efforts at KU, i.e., the Sprint-funded project "Development of Design Rules and Associated Tools for ATM Networks" and the MAGIC testbed project.

To support voice over both an ATM network and the PSTN network, two interfaces need to be implemented: the phone-ATM network interface and the ATM network-PSTN interface. Figure 1 shows the desired configuration. During the whole course of this project, our goal has been that the implementations will be based as much as possible on existing supported software and equipment. As initially proposed, the interface devices could be interfaced directly to the MAGIC network through DEC AN2 ATM switches. But after carrying out feasibility studies, it was found that DEC Alpha workstations are the best interface choices. Three main reasons behind this decision are: their performance, software support under OSF/1, and most importantly, the less complex nature of the implementation. Another reason why Alpha workstations are chosen is that the "otto" board which interfaces to the ATM is implemented for use with Alphas. Thus, software development around the otto device driver would be possible.

1.2 Hardware and Software Considerations:

To implement the two interfaces, both hardware and software design and implementations were considered. In the initial project proposal, a hardware interface was to be developed that, besides having all the neccessary interfacing functionalities, would allow for flexibility to be added in the future. The possible flexibility includes using digital signal processing (DSP) chips, field programmable gate arrays (FPGAS), and/or microprocessors. We initially proposed to implement such a prototype interface; however, as it turned out, Digital Equipment Corporation (DEC) has a commercial product known as DECaudio which has most, if not all, the hardware functionality neccessary for making the interface possible. Section 3 will give an overview of the DEC Alpha's audio capabilities, and the DECaudio audio device. Following this section, a description of the software required to drive these audio devices will be presented in section 4.

2 Summary:

2.1 A Review Of The Work Done:

The work actually carried out in this project is more than we initially planned, particularly due to unexpected hardware problems. As mentioned earlier, there was no hardware development involved due to the availability of DECaudio, an audio device which has the telephone interface capability. All the work done was on software development. In particular, much work has been focused on implementating the system solutions proposed in [1] and [2]. As far as the software development is concerned, the developed demonstration tool has almost all the functionalities as proposed in the system solutions. Some cases in [2] were not implemented because they either rarely occur in ATM networks (the case with bit errors in HEC, SN, or SNP) or they are not suitable to be implemented in software (the case concerning the idle state).

In implementing the proposed model as accurately as possible, extensive modifications to an existing software package, namely *AudioFile*, were carried out. Extensive work on echo cancellation was also done; unfortunately, however, all attempted approaches could not eliminate the echo problem that was discovered in the DECaudio hardware.

To enhance the demonstration capability of the developed demonstration tool, graphical user interfaces (GUIs) were implemented. With these GUIs, the complexity of the applications developed is effectively hidden, providing the user

an easy-to-use tool to evaluate voice quality under various controlled loss and delay conditions.

2.2 Capabilities Available On The Demonstration System:

Two GUIs, namely the Alpha-Alpha GUI and the Alpha-DECaudio GUI, were implemented to enhance the demonstration capability. Section 7 provides more detail about the GUIs. With the Alpha-Alpha GUI, users (local and remote) can talk over an Ethernet or ATM network through handsets attached to the hosts. The voice transport accurately emulates the voice-over-ATM system presented in [2]. Each user can change cell loss rates, cell delay and transmit/receive gains during the connection. Since the GUI is menu and button-based, users would find it a very convenient and easy-to-use tool.

In addition to the Alpha-Alpha GUI capabilities, the Alpha-DECaudio GUI allows the user to dial any telephone and conduct a full-duplex telephone conversation using a handset attached to the host.

2.3 Limitations Of The Demonstration System:

As discussed in section 8, the DECaudio device has serious echo and noise problems. Different approaches were carried out in attempting to solve the problems; unfortunately, none have yielded any successful result. The echo and noise problems on the DECaudio device significantly limit the demonstration capability of the Alpha-DECaudio GUI. However, only the handset side has the noise and echo problems; the remote telephone side has proved to be quite clear and echo-free. Thus, the Alpha-DECaudio GUI could be still an useful demonstration tool if users are only concerned about the voice quality on the remote telephone side. The Alpha-Alpha capability does not have these limitations because it does not use the DECaudio device.

3 Audio Hardware:

3.1 DEC Alpha's Base Board Audio:

As shipped with all Alphas, an audio port, called the Baseboard, is available for use in audio proccessing. The Baseboard includes a CODEC device operating at

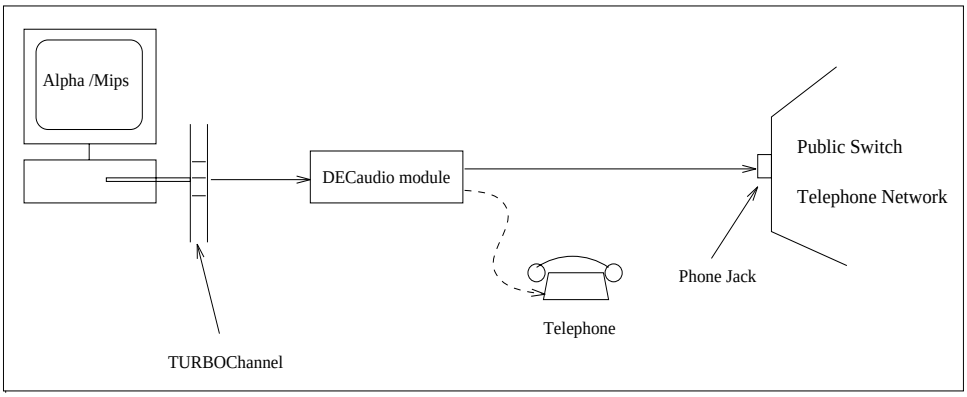

Figure 2: DECaudio-Phone Configuration

a fixed sampling rate of 8KHz (8000 samples/sec); thus, it is suitable for phone-grade audio processing. The sampling type can be configured to either A-law or mu-law; however, samples are encoded in mu-law format by default (8 bits/sample). Through this audio port, a handset, a microphone, a headphone, or a headset can be attached for voice recording and playback. With appropriate software provided, the recording samples can be captured inside the host, and can then be used for further audio processing. Similarly, samples available inside the host can be readily played out to this port.

3.2 DEC Alpha's TURBOChannel DECaudio:

DECaudio is an optional module for the Alpha/DECstation which interfaces with the Alpha/Decstation system module through the TURBOChannel bus. The TURBOChannel bus is a high-performance, synchronous asymmetrical I/O channel that is used as an interface between the Alpha system and external modules. Via this bus, the host and the external module will have both read and write access (sharing memory for instance) to each other. DECaudio, besides having the same audio capabilities as the BaseBoard, also supports an interface between an Alpha and the telephone equipment via the DAA (Data Access Arrangement) module. Figure 2 shows the general configuration when DECaudio is interfaced with the PSTN (Public Switched Telephone Network).

Other interesting features of the DECaudio, according to [5], include:

- The ability to detect incoming rings from the telephone line and the states of the telephone (on-hook or off-hook). The host system will always have this information available at any instant of processing time via the TURBOChan-

nel bus. Besides, via the telephone interface, the host system can dial any phone when appropriate software is provided.

- A DSP chip for off-loading real-time, compute-intensive activities from the host. This feature enables the DECaudio to provide many useful audio processing applications including echo cancellation, and HiFi audio processing.
- An ISDN 'S' interface (basic rate interface) for future ISDN applications.

Unfortunately, we discovered that the hardware design of the DECaudio telephone interface is seriously flawed, limiting its usefulness. These problems and our attempted solutions are discussed in detail in Section 8.

4 Audio Software:

The audio software to support the Baseboard and DECaudio audio devices is called AudioFile. AudioFile is a portable, device-independent, network-transparent system for distributed audio applications. It was developed by the Digital Equipment Corporation Cambridge Research Lab. Similar to the X Window System, AudioFile allows multiple clients, supports a variety of underlying hardware, and permits transparent access through the network. Many platforms are supported with AudioFile system: Digital's RISC DECstations under ULTRIX, Digital's Alpha AXP systems under DEC OSF/1, Sun SPARC systems under SunOS, and Silicon Graphics Indigo workstations under IRIX. With AudioFile, many audio processing applications are available which include: audio recording, playback, audio/video teleconference, answering machines, voice mail, telephone control, speech recognition, and speech synthesis. A number of audio data types and sample rates are supported, from 8 KHz phone-grade quality to 48 Khz high-fidelity stereo. The following section will present more features of AudioFile, including its core client applications and its servers.

4.1 Protocol Description:

The current version of AudioFile supports TCP/IP and UNIX-main sockets. Thus, AudioFile gaurentees that data transport between the client and the server is reliable. Figure 3 [13] shows a time line of the typical scenario that takes place for a connection-oriented transfer at the socket level. Based on the same principles as the X Window System protocol, control and audio data are multiplexed over a single byte-stream connection between the client and the server. Over this connection,

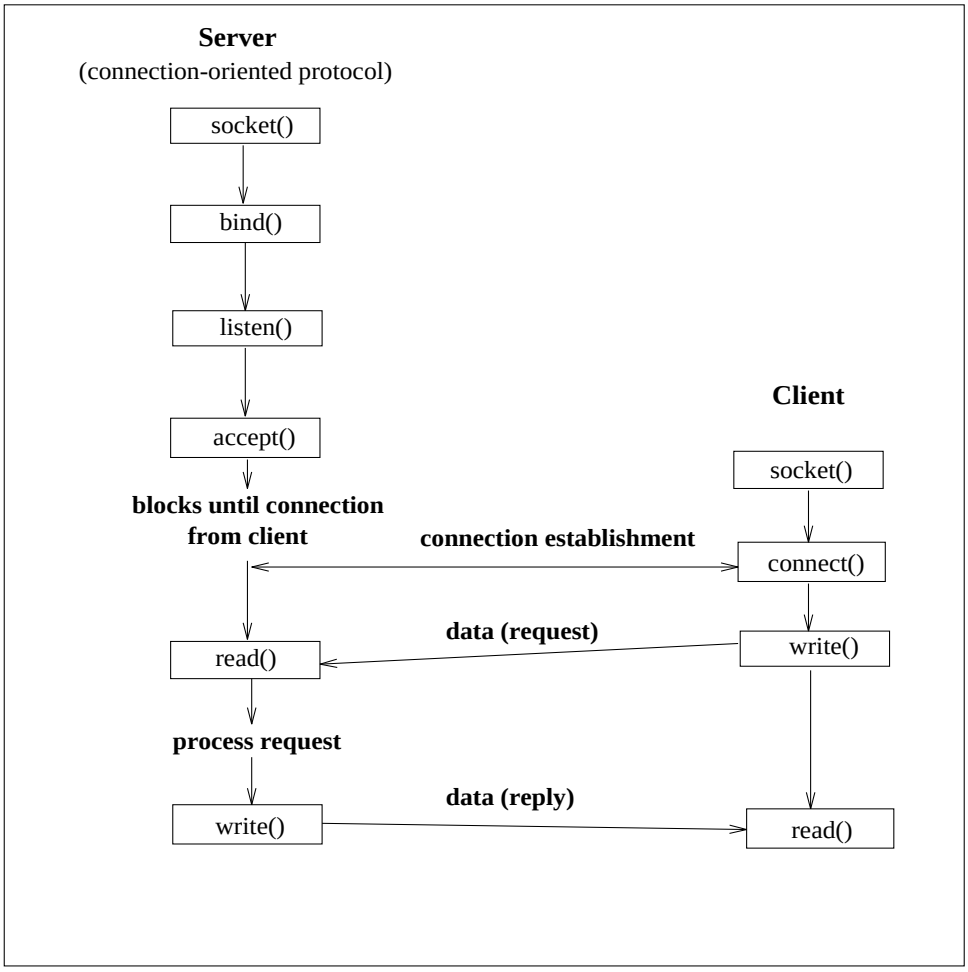

Figure 3: AudioFile Transport Protocol

more than one audio stream can be carried. With multiple clients running on multiple machines of different architectures, AudioFile allows these clients to share the same server at the same time.

At connection setup, the client and the server exchange version information besides the provision of the server authentication information from the clients, just as in the X Window System. Once a connection is established, clients communicate with the server through protocol requests for a particular service. All protocol requests have a length field of 16 bits (expressed in 32-bit units), an opcode of 1 byte, and an optional opcode extension. User audio data encapsulated inside the requests are kept naturally aligned (on a 32-bit boundary) inside the request header. The length field of requests limits the longest request to 262144 bytes, although in practice, AudioFile's longest request is substantially shorter. For instance, in the *apass* program, long play and record requests are segmented into 8K byte pieces.

In this way, AudioFile ensures that no single request will monopolize the server for a very long time.

AudioFile has a total of 37 requests, most of which are related to audio (play and record requests). The remaining requests are for access control and inter-client communications. A complete list of these requests can be found in [4].

4.2 AudioFile Abstraction:

4.2.1 Time:

Audio time is used in the protocol and at the client library API (Application Program Interface). It is also fundamental to the correct operation of the audio server where recording and playback operations are processed in association with audio time. In particular, recording and playback operations in the AudioFile system are tagged with time values that are directly associated with the relevant audio hardware. For instance, the DECaudio audio device has its own clock running, so when a playback is destined for DECaudio, the playback time associated with the request packet is with reference to DECaudio's clock. If there is any time difference between the server's clock the DECaudio clock, a time update will be carried out so that time consistency is guarenteed.

At any time, the server can directly access time information of the audio device to which it is connected. In fact, the server maintains a representation of the audio device's clock in a "time register". The server uses this device time information for scheduling events, such as playbacks and recordings, for a particular audio device. At server startup, the device time is initialized to 0 and advances thereafter. As implemented, the device time is represented by a 32-bit unsigned integer that increments once per sample period and wraps around on overflow. At a sampling rate of 8000 samples per second, the overflow period is equivalent to 3 days, thus posing no significant problems for real-time applications such as voice conversations.

4.2.2 Input and Output Models:

The input model and the output model for AudioFile are shown in figures 4 and 5 [4], respectively. The models represent the operations actually carried out within the AudioFile server. Both models buffer 4 seconds of sample data, and the sample data within a particular buffer are indexed by the current value of time. For the input model, clients requesting input data older than four seconds in the past are given silence by the server. Record requests that fall within the past four seconds

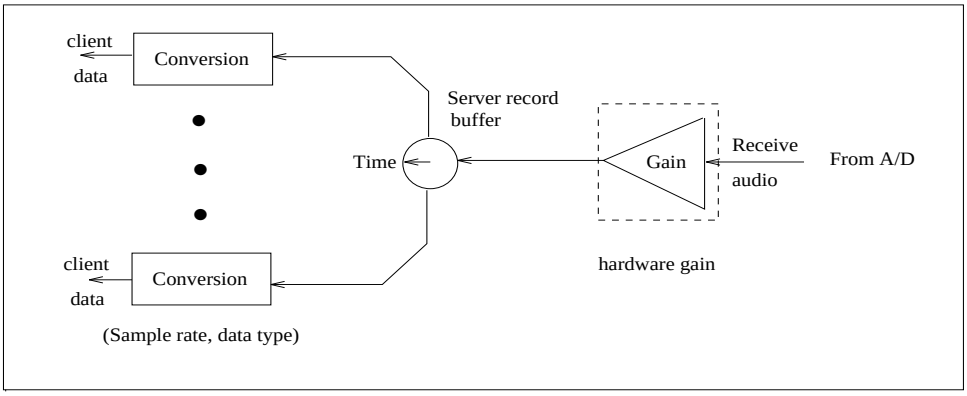

Figure 4: AudioFile Input Model.

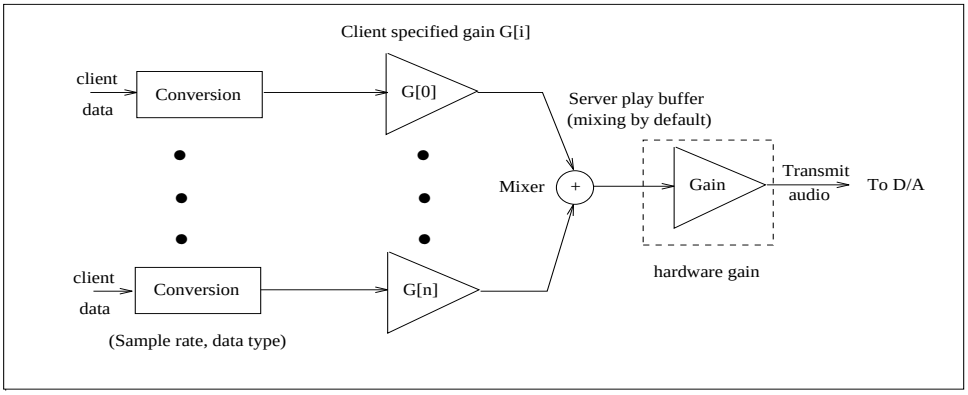

Figure 5: AudioFile Output Model.

return the buffered data. Record requests that fall into the near future will have the server not to return until the time advances far enough to service the request. For the output model, clients can schedule playback requests at any time from the present to four seconds into the future. Playback data that falls in the past is replaced by silence; whereas, playback data that falls into the near future will be buffered and guaranteed to be played out unless it falls beyond four seconds in the future. Also, it is the server responsibility for ensuring that the samples in the output buffer are sent to the D/A converter at their corresponding values of the time register via time indexing into the buffer.

4.3 AudioFile Clients:

AudioFile includes a number of core clients (refer to [4] for a complete list) for audio processing and phone processing. For the software development of this project, only *apass* and *aphone* clients were found appropriate for modification and expansion. The next section will present more detail of these two clients.

4.3.1 apass Client:

AudioFile's *apass* is used to transport audio data by copying audio data from one server (local) to another server (remote). Figure 6 shows the whole transporting process. As mentioned earlier, all playback/recording requests by the clients to the server are tagged with time values. The server will use this time information to schedule the requested event.

The basic operation is that *apass* will request recorded data from the local server which records audio data from an audio device attached to it. It then transports this recorded data to the remote server. The remote server will then playback the received data to a remote audio device attached to it. Obviously, apass will have to establish two connections before any transport of data is to be carried out: the duplex channel from apass to local server, and the duplex channel from apass to remote server. As mentioned earlier, these channels are TCP connections. As to be discussed later, one of the main modifications to the AudioFile system as part of the software development was to create two channels: the TCP channel and the connectionless channel (UDP for instance). The TCP channel is used for transporting control information (such as protocol requests); the connectionless channel is for transporting audio data. Incorporating the requirements of the ATM transport scheme, modifications to the AudioFile system were at low-level interfaces of AudioFile. The modifications will be presented in section 5.

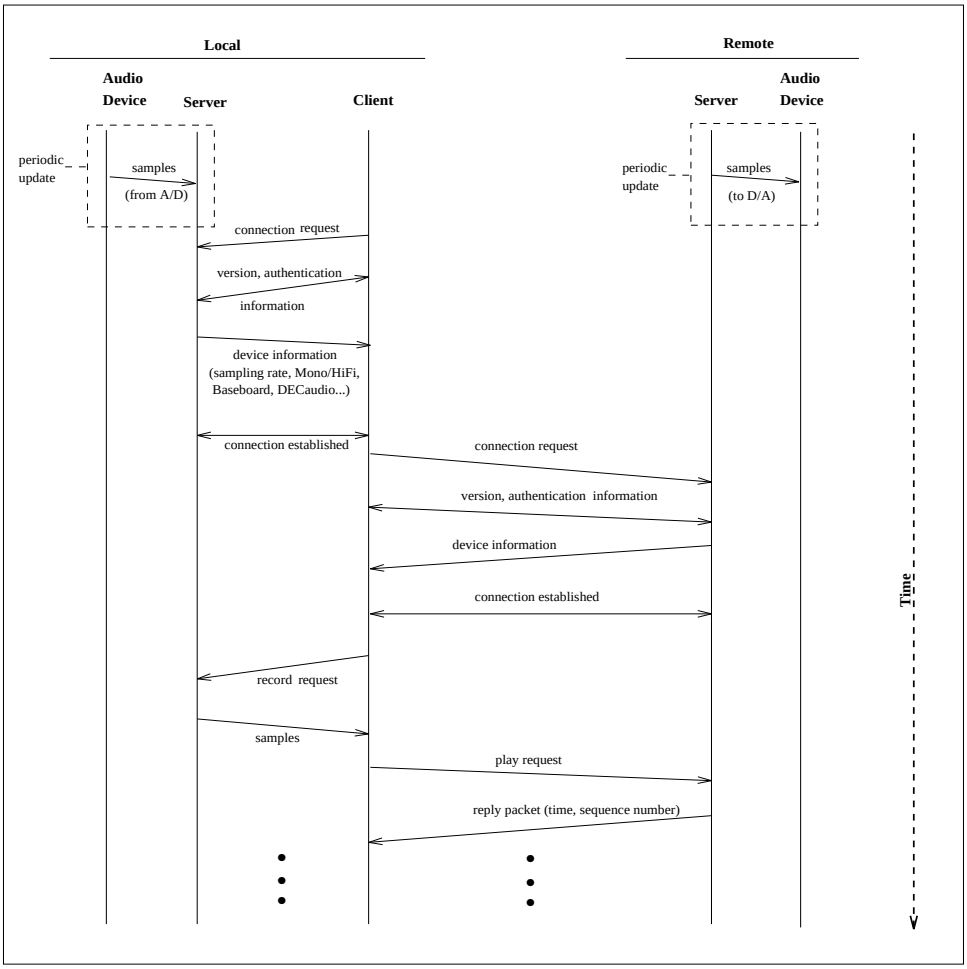

Figure 6: Typical communication between client apass and server (Aaxp/Alofi).

4.3.2 aphone client:

AudioFile's *aphone* client is used to provide phone dialing capability. Its capabilities are mainly used with DECaudio hardware to support phone dialing. The client enables the host to emulate the dialing process to a phone jack connected to the DECaudio audio device. The dialing process is carried out by the server and via the playing process. Before requesting the server, the client will convert each individual digit to the dual-tone-multiple-frequency (DTMF) format. Then each tone pair will be sent to the server as separate play requests. Upon receiving these play requests, the server treats them as no different from the regular play requests; thus, each sample of the tone pair will be considered as a play request. Since the DECaudio has internal dialing hardware, the tone pair received from the server will be played out as conventional tone dialing. Client *aphone* is actually not used in this project, but some of its operational principles were used in the software development.

4.4 AudioFile Servers - Aaxp and Alofi:

For this project, only the Aaxp server and the Alofi server were found appropriate for use and modifications. As mentioned earlier, the Aaxp server can run only under Alpha/OSF and is for the built-in baseboard; whereas, Alofi runs under both Alpha/OSF and DECstation/Mips, and is for DECaudio audio hardware. The two servers, even though they have different capabilities, share some common code of AudioFile. In particular, Alofi has all the capabilities that Aaxp server has; besides, it also has phone-control capabilities.

4.4.1 Server Implementation:

Like an X server, AudioFile is organized into three main components: the device independent audio (DIA), the device dependent audio (DDA), and the operating system (OS) component. Figure 7 shows the structure of AudioFile. The DIA section is responsible for managing client connections, dispatching client requests, sending replies and events to clients, and executing the main processing loop. As the name implies, the DIA section provides common code for both the Aaxp and the Alofi server. On the other hand, the DDA section, as the name implies, is used in according with a particular device. It presents the abstract interface for each supported device and contains all device-specific code. In particular, AudioFile has axp and lofi code under the DDA section for the Aaxp server and the Alofi server, respectively. Finally, the OS section consists of all the operating system-specific code for maintaining networking operations at the socket level. Much of the OS section and the DIA section is based on X11R4.

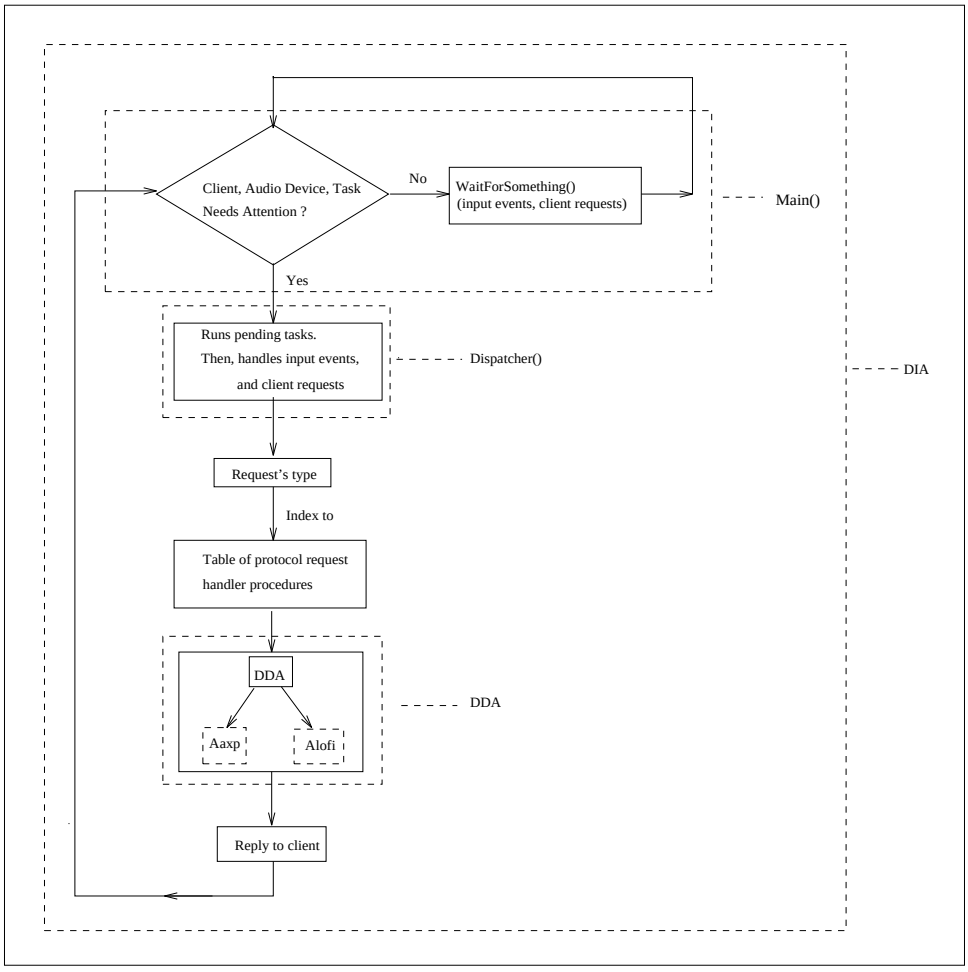

Figure 7: Server implementation.

The DIA and the DDA sections share the *AudioDeviceRec* structure. This structure encapsulates the information specific to an abstract audio device such as type of device (codec, DECaudio, HiFi...), properties of this device, and other device-specific information. This structure also contains the audio device time. This time information is the server's copy of the time register, and represents the server's view of current time. As will be presented in the next section, this time information will be periodically updated to reflect time consistency between the server and the audio device.

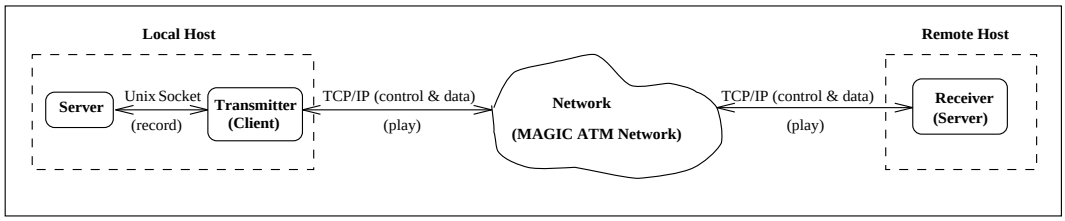

Figure 8: AudioFile with its original functionalities.

4.4.2 Server Update Process:

To meet real-time applications, sample data recorded from the audio device should be captured and moved periodically to the server buffer independent of any client request activity. This mechanism is an update task that keeps the hardware buffer consistent and up-to-date once every 32ms (250 samples). In this way, the hardware buffer always reflects the server's buffer at the time the hardware consumes an output sample. At each invocation, the update process first moves new record data from the hardware buffer to the server buffer, and then moves the next batch of playback data from the server buffer to the hardware buffer.

5 Software Modifications:

AudioFile, in the original version, supports TCP/IP and Unix sockets for its transport of data. Thus, AudioFile guarantees reliable and in-sequence transport of audio data across the network. As it turned out AudioFile had to be modified so that the demonstration system would match the proposed real system as closely as possible. In particular, AudioFile was modified to support the following capabilities:

- Cell loss under controlled conditions.
- Cell delay variation under controlled conditions.
- Independence between transmitter and receiver.

Figure 8 and Figure 9 show the original AudioFile and AudioFile after modifications, respectively.

The ability to introduce cell loss and cell delay variation under controlled conditions implied that AudioFile had to be modified to support unreliable transport of audio data. This required that AudioFile support packet mode besides its only

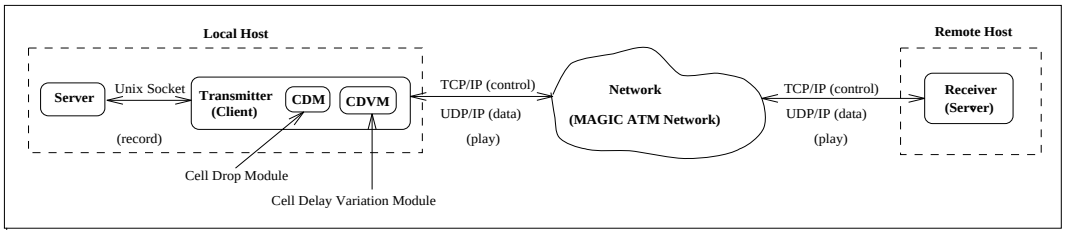

Figure 9: AudioFile with modifications.

stream mode (TCP/IP). UDP/IP protocol fits this requirement best since packet dropping does not require retransmissions. Thus, the software was modified so that TCP/IP carries control information whereas UDP/IP carries packetized audio data which can be dropped under the user's control. Even though UDP/IP is not the ultimate solution for the proposed system, it is a perfectly good transitional step toward the ATM-packet mode (AAL1) solution. In the ATM-packet mode, audio data is packetized directly into ATM-sized cells of 53 bytes. The ATM cell format currently supported by the ATM interface card (between a host and the ATM switch) is the AAL5 format. To model the proposed system even more accurately, the device driver for the interface card (otto board) could be modified in the future to add support for the AAL1 cell format. With AAL1, each ATM cell of 53 bytes of packetized audio data will be passed transparently through the interface hardware without any processing done on each cell by the interface. When AAL1 format is supported, we envision no major difficulty to go from UDP/IP to AAL1 mode since, as far as the kernel is concerned, AAL1 is just another protocol.

Another feature of AudioFile that needed to be modified is the dependency between the transmitter (client) and the receiver (server). In particular, in the original implementation of AudioFile, the transmitter and the receiver have a very strict time dependency . For each play request, the transmitter sends a request packet which contains the time (usually near future) the server should start playing the data along with the audio data itself. The transmitter then waits for a reply packet from the receiver to come back. Upon receiving the request packet, the receiver will extract the time field from the request packet and process the request only when the receiver's local lock (extracted from the local system host's clock) is equal to the time of the request. After processing the request, the receiver will then send back the reply packet to the transmitter. The reply packet contains the sequence number of the last request packet, the current time value of the receiver's local clock, the packet type (reply in this case), and the length of the reply packet. Thus the reply packet from the receiver to the transmitter is nothing but an acknowledgement to the transmitter from the receiver . Only when reply packet comes back from the receiver will the transmitter start another request. The transmitter will use the time value in the reply packet to adjust for any time difference between the transmitter

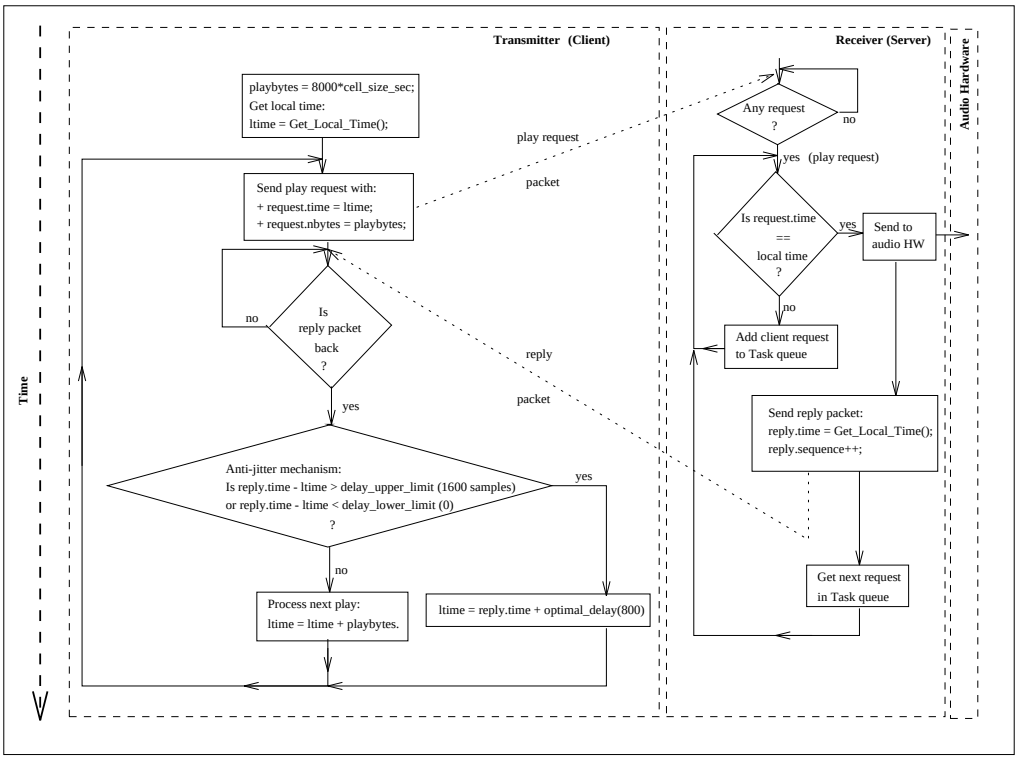

Figure 10: Time dependency between transmitter and receiver for a play request.

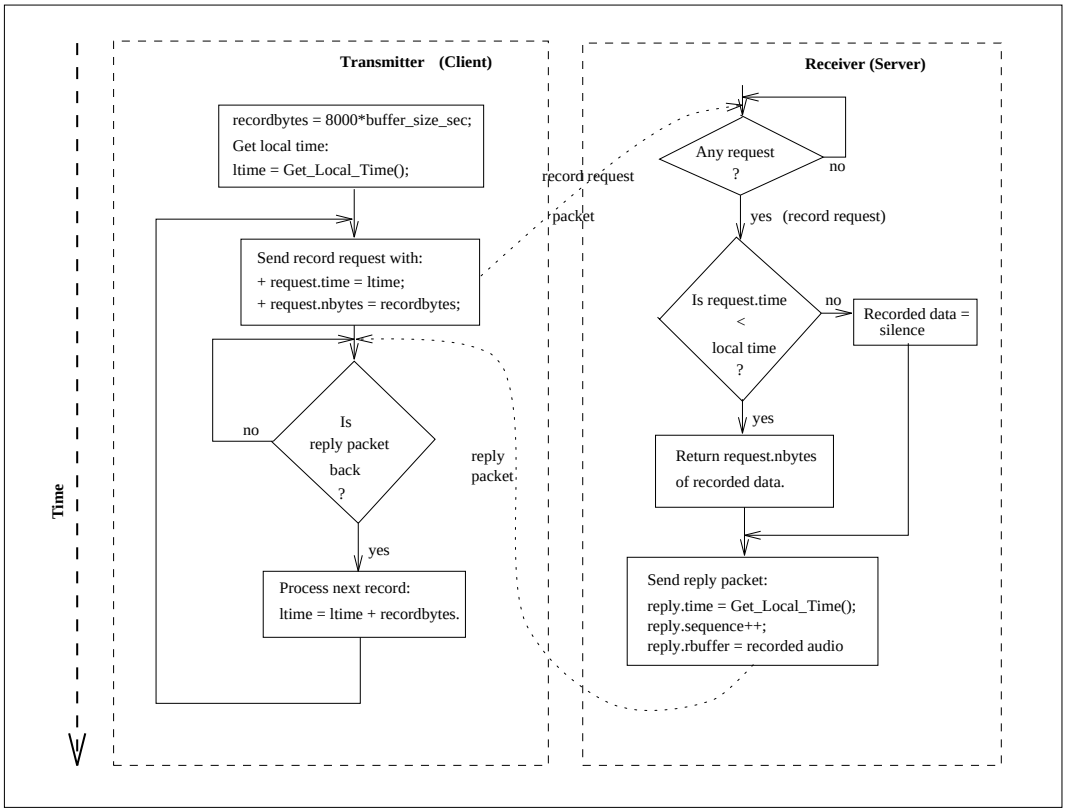

Figure 11: Time dependency between transmitter and receiver for a record request.

and the receiver (anti-jitter mechanism). If the receiver's clock seems to run faster than the transmitter's clock (the time extracted from the reply packet is larger than the transmitter's current time), beyond an unacceptable amount, 800 samples for instance, the transmitter will try to resynchronize by forcing the time difference to an optimal amount. Figure 10 shows the time dependency and the anti-jitter mechanism between the transmitter and the receiver for a play request.

Time dependency also exists in the original AudioFile when a record request is carried out. It is almost identical to the time dependency for a play request. When requesting that data be recorded, the transmitter (client) will send a request packet which contains an empty space for holding the recorded data. Unlike the play request, the receiver will not return anything (return silence) if the request time is in the future (the request time value is larger than the receiver's local clock time). Thus, only a request that falls in the latest four seconds of the receiver buffer will be successfully processed. In either case, the reply packet and the recorded data will be sent back to the transmitter. Unlike the play request, the transmitter does not carry out the anti-jitter mechanism to make sure that it does not request a record in the future, even though it acknowledges the problem (slip). Figure 11 shows the

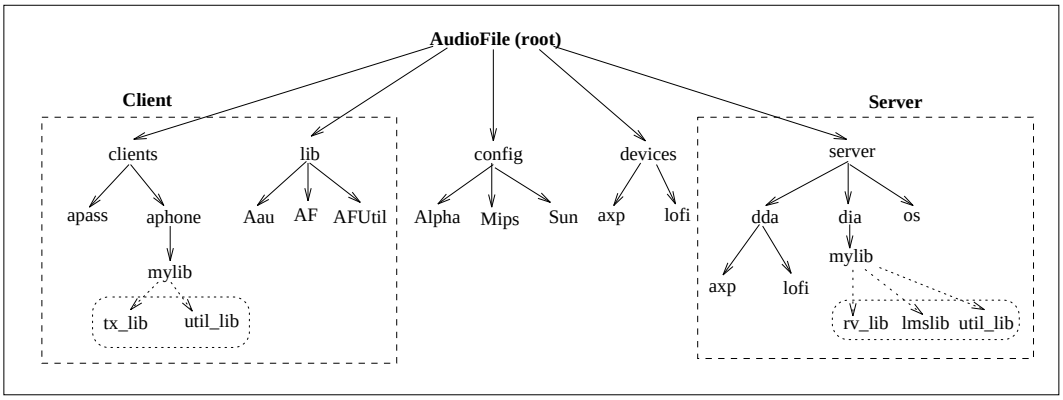

Figure 12: Overall structure of AudioFile.

time dependency between the transmitter and the receiver for a record request.

This time dependency between transmitter and receiver had to be removed to model the proposed system accurately. For the proposed system to be suitable in a real-time environment, the transmitter should not wait for the reply packet just for a reference of the receiver's local clock. Waiting for a reply packet is obviously an undesirable feature for a teleconference audio application when the transmitter and the receiver are far apart. In that case, significant delay would be introduced at both the transmitter and receiver since the reply packet would experience a significant propagation delay. The following section will provide more on the modifications introduced above.

It should be noted that all the modifications on top of AudioFile are put into separate libraries from AudioFile's libraries. Figure 12 shows the structure of AudioFile from the top level where subtrees correspond to subdirectories of AudioFile. The *config* and *devices* are the platform-dependent directory and device-dependent directory, respectively. There were no modifications done on these two libraries. The files in the dotted boxes indicate the developed libraries, namely, **tx_lib**, **rv_lib**, **util_lib**, and **lmslib**. Libraries **tx_lib** and **rv_lib** contain the transmitter's and receiver's added capabilities. Library **util_lib** contains error-handling functions. Finally, library **lmslib** contains procedures relating to the LMS algorithm which could be used for echo cancellation purposes.

5.1 Protocol Modifications:

The process of protocol modification turned out to be a rather difficult task. One of the main factors contributing to this difficulty was the fact that AudioFile is built completely on TCP/IP for its transport of data (control and audio data). Thus, to

modify AudioFile so it supports both TCP/IP and UDP/IP, one major question arose: where in AudioFile are modifications appropriate? In other words, at what level of interface of AudioFile would modifications be appropriate so that other functions of AudioFile which do not require changes would still work and be unaffected by the modifications? To be able to solve this complication, extensive research into AudioFile's source code and its technical report were required. It also required a full understanding of the structure of AudioFile, especially in the server section (receiver).

Before successful modification was achieved, an attempt was made to modify AudioFile almost completely so that it only supported UDP/IP. Even though we knew conceptually that this approach would not work since control information under UDP/IP would be lost unexpectedly, we still considered examining this approach to see how feasible it was and to learn more about AudioFile. It turned out to be much more difficult than initially planned since there were a lot of modifications required at different sections and levels of AudioFile. As this time, we decided to modify only certain parts of AudioFile. This approach turned out to be a successful one, and it has the following advantages: less modifications required, and thus less complexity; less levels of interfaces need to be modified; guarantees parts of AudiFile that are not modified to still operate as before. Thus, as a whole, this approach guareenteed a higher success probability than the approach above since only parts of AudiFile are modified. In the following discussion, more detail will be presented for this partial modification method.

To modify AudioFile so it supports both TCP/IP and UDP/IP, both the transmitter side and the receiver side were required to be modified.

With reference to Figure 12, client *apass* uses the functions of *lib* directory for transporting data. Particularly, directory *lib/AF* contains file Play.c which carries out the TCP/IP transport of audio data. Actually, procedure *AFPlaySamples* in Play.c is used to send requests and data. Its sequence of operations is shown in Figure 13.

The dashed box was modified such that sending the request packet is still carried out over the TCP/IP channel, but sending of audio data is now done on a newly created UDP/IP channel. For each TCP control packet, the transmitter sends 10 49-byte UDP packets (refer to later section on Software Development for detailed structure of the 49-byte UDP packet). *_ASend* procedure is no longer used and replaced by procedure *cell_delay*. This procedure will carry voice samples over a UDP/IP channel. It also has such functions as cell dropping emulation at a given bit error rate and cell delay variation emulation. Since the server, when receiving the play request packet, checks for the req->nbytes field to see how much audio data is to be played out, req->nbytes, has to be set to zero for every play request. In this way, the server will not wait once it gets the request packet; instead, it will

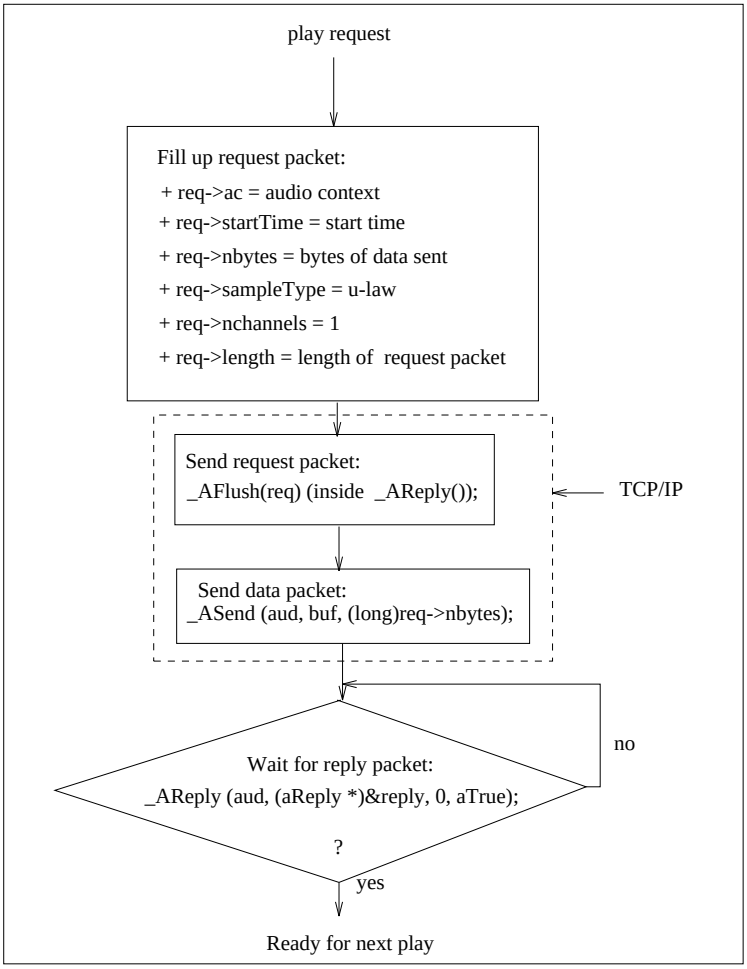

Figure 13: AFPlaySamples operation in the original AudioFile.

immediately switch to the UDP/IP channel and wait for the UDP data packets once it gets the request packet. With this implementation, there is one problem, however, which is the racing condition. Racing condition occurs when a TCP request packet and its UDP packet arrive at the receiver at different times. But when incorporating all the proposed system's error-handling cases [2] into the modifications, the racing condition is automatically solved thanks to the replacement algorithm. Thus, like the transmitter side, the receiver also has to be modified so it will read data packets from the UDP/IP channel. More detail of server modifications will be presented in section 5.4.

5.2 Removal Of Time Dependency Between Client & Server:

As mentioned before, the original AudioFile has a strict time relationship between transmitter and receiver. After sending a play request to the receiver (server), the transmitter will wait until the reply packet comes back. It was pointed out that this dependency needs to be removed in order to model the proposed system accurately. Furthermore, it is not needed since its main use is only to return the local time of the receiver and the sequence number of the last request. The time information returned from the receiver is used for an anti-jitter delay mechanism while the sequence number is used for acknowledgement purposes. Removing this dependency will consequently improve overhead efficiency since there is no need to have a reply packet for every request.

So, if there is a way to let the transmitter know that its last request was always guaranteed a success, it should be possible to completely omit the reply packet. One way of doing this is to fake the return of a reply packet. This process is a 3-step process and requires three new procedures: _fakeASend, _fakeAReply, and cell_delay. Basically, the first two procedures replace the _ASend (thus _AFlush), and _AReply procedures for the purpose of sending a reply packet and for fake acknowledgement of the reply packet. The cell_delay procedure is the developed procedure to transport packetized audio data over the UDP/IP channel. The modified AFPlaySamples is shown in Figure 14.

It should be noted that the field req->nbytes of the request packet is now set to zero. Setting it to zero will implicitly signal the receiver not to wait for the audio data on the TCP/IP channel, as discussed earlier. The _fakeAReply basically just acts as if a reply packet is actually received by carrying out all the operations originally done at the receiver. These operations include: keeping track of the type of the packet coming back (reply packet), keeping track of sequence number of the current request at the transmitter and of sequence number of the next expected request packet at the receiver. The two sequence numbers have to be the same for

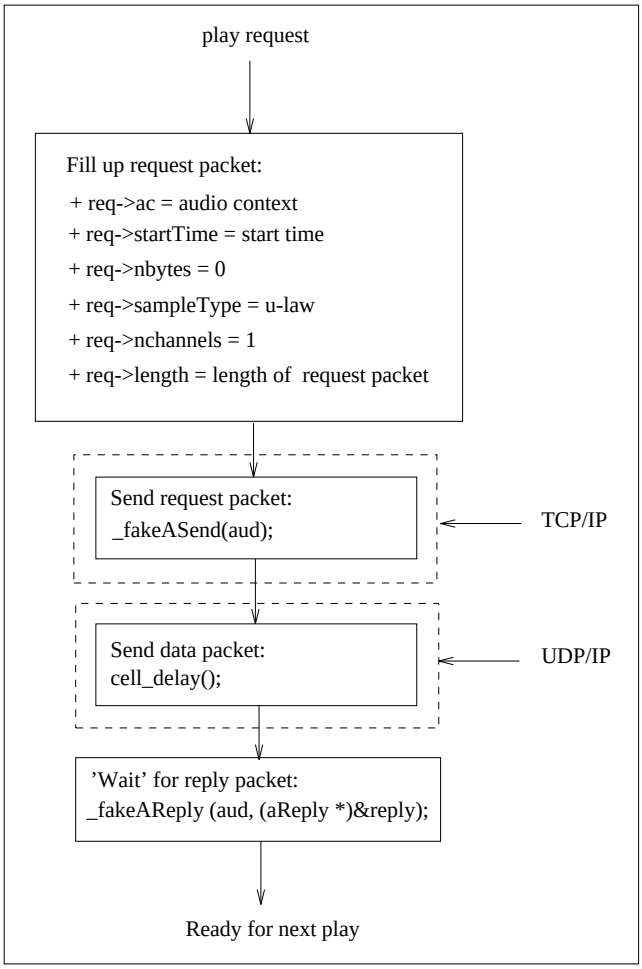

Figure 14: AFPlaySamples with modifications.

each invocation of AFPlaySamples; otherwise, the transmitter will recognize there is sequence loss (loss of reply packet) and will try to resynchronize by adjusting the sequence numbers. In reality, the loss sequence problem seems to be very rare as experienced during the course of using AudioFile. In fact, it has never occured.

Another time dependency feature of AudioFile, the specification of playout time, was also removed. The modifications effectively enable the transmitter not to worry about the playout time for each play packet. Besides, the transmitter no longer needs to carry out the anti-jitter mechanism as done before. Instead, the scheduling of playout time is now the responsibility of the receiver. Section 5.4.2 will present detail of modifying the DDA section of AudioFile to accommodate this capability.

5.3 Client Modifications:

Besides the protocol and dependency modifications to AudioFile's transmitter (client *apass*), additional modifications to accommodate the capabilities shown in Figure 9 include a recorder/dispatcher model. Basically, the recorder/dispatcher model is embedded into the transmitter. Its capabilities include capturing the audio data, sending captured data to a pipe (FIFO), forming ATM-sized (49 bytes for now) packets, introducing packet loss and delay variations to cells transmitted, and finally, dispatching (transmitting) the packets over the network. The detail of the recorder/dispatcher model will be provided in section 6.1.

5.4 Server Modifications:

5.4.1 DIA modifications:

As presented earlier, the DIA section of AudioFile is responsible for managing client connections, dispatching client requests, sending replies and events to clients, and executing the main processing loop. When a request arrives, the dispatcher looks at the request packet and determines the requested service via a lookup to the service table. The result of this lookup is the handler name for that service. The dispatcher then calls that procedure to process the request. Figure 15 shows the operations of the dispatcher, which match closely to the behavior of the *Dispatch()* procedure of AudioFile.

Note the DDA section of the figure. Depending on the request's specification of audio device (each request packet carries the particular device information), the server will call a device-dependent handler.

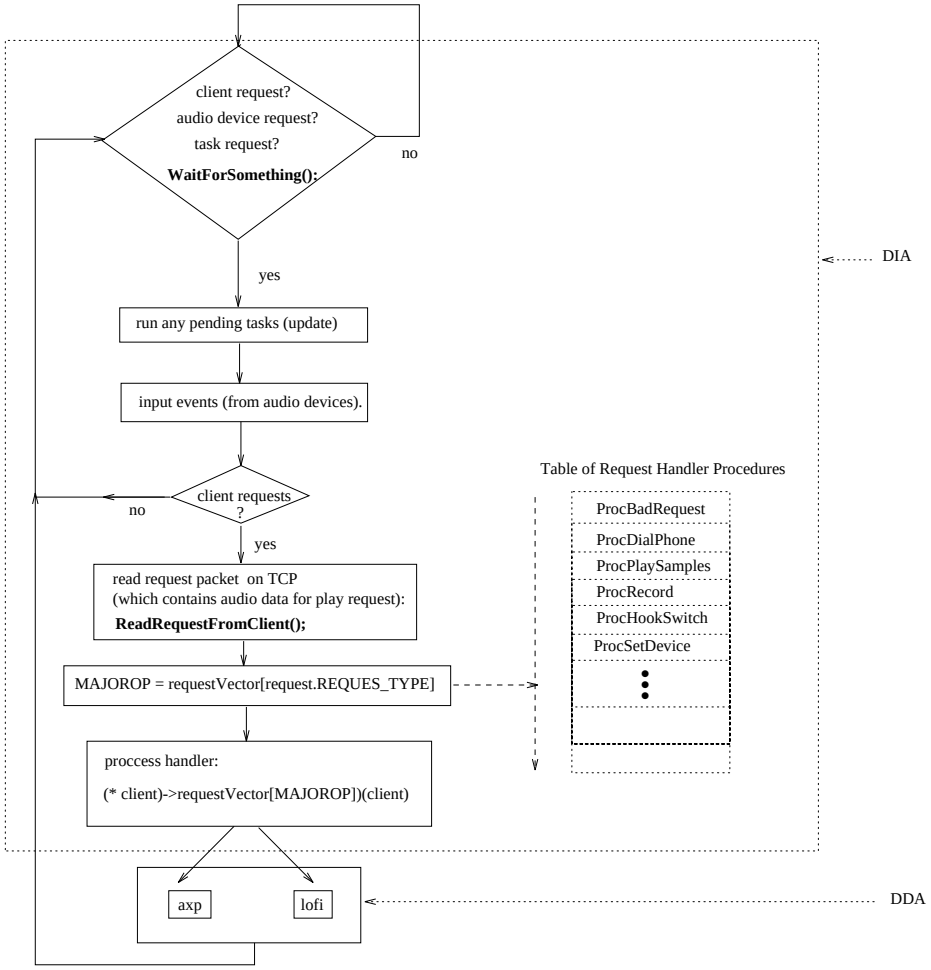

Figure 15: Operations of original AudioFile's dispatcher.

To add the UDP/IP functionalities to AudioFile, modifications must be done just after reading the request packet and before processing the handler procedure. So after reading the request packet with the *ReadRequestFromClient* procedure, the request type is known. Also, since the *request->nbytes* field of the request packet is set to zero, as noted in section 4.1 above, the dispatcher so far has not received any transmitter's audio datat. Thus, inserting the UDP/IP procedures for reading audio data from the UDP port will guarantee that the dispatcher will also get data, besides the request type. Note the advantage of the modification approach taken: other operations of AudioFile such as processing events and tasks are still unaffected. Thus, the modifications to the DIA section are completely transparent to AudioFile. Taking advantage of this transparency, new functionalities were added for user-specific needs without affecting its basic operations. Indeed, the insertion point chosen is perfect for providing expansion room for the software development part of the project. Software development for the proposed solutions is presented in detail in section 6.

5.4.2 DDA Modifications:

Modifications to DDA are required to remove the time dependency between the transmitter (client) and the receiver (server). The two DDA's procedures that were modified are *codecPlay* and *codecRecord*. The procedure *codecPlay* is the AudioFile-device interface for playing audio samples to the audio device; whereas, the codecRecord is the AudioFile-audio device interface for recording audio samples from the audio device. Even though DDA has axp and lofi as separate sections, the two sets of source code share a lot of common features. In fact, since AudioFile's lofi section was implemented before the axp section, lofi's functionalities, besides having those capabilities to support the DECaudio device, contain all of the axp section's functionalities. The *codecPlay* and the *codecRecord* of the two sections are very much the same.

Modifications to codecPlay:

Before going into detail, it would be helpful to understand how received samples are buffered before they are handed to the hardware buffer. Figure 16 shows the data sample flow between the server, *codecPlay*, and the hardware buffer for a play request.

With a reference to the figure, after samples are received by the server, they are stored into the server's buffer. The location of the first samples (out of nbytes) will be determined by start_time which was extracted from the request's packet. Subsequent samples are stored in consecutive locations, one after another. The server next calls the protocol handler, in this case *codecPlay*, to process the

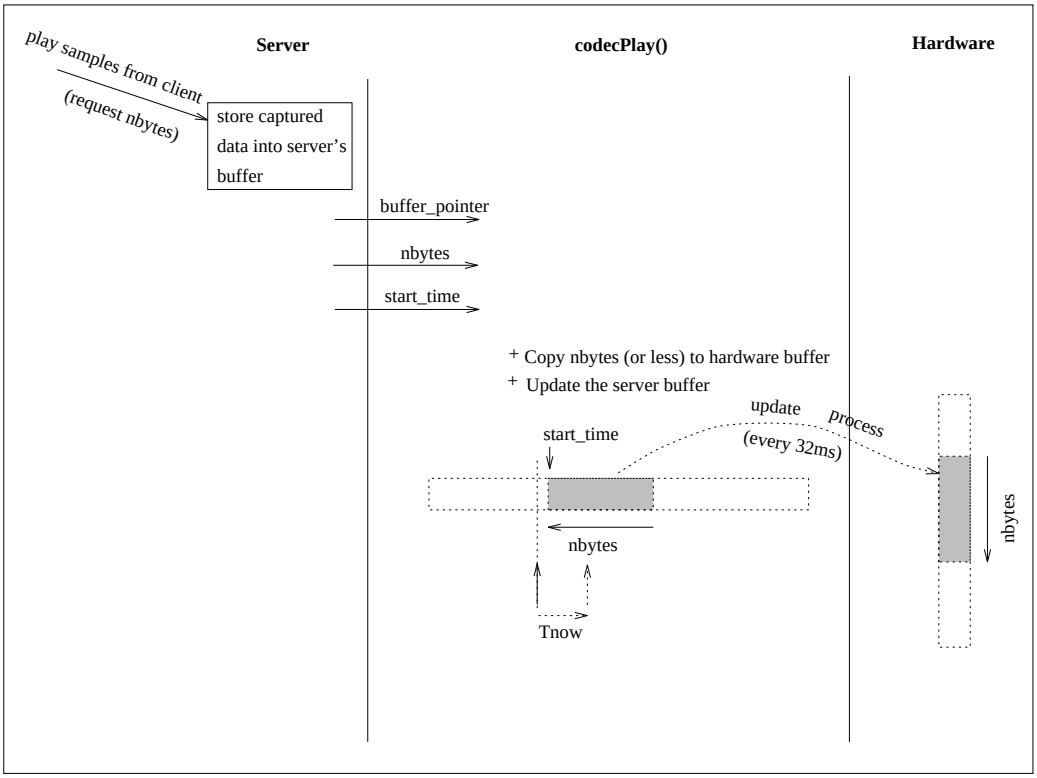

Figure 16: Data sample flow between server and hardware buffer.

requested service (playback). Passed to the *codecPlay* handler are the pointer pointing to the server buffer, the start time, and the number of bytes requested for playback. The *codecPlay* then does the actual copying of samples from the server buffer to the hardware buffer. Note, however, that not all of the requested nbytes will be moved to the hardware buffer. The loss of data is due to the time adjustment mechanism implemented in AudioFile's server. The mechanism ensures that only samples scheduled (requested) in the near future will be played out. The near future must be beyond update time, and within the server's buffer. Those falling in the past, in which case the requested start time is lagging behind the current server's time, will be silently discarded. This mechanism was already illustrated and discussed in Input and Ouput Models Section. As mentioned before, the update process is independent of the server's operation and is periodically carried out (every 32ms, or 250 mu-law samples). It is used to ensure that the server gets the latest samples from the A/D, and that the server passes playback samples to the D/A if there are requested play samples from the transmitter.

The operations of the original *codecPlay* handler are shown in Figure 17. With reference to the figure, play requests' samples (delta) that fall in the past will be silently discarded, as done by the dotted box. The time index, play_time, which corresponds to the first sample's location in the server's buffer, is also advanced by delta amount.

Modifications to the *codecPlay* handler involve establishing a time reference inside the handler and are only within the dotted box. This time index is memorized by declaring it as static variable inside the handler. The time index will be advanced for every invocation of codecPlay by an amount of nbytes of samples or more, depending on whether the time index is falling behind the device's current time. With this local time reference, even though play request packets sent from transmitter do not contain time information, the receiver always knows where to store the samples in the server's buffer. However, unlike before, the modified version of the handler does not discard data; thus, as a whole, packet loss is only caused by the transmitter or by the network (very rare). It should be noted here that the new time index kept by the receiver does nothing but to ensure received samples are always buffered and will be played out. Thus, a no-loss-of-data condition is guaranteed at the receiver, to meet the proposed solutions specifications. The modfications also ensure the first sample will be delayed by a variable blind delay amount. This is done by adding the blind delay amount to the play-out time of the first sample.

Modifications to codecRecord:

The modifications to the *codecRecord* handler are similar to those of *codecPlay* handler. The time dependency is removed inside the *codecRecord* handler by establishing a local time reference just as in the *codecPlay* handler. This time index

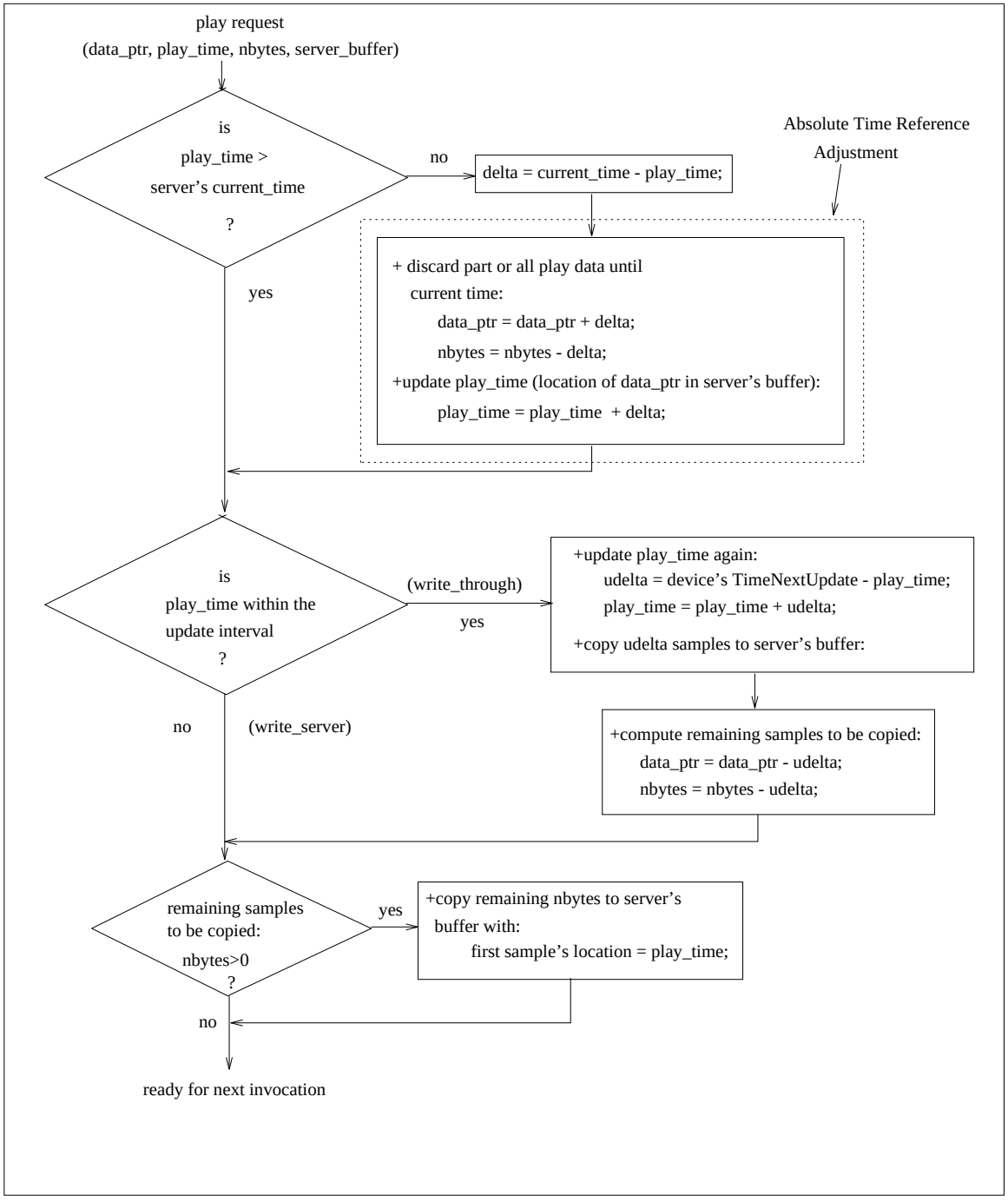

Figure 17: Operations of codecPlay handler in the original AudioFile.

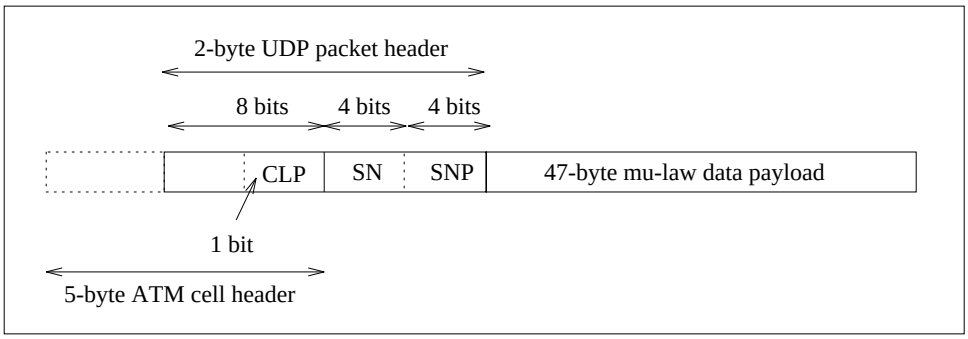

Figure 18: Format of the ATM-emulated UDP packet used.

is advanced for every invocation by an amount of nbytes of samples as requested from a client (transmitter). This results in a continuous recording of samples from the A/D, for recording update is done periodically as discussed before. With the original *codecRecord* handler, continuous recording is not supported for it causes some of the recorded samples to be lost (silence replacement) if the transmitter's request falls in the past (record slip). With the modified handler, the server knows where to continue taking the next chunk of requested samples from the server's record buffer, without needing the transmitter's request time information. Because of the continuous advancement of the time index, continuous samples from the server's record buffer will be returned to the requester, without any data loss.

6 Software Developments for DS0 MSB/LSB Scheme:

6.1 Phase I: Emulation of ATM cells using UDP packets:

In this phase of development, each packet transmitted over the ATM network is a UDP packet of 49 bytes (excluding the UDP overhead), which includes the a 2-byte header for sequencing and loss priority purpose, and the 47-byte payload of mu-law data. Figure 18 shows the format of the UDP packet, emulating the ATM cell format. As shown, not all bytes of the ATM cell header are used; only the CLP (Cell Loss Priority) is needed. The CLP bit information is used at the transmitter to indicate the MSB or the LSB payload; whereas, at the receiver, it will help the receiver to recognize the type of the packet (MSB/LSB) received. It is significant information because it directly affects the way the receiver carries out the proposed replacement algorithm. The HEC (Header Error Check) field of the ATM cell is not used here because it is assumed that the transmission facility has a very low bit error rate.

The SN (Sequence Number) and SNP (Sequence Number Protect) fields emulate AAL-1 header information. They are embedded in the payload portion of the UDP packet. According to the proposed solutions, the SNP field is not yet used for it is assumed that there is no error in the SN field. Again, this is a reasonable assumption due to fiber optic transmission in the ATM backbone. The SN field will be assigned at the transmitter, having values from 0 to 7. MSB packets always have even SN; whereas, LSB always have odd SN. Thus, a check is always made to ensure a MSB packet always having an even SN and a LSB packet always having an odd SN, before it is sent out to the network .

From the user's point of view, phase I presents a transparent emulation of transmission of ATM cells. This emulation includes:

- Forming MSB and LSB buffers from the mu-law audio stream.
- From each buffer, forming MSB and LSB payload portions of an ATM cell (47 bytes).
- For each type of payload, appending a 2-byte header and a 1-byte SN/SNP with the CLP bit and the SN field being assigned by the transmitter.

At the receiver, the reverse process in which MSB and LSB cells, after their headers are stripped off and examined, are recombined to form back the mu-law samples. The reconstructed samples are then fed to the D/A for playback.

6.1.1 Transmitter:

Figure 19 shows the implementation of the transmitter. Even though there are software limitations, the transmitter is modeled as closely to the model proposed in [2] as possible. As shown, the transmitter actually consists of two independent modules: the Recorder and the Dispatcher. They are independent in the sense that one does not have to wait for the other for a particular processing operation. In particular, the Recorder performs continous recording, with 470 mu-law samples (10 47-byte udp cells) for each request to its local server. Each recording request is an atomic process because the server will not return the samples until they are all recorded. The Recorder then places the samples on the FIFO which can be readily read by the Dispatcher. Upon placing the data on the queue, the Recorder immediately starts another recording. In reality, however, there will be some small finite delays between recording requests because the server has to process other tasks, such as play requests, update tasks. However, because of the periodic update operations of the server, data loss due to gaps between recording requests is very

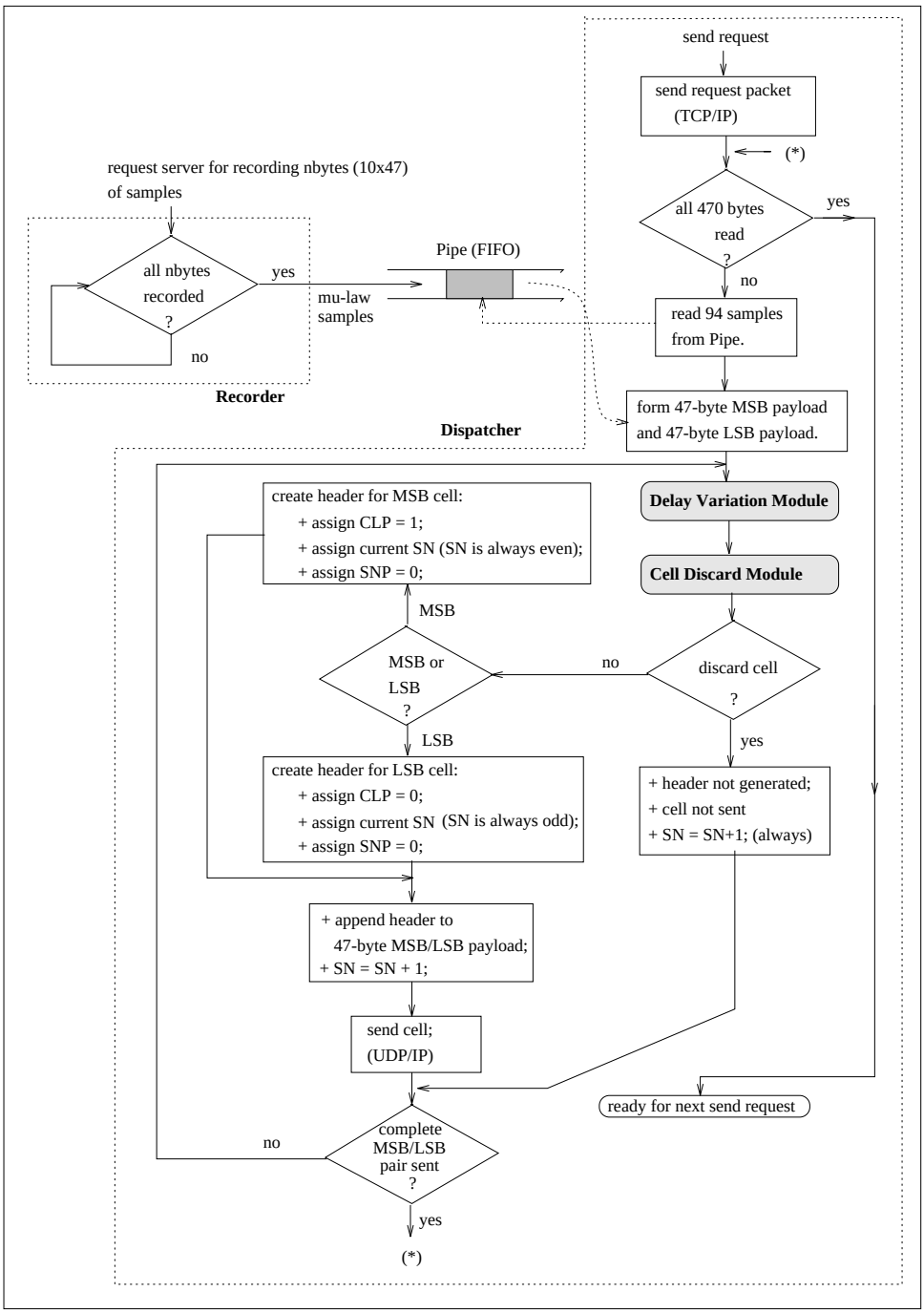

Figure 19: Transmitter's implementation.

small and not audible. To implement the independence between the Recorder and the Dispatcher, Unix process forking and pipe operations were used. Thus, in effect, the Recorder and the Dispatcher are completely separate processes, sharing a common data pipe.

The Dispatcher first checks the data pipe to see if there is data to be sent; only when there is data, the Dispatcher will read the data from the pipe, packetize the data, and then transmit the packets. With reference to the figure, upon receiving a send request (play request from its main loop), the Dispatcher then sends the request packet over TCP/IP channel. Next, it will read 94 samples (2 47-byte udp packets) from the pipe previously placed by the Recorder. The 94 samples obtained are then fed into the packetizer module (_ATMCellPacketizer) to produce two classes of payload: the MSB payload and the LSB payload. Following this operation, the MSB payload is processed first, then the LSB payload; that is, pairs of MSB and LSB cells are always sent if there is no cell loss.

For each payload type, an amount of cell delay variation is calculated in micro-second resolution (x micro-seconds). A uniform distribution is used where the lower and the upper bounds are user-assigned and can be changed even during a running process. This enables the user to easily carry out experiments and observe voice quality under different delay variation conditions. To introduce randomness, the *drop_OR_ndrop* procedure uses the *random()*, and *gettimeofday()* functions. The *gettimeofday()* was used to get the seed number to be used in the *random()* function. Obviously, the seed number will be very random.

To delay an amount of real-time x micro-seconds, the *select()* system call is used. With *select()*, timing is reasonably accurate because it interacts directly with the kernel. After x micro-seconds of delay, the Dispatcher then continues operating by considering whether or not to drop the current payload type with the cell loss probability given. Like the delay variation parameter, the cell loss probability parameters (MSB and LSB cell loss probabilities) are user assigned and can be altered during a running session of the program without having to actually kill and restart the running process. If the payload is declared to be dropped, by the *drop_OR_ndrop()* procedure, there is no more processing required. In this case, no header is generated, the payload is dropped, and the SN (sequence number) variable is incremented by 1. By always incrementing the SN by 1 even if there is no actual transmission of a cell, the SN of the next cell to be transmitted (or dropped) will be ensured to be correctly assigned. As mentioned above, because cells are transmitted in pairs, i.e., a MSB cell followed by a LSB cell, an error in SN assignment will be dectected if the cell to be transmitted is a MSB cell and the current value of SN is odd, or if the cell is a LSB cell and the current value of SN is even. The implementation of the transmitter prevents this incorrect assignment from happening.

On the other hand, when the payload is marked as a no-dropped payload, its 2-byte header will be generated and appended to the payload. If the payload is of MSB class, its header's CLP bit is set to 0, and to 1 for the LSB-class payload. The header's SN field is then assigned the SN's current value. The whole cell is then sent over the network over the UDP/IP channel. Regardless of the class of the payload to be transmitted, the SN variable is always incremented by 1 once a cell is sent. A check then is made to see if a pair of MSB and LSB cells has been completely sent. If it is, another pipe read is initiated to get the next 94 samples. When all 470 samples are sent, the Dispatcher is released and ready for the next send request from the main module.

6.1.2 Receiver:

Figure 20, 21 and 22 show the complete diagram of the receiver. Again, the receiver, like the transmitter, is implemented such that its operations are closely matched to the operations of the proposed receiver model [2].

With a reference to Figure 20, after receiving the request packet, the receiver will identify the the type of the requested operation, via the request packet's request type, *requestBuffer->requestType*. If the request type is other than 7 (play request), the receiver services the request through its normal protocol handlers, bypassing the developed procedures. When the request is a play request, the request packet's *nbytes* field is checked. If the field's value is zero, the play request is for playing UDP packets (MSB/LSB cells); otherwise, the play request is for playing dialing samples (DTMF samples). As far as the receiver is concerned, the two play requests are distinguishable only by the value of the *nbytes* field. It should be noted that the dialing samples are sent over the TCP/IP channel as part of the request packet. These samples represent dialing digits in the DTMF (Dual Tone Multiple Frequency) format and thus need to be reliably protected.

Once regconizing that the play request is for playing UDP samples (*nbytes* field is zero), the receiver then starts anticipating reading the 10 (5 MSB/LSB pairs) UDP cells. Depending on the conditions of the arriving cells, the receiver will perform case-specific operations. There are 10 known cases (round-edged boxes in the three figures) to the receiver. Basically, these cases are actually those of the proposed solutions [2]; thus, their operations need no further explanations.

6.1.3 Telephone Access Capability:

In this section, telephone access capability, via AudioFile, is presented. As presented earlier sections, phone dialing is supported via the DECaudio audio hardware. The DECaudio has a host-to-jack interface to support software dialing from host to any phone via the phone jack connected. As supplied with the original

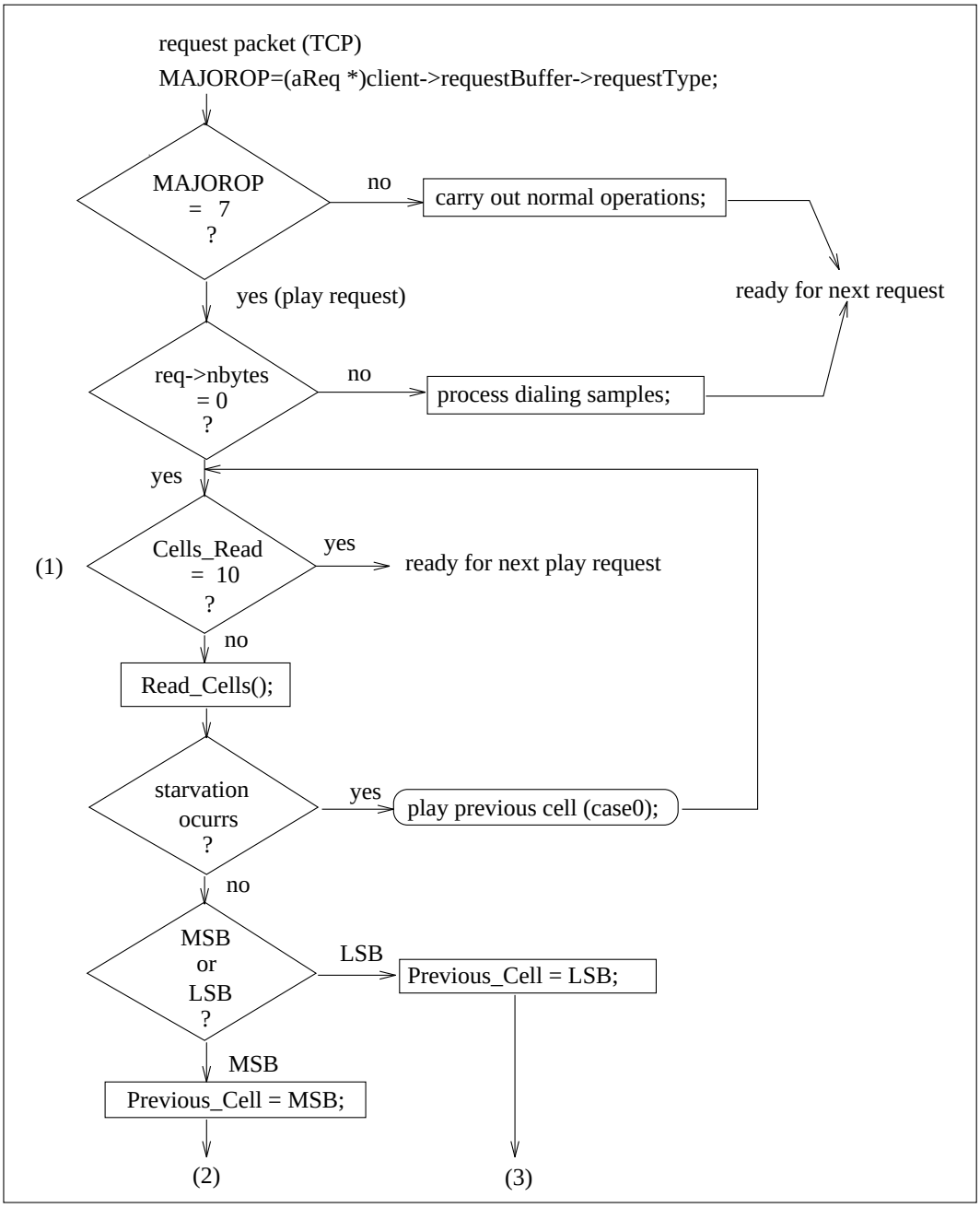

Figure 20: Receiver's implementation, part 1.

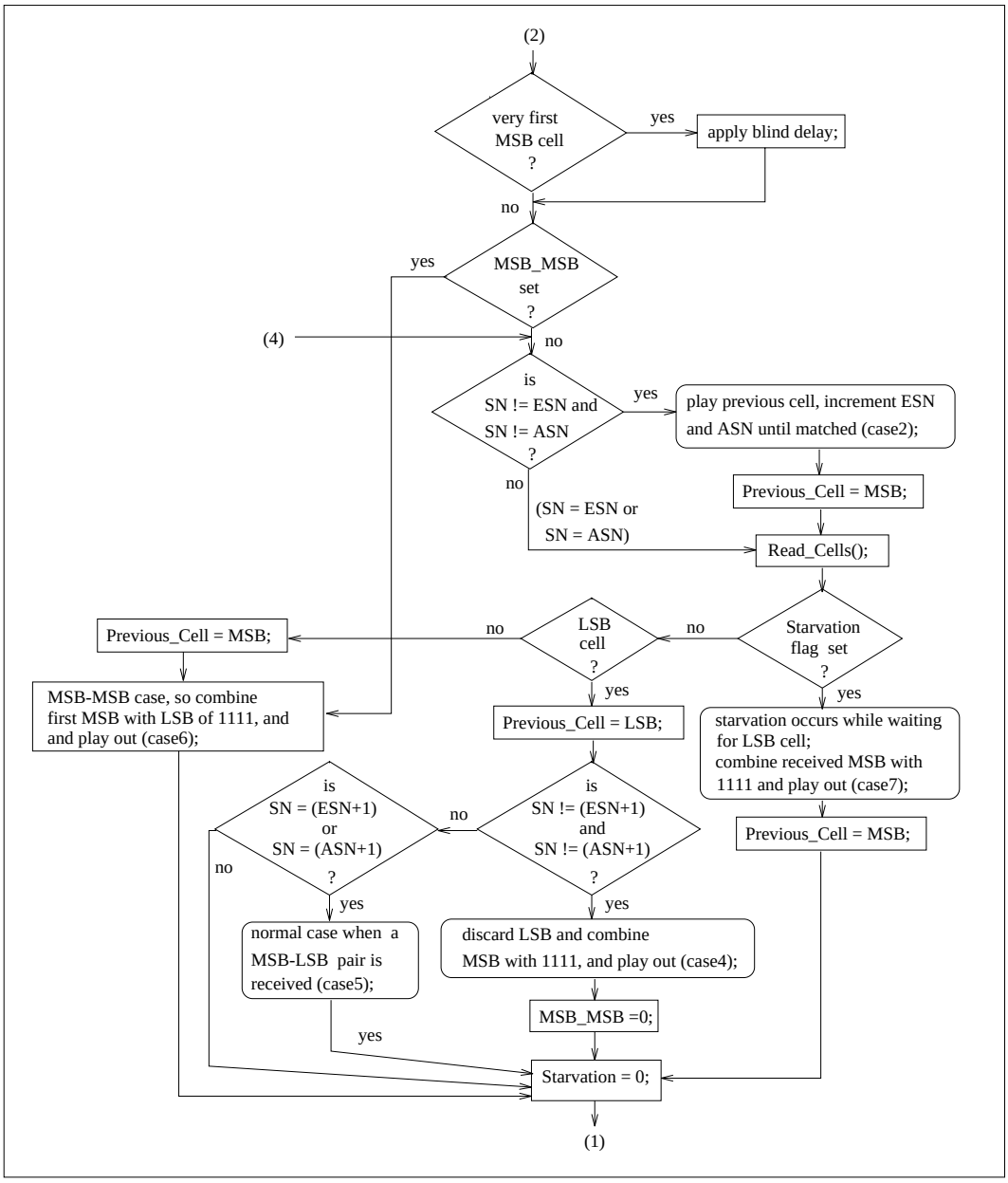

Figure 21: Receiver's implementation, part 2.

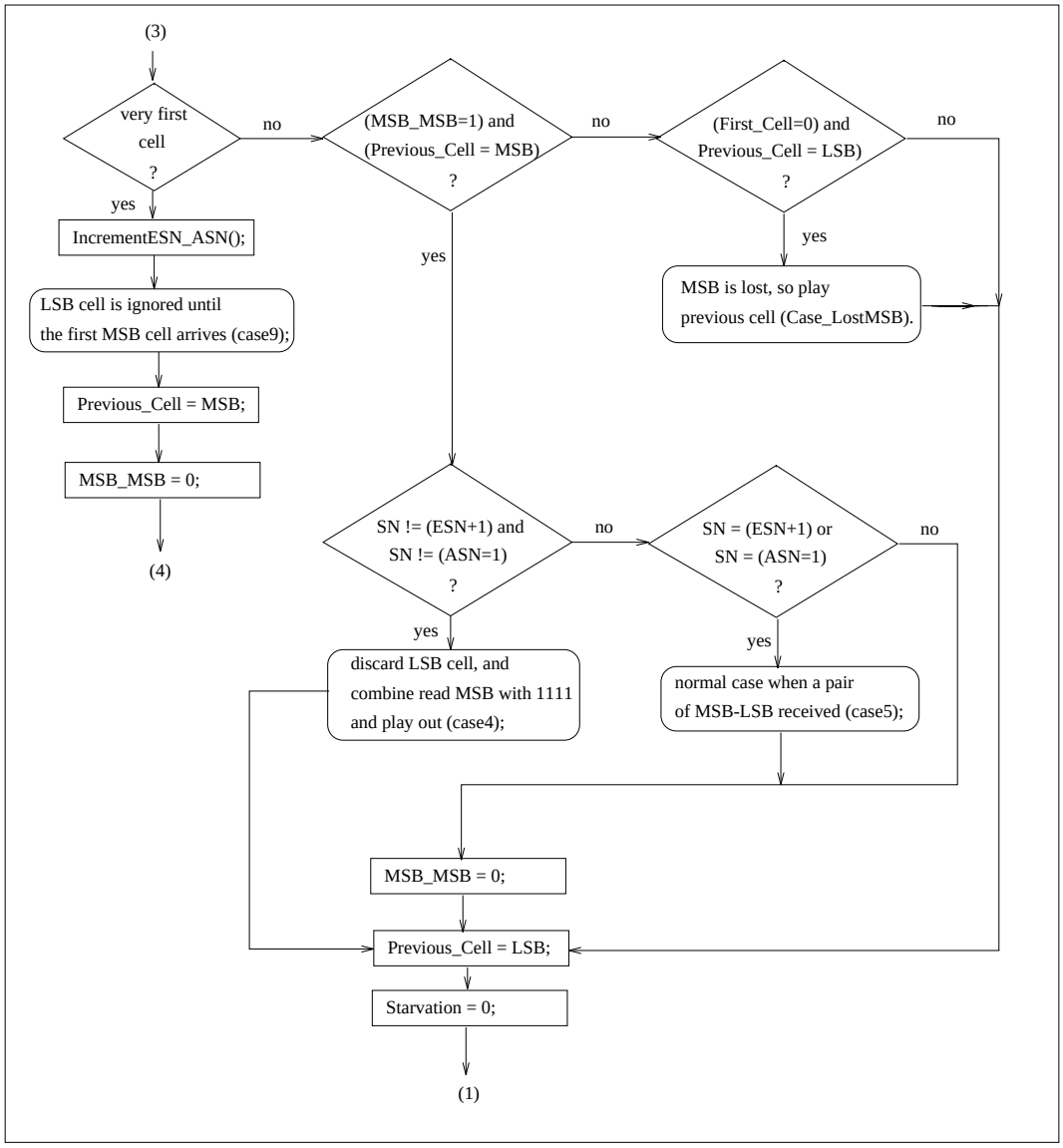

Figure 22: Receiver's implementation, part 3.

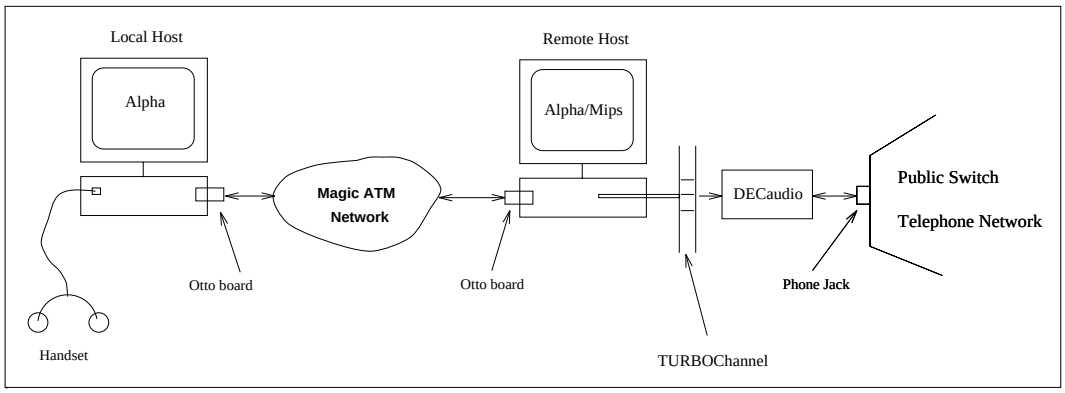

Figure 23: Modified configuration for voice over ATM network.

AudioFile, phone dialing is supported, but is a stand-alone operation and is limited. It is limited in a sense that it only supports "local" dialing, i.e., the phone jack has to be connected to the host which performs the software dialing.

To suit the proposed model, "remote" dialing capability had to be implemented. With "remote" dialing, the dialing host does not need to be physically connected with the phone jack; instead, it can be any host which has internet access to the host which is physically connected with the phone jack. Figure 23 shows the configuration. It should be noted that this configuration is different from the proposed configuration shown in Figure 1. With the new configuration, not only is less interface hardware needed, but also remote dialing is supported. The feature of "remote" dialing presents to the user a real convenience when he or she carries out experiments to test voice over the ATM. In fact, as the phone dialing and the UDP transport are incorporated, users can just sit at any Alpha host, "remote" dial a phone, and then carry the conversation over the ATM network, via the handset. During the conversation, cell loss and cell delay variation can be altered to vary the voice quality.

To combine the phone dialing capability and the UDP transport feature, *rdial* is implemented. Its operation is shown in Figure 24. Some of the application's principles are borrowed from the *aphone* client application of AudioFile. But it is a newly developed application in the sense that it incorporates the phone dialing operation with all the developed procedures. As a result, the application presents to the user a real-time tool for conveniently testing voice over ATM network. With reference to the figure, it should be noted that the transport of dialing samples (dialing digits) is over the TCP/IP channel, using the *AFPlaySamples* procedure of the original AudioFile. The samples are first converted to the DTMF format before they are sent over the remote host, inside *AFPlaySamples*.

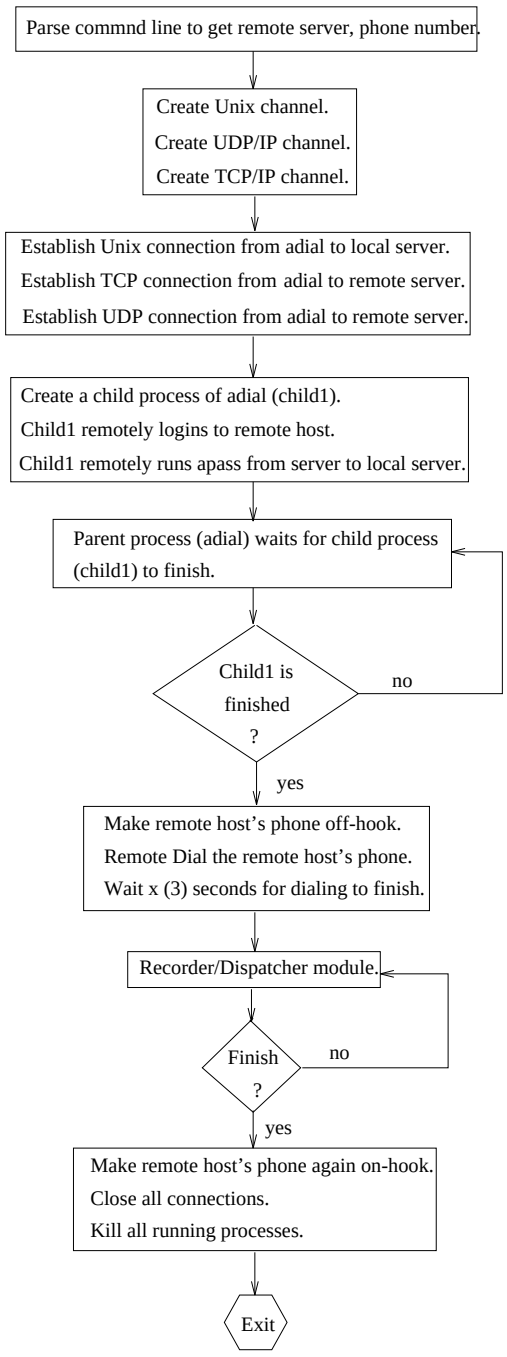

Figure 24: Flow diagram of rdial application.

6.2 Phase II: Conversion From UDP To AAL1:

Work has been carried out to modify the AN2 device driver so it supports AAL1 format instead of the current AAL5 format. As of now, AAL1 is already available for use, along with AAL5 format. To have the developed demonstration system support AAL1, modifications will be required both at the transmitter and the receiver. It is planned that the control information of AudioFile will be transported over a AAL5 VC (virtual circuit), while the voice data will be transported over a AAL1 VC. To convert from UDP/IP to AAL1, anywhere in the demonstration system that uses the UDP/IP will be replaced by the AAL1 system calls. Because of the modular implementation of the system, it is anticipated that modifications will not be difficult.

7 Graphical User Interfaces:

To ease the demonstration process, graphical user interfaces (GUI) were implemented. Tcl/Tk [14] scripts were used due to its easy-to-implement but effective feature, given the time constraint. The GUIs allow the users a very convenient tool of running the demonstration application without much difficulty, effectively hiding away all the complexity of the APIs beneath. The following discussions will be about the two GUIs implemented: the Alpha-Alpha, and the Alpha-DECaudio.

7.1 Alpha-Alpha GUI:

The Alpha-Alpha GUI implemented is shown in Figure 25. The Alpha-Alpha GUI's purpose is to allow the users (local and remote) to talk over the existing Ethernet or ATM network, through the handsets attached to the hosts. Its principle of operations is similar to that of *talk*, a popular and simple communications application found on most Unix workstations. As with *talk*, the remote user needs to have a server, *voice-serv* already run, listening for a call request. The user at the local host can then run an Alpha-Alpha GUI, *voice-cw.local*, which will have its *Host Name* field filled with name of some remote host. A pre-set list of values for *Host Name* is supplied with the GUI. The pre-set values can be found by activating the extension buttons (buttons with ...). The list can be altered easily to fit the user's personal configuration. Instructions for adding/deleting pre-set values can be found in the README file. The user can also fill in the fields manually. After specifying the name of the remote host to where the connection will be made, the user can press the *Talk* button to initiate a request to the remote host. The listening server, *voice-serv*, receives a request packet, and will pop up a small window and will wait for the input

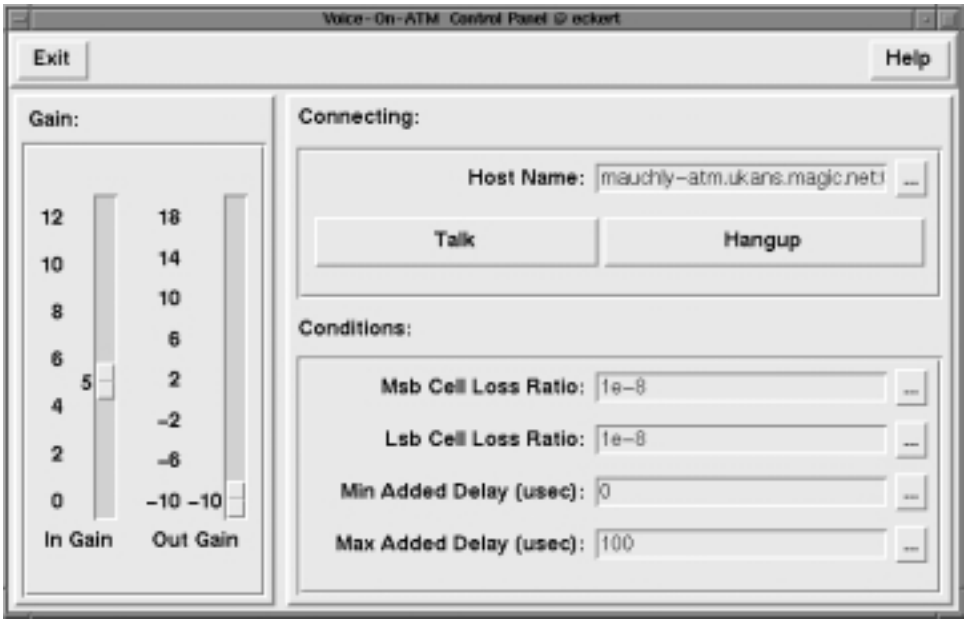

Figure 25: Alpha-Alpha GUI.

from the user. If the user presses the *Accept* button, the server will pop up a GUI exactly like the one shown in Figure 25. From the GUI, the user can then initiate the connection to the caller by only pressing the *Talk* button, without filling out the *Host Name* field. On the other hand, if the user selects the *Reject* button, there will be no GUI displayed. Whether the user accepts or rejects the connection request, the server will always send back the *Reply* packet to the caller. Upon receiving the *Reply* packet from the server, the caller will either have the connection automatically set up, or the refusal message displayed on his/her screen.

Once the connection is established betwen two users, voice quality can be controlled by changing the condition parameters: *MSB Loss Ratio, LSB Loss Ratio, Min Added Delay, and Max Added Delay* fields. Like the *hostname* field, the condition fields can be either filled in manually or chosen from the list of the pre-set values. The list can also be altered just as that of the *Host Name* field. The loss rates are limited between 0 and 1; whereas, the added delays are in units of micro-seconds. The *Min Added Delay* and the *Max Added Delay* specify the low and high limits of the uniform distributions, respectively. By varying the condition fields, the user can easily observe and evaluate the voice quality over the ATM network. To change input/output gains, the user can change the gain scale as shown in the main window.

Also shown in Figure 25 is the *Hangup* button which enables the user to shut down the one-way connection from him/her to the remote user. To have the

Figure 26: Alpha-DECaudio GUI.

whole two-way connection closed, the remote user also needs to activate the *Hangup* button. The *Exit* button will close down the GUI window. Instructions of how to use the GUI can be found by using the *Help* button.

7.2 Alpha-DECaudio GUI:

Figure 26 shows the Alpha-DECaudio GUI implemented. It implements the *rdial*'s operations as described in 6.1. Its purpose is to allow the user to dial any telephone, and conduct a full-duplex telephone conversation only via a handset attached to the host. As shown, the GUI only differs from the Alpha-Alpha GUI by the *Phone Number* field. The *hostname* specifies the name of the host which has the telephone interface hardware, DECaudio. Basically, the user at the local host, after running the *voice-cw.phone* to open up the Alpha- DECaudio GUI, can dial any telephone by specifying the name of the remote host which has DECaudio attached to it, and the *phonenumber* field. Pressing the *Dial* button will initiate the dialing service, during which dialing tones can be heard via the handset. The DECaudio is the hardware that is called to carry out the actual dialing. What happens is that the Alpha-DECaudio GUI communicates with the server, Alofi, at the remote host which has the DECaudio attached in requesting for a dialing process. As already described in earlier sections, Alofi has the ability to interact directly with the DECaudio hardware

in generating dialing tones, along with capturing record samples and playing play samples. The purpose of the condition fields is identical to that of the Alpha-Alpha GUI. Again, the input/output gains can be adjusted using the gain scales.

8 Problems associated with DECaudio's phone interface:

When purchasing DECaudio, we expected it to delivery a very good quality since it is a commercial product of Digital Equipment Corporation (DEC). Unfortunately, the telephone interface section of DECaudio has echo and noise problems which DEC now acknowledges. We did not find out the problems until six months after it was purchased due to the fact that we did not use it much during that time frame. In fact, according the plan we proposed, we were involved mostly with software development in implementing the proposed solutions during this early period. We did not plan to implement software that has the phone functionalities during this period. Therefore, we were unable to carry out test operations with the DECaudio telephone interface, particularly in our intended configuration, which happens to accentuate the echo problem. Furthermore, we found no acknowledgements of the problems from DECaudio documentation.

In searching for solutions to the echo problem, we tried to implement an echo canceller both in the Alofi server (software) and in the DECaudio itself (DSP). In moving the echo canceller from inside the Alofi server to inside the DECaudio, we were very hopeful that we could eliminate the echos due to the reduction in the return delay; unfortunately, it seems that even with echo canceller being placed inside the DECaudio hardware, it is unable to eliminate the echo. Several engineers who were involved in designing the DECaudio from DEC also believe the same. Ironically, these engineers were aware of the echo problem even before we purchased the DECaudio, but no re-design was ever attempted by DEC.

In an attempt to point out the nature of the echo and the noise problems, the following discussion will be devoted first to the configurations (hardware and software) of DECaudio's telephone interface, then to the different approaches we took in attempting to solve the problems.

8.1 Hardware Configuration:

Figure 27 [5] shows the detailed block diagram of the DECaudio Hardware. The four major components are the DSP, the shared RAM, the codecs, and the DAA

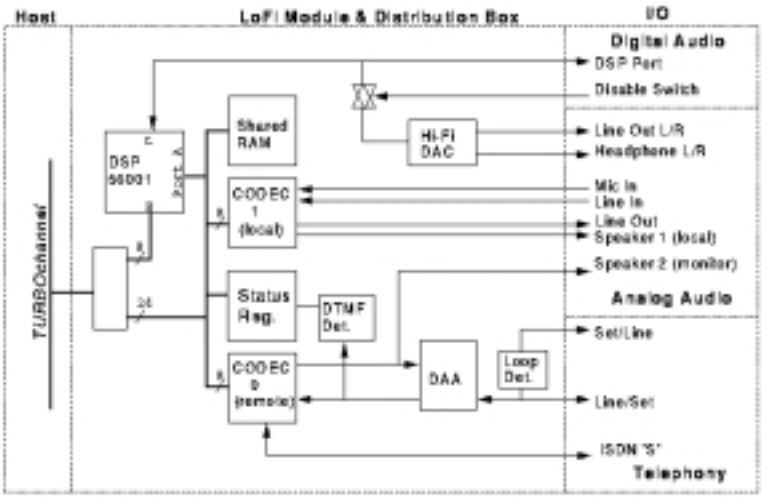

Figure 27: Detailed Block Diagram Of The DECaudio Hardware.

(Data Access Arrangement). The DSP has the following functionalities:

- DSP56001 has 3 memory sections: on-chip X Data Memory (X data RAM, and X data ROM where ROM contains pre-programed Mu-law and A-law Expansion Tables), on-chip Y Data Memory (Y data RAM, and Y data ROM where ROM contains pre-programmed sine-wave table), and Program Memory (512-word by 24-bit RAM) which provides a method of developing code efficiently. It also allows programs to be altered dynamically, allowing efficient overlaying of DSP software algorithms.

- After power-on reset of the DSP, the bootstrap mode provides a convenient way of loading the DSP56001 Program Memory with a program (in absolute format).

- The Shared RAM is an off-chip 34Kx24 bit static RAM array, and is shared by the DSP56001 and the host processor. Play and record samples to and from the codec0 (telephone codec) are temporarily stored in this shared RAM.

Figure 28 [5] shows the details of the DAA. As will be discussed later, the DAA's hybrid section (2-4 wire converter) is believed to cause the echos due to impedance mismatches and/or poor design.

8.2 Software Configuration:

Codec0 & Codec1:

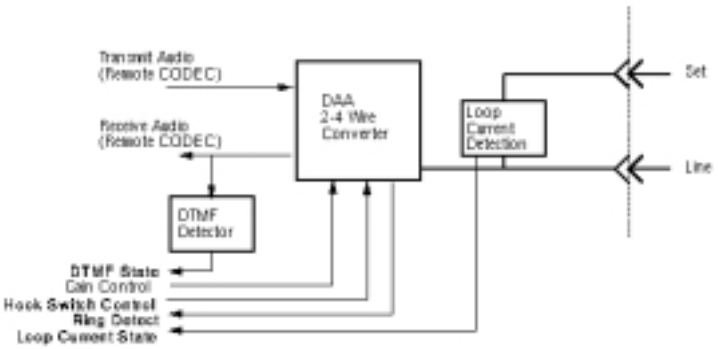

Figure 28: Detailed Block Diagram Of DECaudio's DAA, an analog telephone interface.

Programable Gains			
	Minimum	**Default**	**Maximum**
GX	0db	0db	12db
GER	-10db	0db	18db
GR	-12db	0db	0db
STG	-18db	-18db	0db
GA	0db	0db	24db

Table 1: Programmable Registers

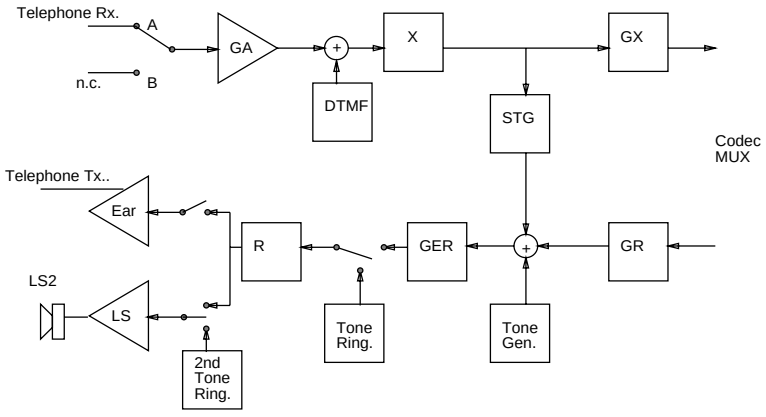

Figure 29: Detailed Block Diagram Of DECaudio's Codec0 (Telephone Codec).

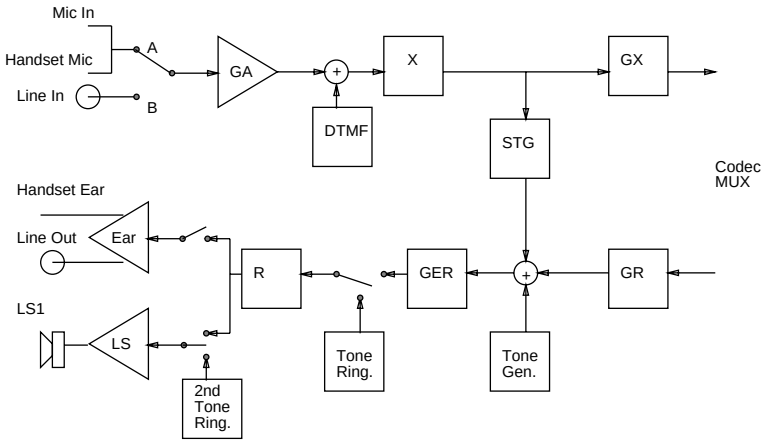

Figure 30: Detailed Block Diagram Of DECaudio's Codec1 (Handset Codec).

Table 1 [15] shows possible settings for the codecs. The detailed block diagrams of the codes are shown in figures 29 and 30 [5]. Programmable gain registers GX, GER, GR, and STG are each 16 bits in length, and can be programmed for infinite attenuation to break the signal path if desired. In particular, the coefficient 9008 (MSB=90, LSB=08) provides an attenuation of infinite on registers GR, GX, and/or STG, when they are enabled. Two consecutive register locations correspond to one gain coefficient. The LSB is transferred first to (or from) the microprocessor. It should be noted that the *Codec MUX* is on the host processor's side.

8.3 Echo Problem & Attempted Approaches:

In trying to determine where the echos were coming from, a number of test experiments were carried out which include:

- Carrying out the experiments as shown in figures 31 and 32. In both situations, we play samples captured from the handset to the DECaudio's codec0 (telephone codec), while recording samples captured from codec0 back to the handset. As expected, with configuration as in Figure 31, the return echo delay is significantly small; consequently, one could not hear any echo (but noise still !) at the handset. Actually, the echo is present, but the small delay causes it to appear as increased sidetone. In contrast, the configuration shown in Figure 32 introduces a large return echo delay which evidently includes the network delay; consequently, one will obviously hear the echos at the handset side. It should be noted that, in both experiments, we did not observe any echo and noise at the remote telephone side at all; in fact, the telephone side is amazingly clean.

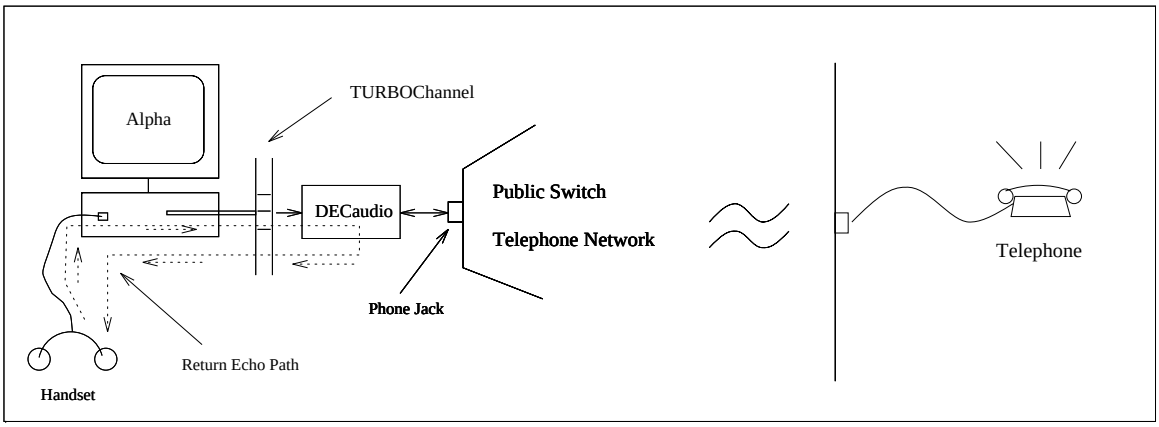

Figure 31: Configuration with no echos observed.

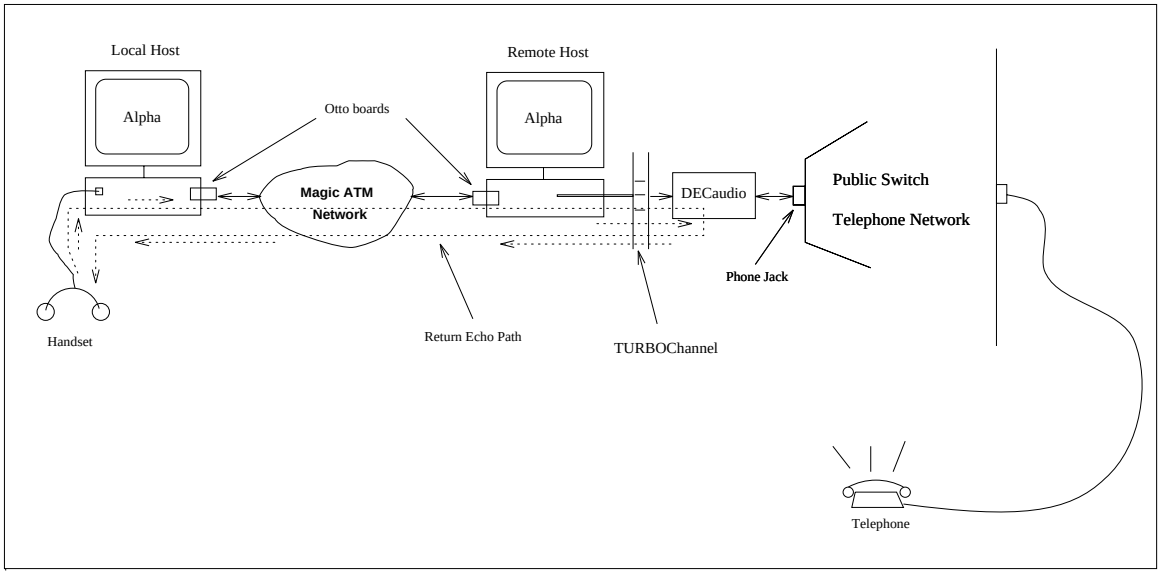

Figure 32: Configuration with echos observed.

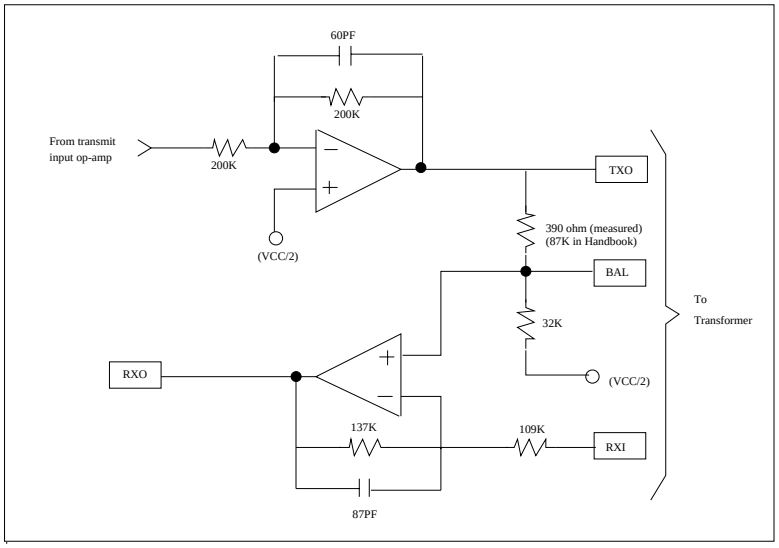

Figure 33: Transmit Driver/Receive Hybrid Section.

- Carefully examining the Alofi's dda codes to make sure that there is no loop back.
- Varying the sidetone, input and output gains to see if they have any effect on the echos.

After carrying out these experiments, we strongly believe that the DAA's hybrid circuit causes the echo problem. In particular, due to impedance mismatches and/or poor design, samples played out to the hybrid section are reflected back. Thus, when the Alofi server captures the recorded samples, the reflected samples are also captured.

Attempted Approaches:

8.3.1 Adding a compensation network:

We considered adding more resistors to the DAA's hybrid circuit to have a better impedance match. Figure 33 [16] shows the transmit driver/receiver hybrid section. In [16], the register between TXO (Transmit Output) and BAL (Balance) pins are specified at 87K; however, by using an ohmmeter, its actual value was found to be only 390 ohm. As shown, the hybrid consists of an op-amp which sums a portion of the TXO signal with the signal coming from RXI which is usually tied to the external coupling transformer. The receive gain from RXI (Receive Input) to the output of the hybrid is about 1.5 dB to make up for the typical insertion loss

of a coupling transformer. By design, the hybrid assumes that the TXO component at RXI will be 6 dB lower than that at the TXO pin. However, due to variations in the phone line impedances, the signal at RXI varies in phase and amplitude from the ideal case. As a result of this, the TXO component is not completely cancelled by the hybrid. To adjust the return loss of the hybrid, we tried to add an external compensation network attached to the BAL pin (as supplied with DECaudio, the BAL pin was left floating), as suggested by the Handbook. This did not help to reduce the echos.

8.3.2 Implementing an echo canceller inside Alofi server:

An attempt was carried out to implement an echo canceller inside the Alofi server. Figures 34, 35, and 36 show the echo buffers' structures, the Alofi architecture with Least-Mean-Square (LMS) algorithm embedded, and the flow diagram of the lms algorithm implemented, respectively. From Figure 34, as described earlier, the Alofi server carries out the update process once every 250 samples (32ms). At each instant of update process, a number of samples will be transfered from the Alofi's internal play buffer, *aDev->playBuf*, to the hardware play buffer for a play operation, and a number of samples will be transfered from the hardware record buffer back to the Alofi's record buffer, *aDev->recBuf*. The echo buffers, *EchoPrivate->echoBuf, EchoPrivate->recBuf, and EchoPrivate->echoBlock*, are introduced to hold the play and record samples, which are used by the lms echo canceller.

Figure 35 shows how the lms fits into the Alofi's structure. The energy detection is to make sure that adaptation of lms coefficients only occurs when there is no speech recorded; otherwise, the state of the echo canceller will quickly become unstable.

In estimating the number of taps needed, a measurement of echo delays at the Alofi's server was done. It was found out that the delay varied significantly, for each record/play operation. Figures 37 to 40 show four different instants of the reflected signal recorded from the codec0 for a flick. The x axis specifies the strength of the signal (normalized to 1) while the y axis designates the number of samples captured. The locations of the return echos were spotted by finding the maximum-value points which are shown on the figures. Notice the presence of noise in the figures. In obtaining the figures, all four experiments were carried with the same configuration. In particular, first of all, one flick was recorded into a file. The Alofi server was then modified so that it reads this input file and plays to the codec0 of DECaudio. At start up, the Alofi also opens an output file for holding the samples recorded from codec0 (Alofi's update process will carry out a record operation right after a play operation). The figures represent the data written by the Alofi server to those output files. In this way, all four instants were guaranteed to have the same

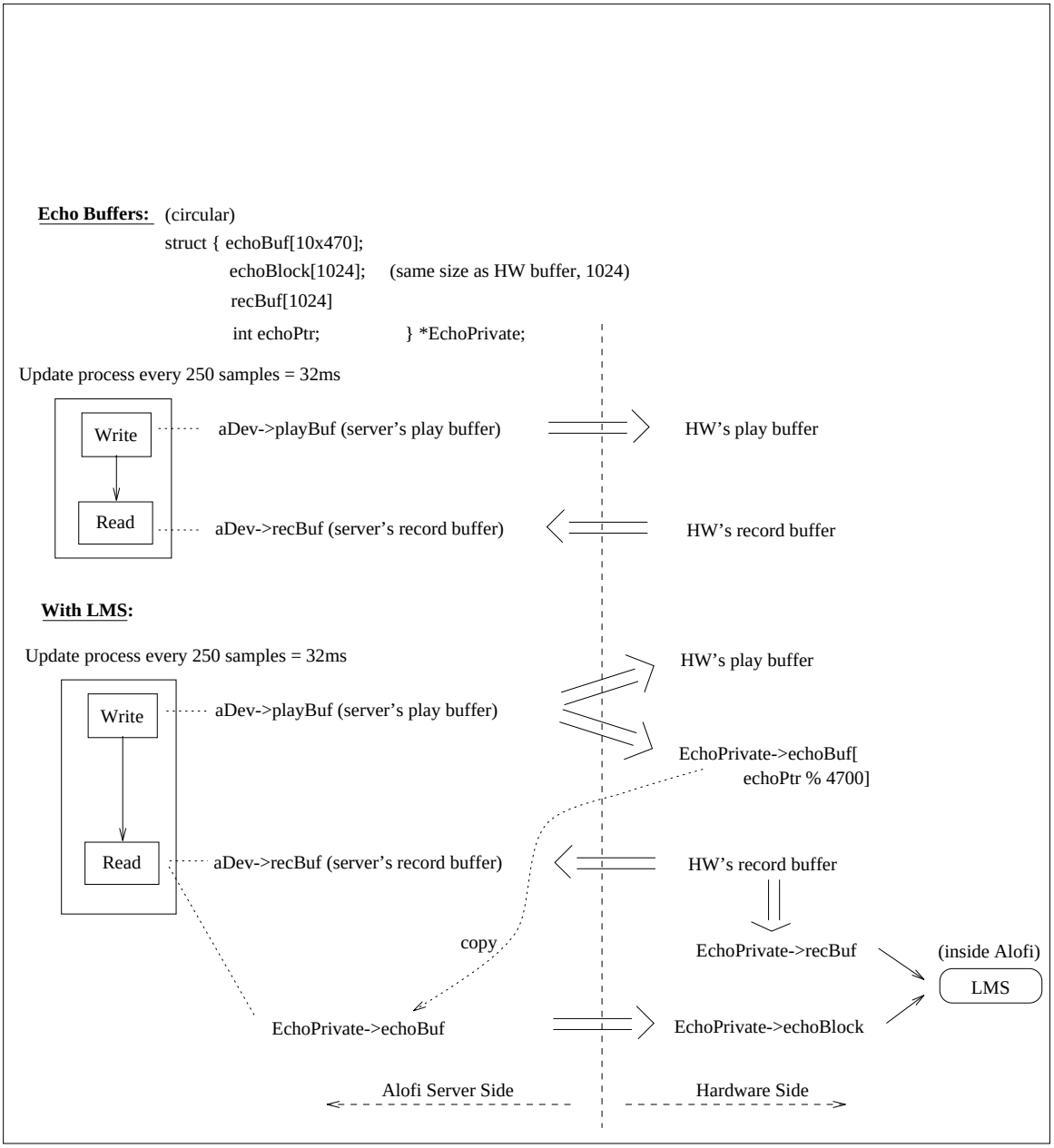

Figure 34: Echo buffers' structures and sample collection process.

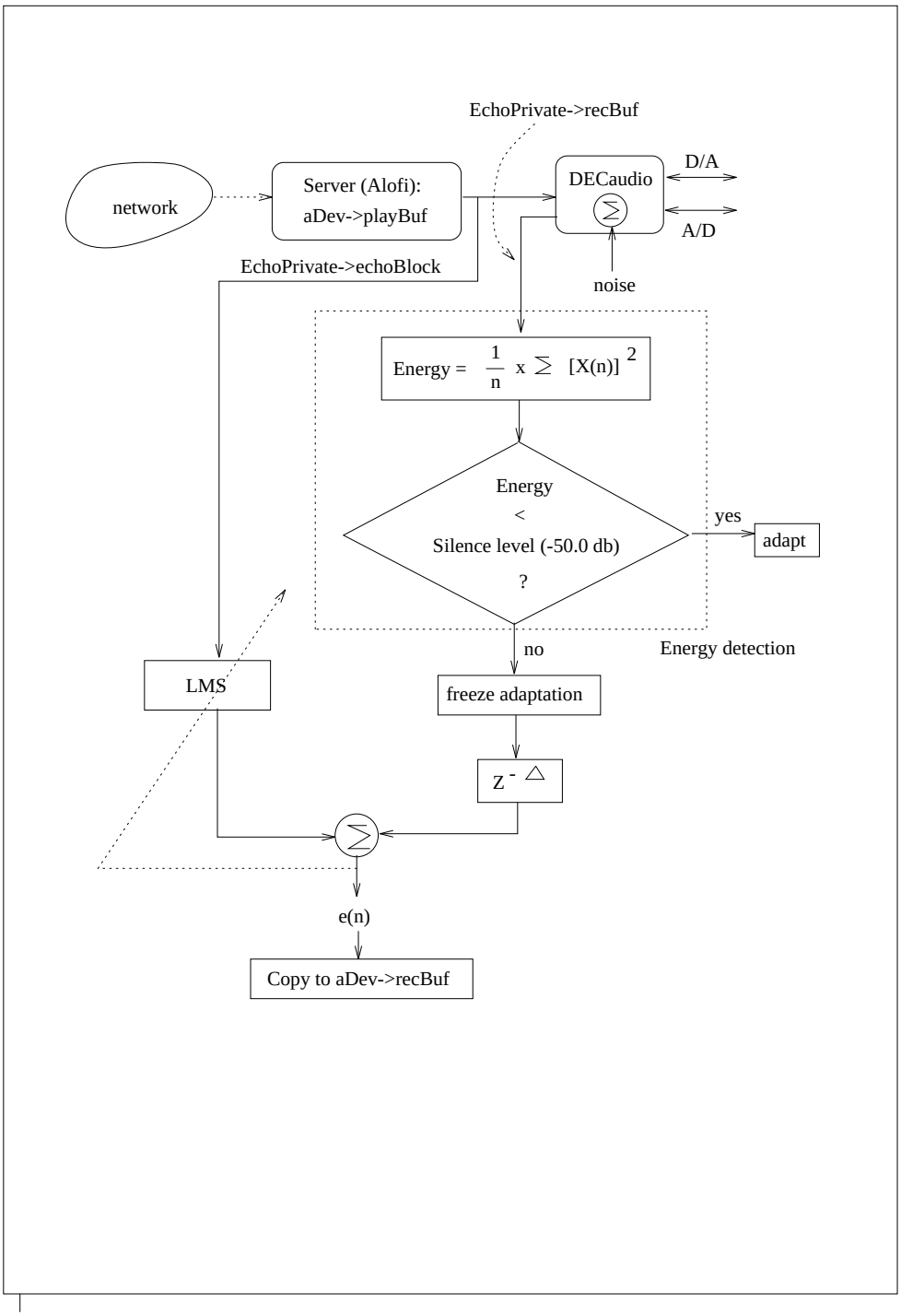

Figure 35: Alofi with echo cancellation embedded.

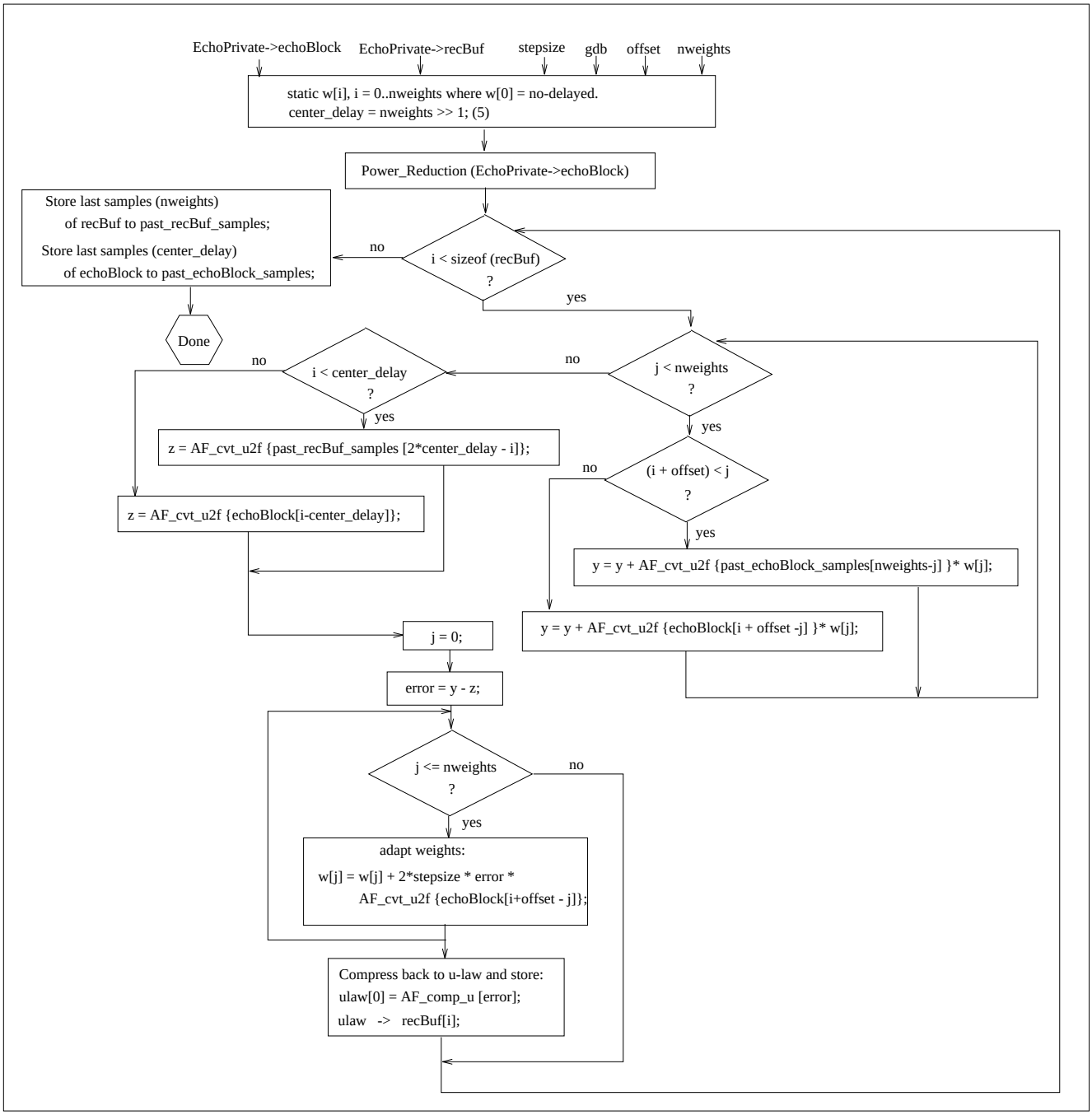

Figure 36: Flow operation of LMS algorithm implemented.

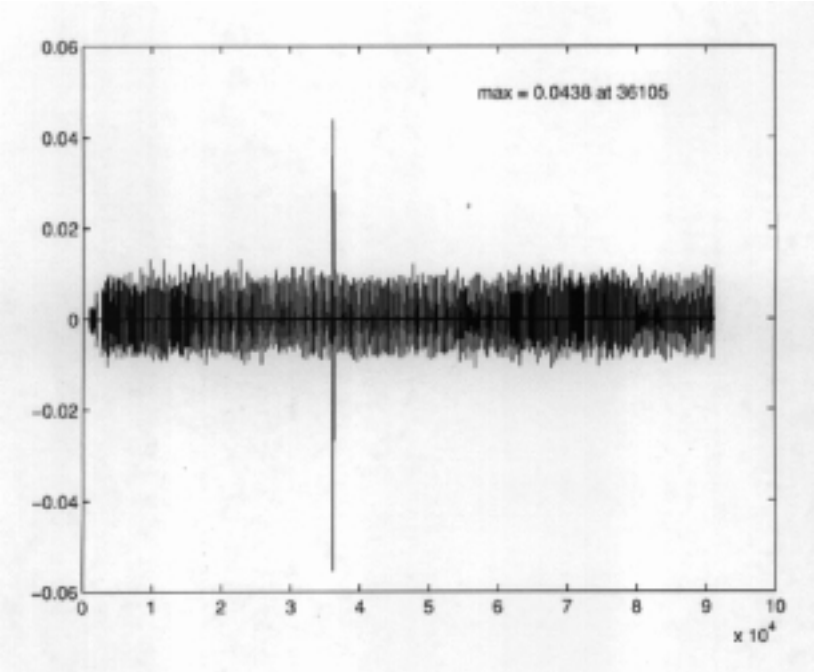

Figure 37: Echo recorded from a flick at time t1.

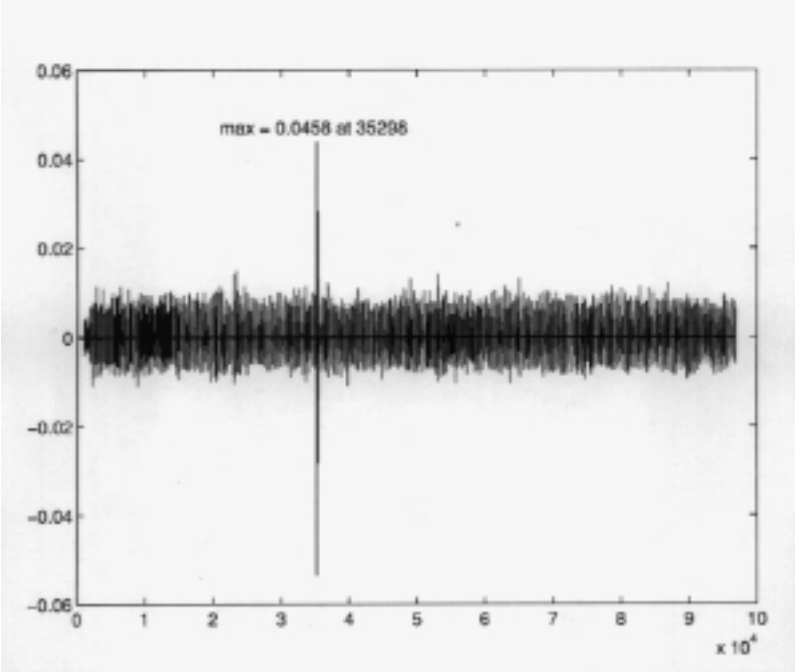

Figure 38: Echo recorded from a flick at time t2.

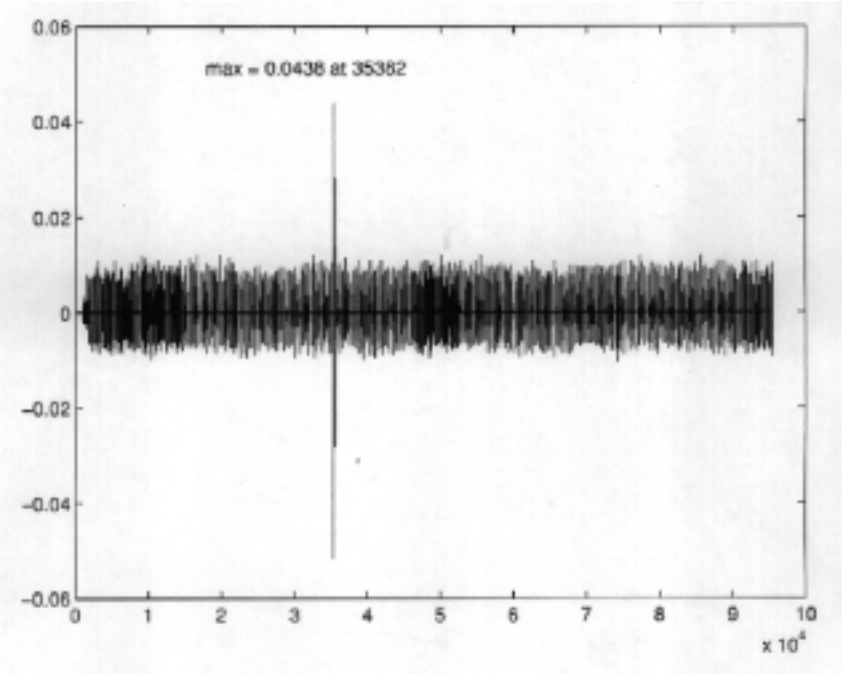

Figure 39: Echo recorded from a flick at time t3.

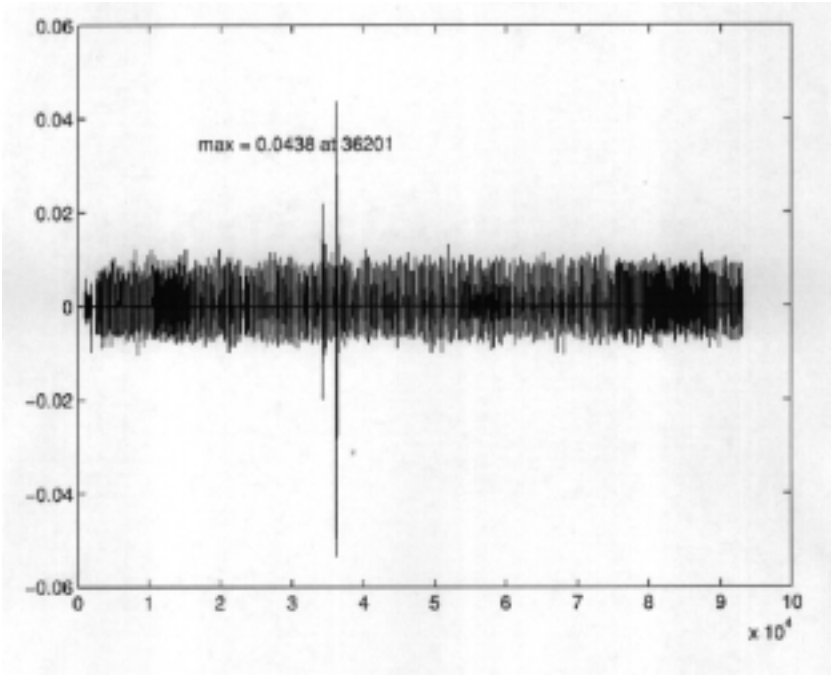

Figure 40: Echo recorded from a flick at time t4.

experimental configurations. As can be seen, the variation is significantly large, specially between Figure 37 and Figure 38 (36105 - 35298 = 807 samples). Given this large variation, an echo canceller with a number of taps of at least 800 samples is required. Taking into the consideration that the Alofi server needs to carry out the update process once every 250 samples, the echo canceller definitely will not work. Even if the Alofi server's update period is increased (as we actually did try), we still experience the problem of adapting to that large variation.

8.3.3 Implementing an echo canceller inside the DSP:

To reduce the return echo delay, we attempted to move the echo canceller closer to the DAA's hybrid circuit. An echo canceller written in assembly by DEC's CRL was embedded in the *ukernel* code of the DSP, main.lod. We purchased the Motorola's DSP Development Software that contains an assembler and a linker. The assembly code is first assembled by the assembler which translates the source statements into relocatable object files (.CLN). These object files are then processed by the DSP linker to produce absolute executable file (.CLD). Finally, the absolute executable files are converted into absolute .LOD files before they can be downloaded into the DSP.

At start-up time, server initilizes hardware (codecs, RAM, ROM, DSP6001), and loads the precompiled absolute .lod file (main.lod) which contains the echo cancellation code into the DSP's RAM section. For each sample played to codec0, the DSP applies the Least-Mean-Squared (LMS) algorithm to the sample. It should be noted that the number of taps implemented in the echo cancellation code is set at 248 which is the maximum number of taps that can be implemented on the DSP chip; otherwise, if it is greater than 248, mixing of X and Y data will occur.

Previous attempts at echo cancellation by DEC engineers had not been particularly successful. In our configuration, the echo may be returning to the DSP in less than one sample time which would be problematic. We eventually abandoned this attempt.

8.4 Noise Problem & Attempted Approaches:

Attempted Approaches:

We also experienced the noise problem from the DECaudio's phone interface. Several experiments were done in attempt to fix the noise problem. They include:

- Measuring the noise level around DECaudio's DAA hybrid section, using an oscilloscope. The noise level was found quite high.

- Shielding the whole DECaudio with aluminum in hope of reducing inter-device noise. We found that when the DECaudio was not shielded, the noise pattern, and thus level, varied with CPU load. This would indicate CPU and other devices (disks, power supply, to name a few) also play some role in the noise problem. When it was shielded, there was still noise but the noise pattern did not seem to vary much with the workstation load.

- Attempting to move the DECaudio card outside the Alpha and shielding it with aluminum. But we later found out that it was not feasible to implement due to the limitation of the bus length that could be extended outside the workstation.

From the information we have gathered, we strongly believe that the noise is coming from the DAA's hybrid section. To reduce the noise level, we can set the gains (input/output) at the appropriate levels which vary on different hardware used.

9 Conclusions & Areas For Future Work:

AudioFile is a very convenient software tool to perform audio processing. With DECaudio, AudioFile provides essential telephone processing and high-quality audio processing. As with the current version (version 3), AudioFile does not have ISDN support even though DECaudio has the capability to interface with an ISDN terminal at the S interface point. We already checked the status of ISDN from Digital Equipment Corporation near the end of last year just to find out that ISDN is still under the phase of field testing. Furthermore, we also considered a field test with DEC, using the Alpha's ISDN capability (there is an ISDN's S interface port on every DEC Alpha). Unfortunately, our local site does not have ISDN capability to carry out a field test. As a result, we have thought that it should be best to delay any ISDN testing until adequate equipment is available.

On the other hand, we envision that future software development on top of AudioFile to support ISDN will provide us a very high quality of voice. Because the S interface bypasses the DAA hybrid circuitry of DECaudio, the availability of ISDN on AudioFile would mean complete elimination of current echo, and noise problems observed in DECaudio. Modifications to AudioFile to add ISDN capabilities would require development of a new device driver under the DDA (Device Dependent Audio) section.

As described earlier, DECaudio supports HiFi audio processing. With appropriate HiFi sources connected to DECaudio, CD quality (44.1-KHz sampling)

audio can be captured and transmitted over the network. On the other hand, HiFi samples received from the transmitter can be played out to the external HiFi device connected to the DECaudio. All the recording and playback of HiFi samples are via stereo device 3 which has two channels (left and right). With these HiFi capabilities, software development for CD-quality applications (broadcastings, for instance) on DECaudio would be undoubtedly of great interest.

10 Acknowledgements:

I would like to especially thank Dr. Joseph Evans for his continous guiding and support in understanding AudioFile's reasonably complex structure. His insightful suggestions of how to go about modifying AudioFile played a significant role in the success of the implementation part of the project. I also owe special thanks to Dr. David Petr for giving me the opportunity to work on the project. His continous help, support and encouragement words, when I have questions and, especially, when things seem rather slow, undoubtedly help me tremendously. I also wish to thank Dr. Victor Frost for his continous support when I need help on different topics. I greatly appreciate the lending of his book on adaptive signal processing. I also would like to thank Dr. Gary Minden for providing me a good start on working with AudioFile. I also would like to express my sincere thanks to Dan DePardo for his assistance in carrying out experiments, as we tried to solve the echo and noise problems. Finally, I would like to thank my co-workers, Mohammed Shanableh and Mark Mischler, for their helpful suggestions on my part. I have found this project a real experience and a first building step in my engineering profession.

References

[1] David W. Petr, Mark Mischler, Mohammed Shanableh, Victor S. Frost, "Voice Transport Via ATM Networks: Proposed System Solutions," *TISL Technical Report TISL-10610-01,* July 1994.

[2] Mohammed Shanableh, Joseph B. Evans, David W. Petr, "Voice Transport Via ATM Networks Using DS0 Packetization" *TISL Technical Report TISL-10610-02,* July 1994.

[3] T1A1.6 Editorial Group, "Draft Technical Report On Voice Packetization," *T1Y1 23 Speech Packetization Project,* August 4 1993.

[4] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart, "Audiofile: A Network-Transparent System for Distributed Audio Applications," *Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab,* 1993.

[5] Thomas M. Levergood, " LoFi: A TURBOChannel audio module. CRL Technical Report 93/9," *Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab,* 1993.

[6] W. A. Montgomery, "Techniques for Packet Voice Synchronization," *IEEE Journal On Selected Areas In Communications,* vol. SAC-1, no. 6, December 1983, pp. 1022-1028.

[7] Steven McCanne, "An Echo Canceler For Workstation Audio," *EE225A Term Project,* Computer Science Division, University of California, Berkeley, May 3, 1993.

[8] David G. Gesserschmitt, "Echo Cancellation in Speech and Data Transmission," *IEEE Journal On Selected Areas In Communications,* vol. SAC-2, no. 2, March 1984, pp. 283-297.

[9] J. W. Emling, and D. Mitchell, "The effects of Time Delay and Echoes on Telephone Conversations," *Bell Syst. Tech. Journal,* Nov. 1963, pp. 2869-2891.

[10] Nikil Jayant, "High Quality Networking of Audio-Visual Information," *IEEE Communications Magazine,* September 1993, pp. 84-95.

[11] Bernard Widrow, Samuel D. Stearns, "Adaptive Signal Processing," *Prentice-Hall Signal Processing Series,* 1985.

[12] Martin De Prycker, "Asynchronous Transfer Mode - Solution For Broadband ISDN," *Ellis Horwood Series In Communications And Networking,* 1993.

[13] W. Richard Stevens, “Unix Networking Programming,” *Prentice Hall Software Series,* 1990.

[14] John K. Ousterhout, “Tcl and The Tk Toolkit,” *Addison-Wesley Professional Computing Series*, 1994.

[15] Advanced Micro Devices, Sunnyvale, CA. *AM79c30A Revision F Data Sheet on the Digital Subscriber Controller Circuit*, June 1992.

[16] Dalas Semiconductor Corp., Dallas, TX. *Teleserving Design Handbook,* March 1990.

[17] Motorola Inc. *DSP56000/DSP56001 Digital Signal Processer User’s Manual*, revision. 2, 1990.

[18] Motorola Inc. *DSP Development Software*, 56000CLASF-5.3-00398, 1994.