Tools for Building Asynchronous Servers to Support Speech and Audio Applications

Barry Arons
Speech Research Group
MIT Media Lab
20 Ames Street, E15-353
Cambridge MA 02139
+1 617-253-2245
barons@media-lab.mit.edu

ABSTRACT

Distributed client/server models are becoming increasingly prevalent in multimedia systems and advanced user interface design. A multimedia application, for example, may play and record audio, use speech recognition input, and use a window system for graphical I/O. The software architecture of such a system can be simplified if the application communicates to multiple servers (e.g., audio servers, recognition servers) that each manage different types of input and output. This paper describes tools for rapidly prototyping distributed asynchronous servers and applications, with an emphasis on supporting highly interactive user interfaces, temporal media, and multi-modal I/O.

The *Socket Manager* handles low-level connection management and device I/O by supporting a callback mechanism for connection initiation, shutdown, and for reading incoming data. The *Byte Stream Manager* consists of an RPC compiler and run-time library that supports synchronous and asynchronous calls, with both a programmatic interface and a telnet interface that allows the server to act as a command interpreter. This paper details the tools developed for building asynchronous servers, several audio and speech servers built using these tools, and applications that exploit the features provided by the servers.

KEYWORDS

Audio servers, remote procedure call, asynchronous message passing, distributed client-server architecture, speech recognition and synthesis, speech and audio applications.

1 INTRODUCTION

The software tools described in this paper draw from predecessor systems, and emphasize the features found to be important in building user interfaces and highly interactive applications involving speech and audio. The goal of this work is to provide an environment for the rapid prototyping of distributed asynchronous servers and applications, with an emphasis on supporting multiple media and multi-modal I/O, while working with existing user interface software and within current software engineering paradigms.

Distributed client/server models have been in use for over a decade, and are becoming increasingly prevalent in multimedia systems and advanced user interface design. In many current graphical user interfaces, such as those using the X Window System, client applications communicate with a window system server that manages screen output and mouse/keyboard input [17]. It is advantageous to use a similar software design methodology for developing new user interfaces that use other forms of I/O, such as speech recognition and synthesis, or incorporate temporal media, such as recorded speech. This distributed design allows multiple applications to share limited I/O resources (e.g., a display, microphone, loudspeaker, etc.) without any knowledge of the other applications. The software tools described in this paper assist in the creation of servers and client applications, and address some of the issues of prototyping and debugging in such a distributed environment.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1992 ACM 0-89791-550-X/92/0011/0071...$1.50

1.1 Application Environment

A multimedia application may play and record audio, use speech recognition input, speech synthesis output, access remote information services, and use a window system for graphical I/O. The architecture of such a system can be simplified if the application communicates to multiple servers (e.g., audio servers, recognition servers, synthesis servers) that each manage different types of input and output. In such an environment, it is desirable to: (1) build many servers on a standardized maintainable low-level architecture, rather than hand-crafting each server, and (2) develop tools that address the specific needs of managing time-varying media, such as audio.

Several audio servers have been based on the X Window System model of client applications communicating with a server that manages all speech and audio I/O in the workstation [1, 2, 4]. Client applications make requests to the server to perform audio functions such as playing a sound file. These requests are ideally asynchronous to avoid round-trip network delays, and to permit the server to send user input events to clients. User interface servers that handle temporal media, such as audio or video, must also be concerned with other types of events. For example, after an application places a request to start playing a sound, the application should ideally receive asynchronous notification when the play action has completed. An application can also request events from the server at regular intervals. For example, while a sound is being recorded, energy information sent by the server can be visually displayed by a client application. These types of events are significantly different from those associated with window systems, yet are essential when using temporal media [7].

1.2 Overview

The *Socket Manager* handles low-level connection management and device I/O. It supports a callback mechanism for connection initiation, shutdown, and for reading incoming data. The *Byte Stream Manager* is a higher level tool consisting of a run-time library and a remote procedure call (RPC) stub generator. The specifications used in the RPC compiler are identical to the syntax used in the calling and called programs, and both synchronous and asynchronous calls are supported. Servers built on the RPC library simultaneously supports both a programmatic interface and an ASCII telnet interface. The ASCII interface is a simple yet powerful tool for rapid prototyping and debugging servers and applications.

This paper first describes the asynchronous server building tools. It then details audio and recognition servers, as well as several applications, that are built using these tools. The paper concludes with a review of related work.

2 SOCKET MANAGER

The *Socket Manager* (SM) library simplifies handling asynchronous input, and the communications setup of distributed programs. The SM hides the underlying interprocess communication code that deals with socket binding, listening, and accepting that is replicated when building network based applications. The SM currently runs under Unix using internet domain sockets with reliable TCP/IP communications [10].

A client opens a connection to a server by specifying a service name and an optional host name. When a new service is created, a callback procedure is registered with the SM. This procedure is asynchronously executed when an incoming service request is received. A callback is also registered for reading data when it is available from a communications socket or device. The application (or the Byte Stream Manager) is responsible for the actual reading of data. In addition to handling information from sockets, additional file descriptors can be registered with the SM. This allows physical devices, such as digital sound equipment or speech recognizers, to be used within the framework of an SM-based server.

The SM library is used by both servers and clients. Once initialized, the server and application processes block until: (1) there is data to be read from a device or socket, (2) there is a request for setting up or tearing down a connection, or (3) an application defined timer expires.

The portion of the SM that executes callbacks when data is available overlaps in functionality with the X Toolkit (Xt). To allow maximum flexibility, the SM library is built in two forms; one is for stand-alone client and server processes without graphical interfaces, the other is a compatibility library that uses the SM functions built on top of Xt routines.

A single server can support multiple clients as well as multiple services (see also section 4.1). For example, a server that provides speech processing capabilities can support two related services: one service can provide the raw signal processing capability to clients, while the other service can provide current status information about the server itself, such as number of clients, memory usage, etc. Each service is provided on a separate communication port.

2.1 Clients, Servers, and Peers

The terminology of "client" and "server" can be confusing, particularly when dealing with asynchronous messages sent in both directions by communicating parties. In the framework described in this paper, a client communicates with a server by a synchronous remote procedure call or an asynchronous message. The server typically sends asynchronous messages, or RPC replies, back to the client. The client and server processes thus act as both sender and receiver of messages.

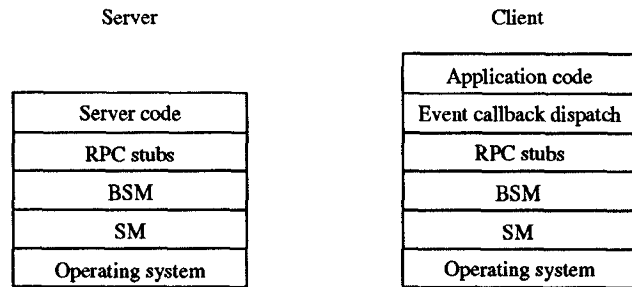

Figure 1: Server and client architecture.

There is a symmetrical peer relationship between an application and a server. The Byte Stream Manager and Socket Manager exploit the duality between clients and servers, thus simplifying the design and implementation of the software. A single mechanism is used for communicating between client and server, and for communicating between server and client.

3 BYTE STREAM MANAGER

The *Byte Stream Manager* (BSM) works on top of the Socket Manager to format and manage asynchronous messages and remote procedure calls. The BSM consists of an RPC compiler (stub generator) and run-time library tuned for writing X-like servers and clients. The overall architecture of a typical server and client is shown in figure 1.

The BSM simultaneously supports three communication modes that are selectable on a per-connection basis:

1. "C" programming interface with efficient (non-human readable) data encoding.
2. "C" programming interface with human readable data encoding. This mode is used for debugging client/server communications.
3. An ASCII telnet interface for interactively communicating with a server. In this mode the server acts as an interpreter, and is useful for testing and debugging purposes.

3.1 The Asynchronous RPC Library Generator

The functions supported by a server are defined in a procedure call specification file. The format of the specifications is based on function prototypes in ANSI C. Complex data types, such as arrays and structures, are deliberately not supported since their inclusion would complicate the ASCII telnet interface to a server.

The specification of a function and its arguments implicitly define whether a call is synchronous or asynchronous. If the function returns a value, or can modify any of its arguments[1], the function is handled as a blocking RPC call; if the function does not return a value and cannot change any of its arguments[2], the function is handled as an asynchronous message. A remote procedure call is implemented as one asynchronous message from client to server, and a corresponding reply from server back to client.

Procedures are declared as either returning an `int` or as `void` (no return value). One argument is required for the connection-specific

[1] I.e., at least one argument is call by reference.

[2] I.e., all arguments are call by value.

file descriptor that identifies the entity on the other end of the peer connection. Any number of other server-specific arguments can be specified. The following data types are currently supported:

data type	description
`int`	32 bit integer
`double`	double floating point number
`char`	single character
`string`	null terminated C string
`bytes`	a structure containing arbitrary data, and its length (such as for sending audio data)

A pointer to a data type (call by reference) is designated with a preceding "*" if an argument can be modified by a remote procedure. The syntax of an RPC declaration can be summarized as:

```
int|void funcname (int fd
                   [, datatype [*]name]...
                  );
```

Typical specifications look like:[3]

```
1. void s_start(int fd, int timeMs);
2. void s_record(int fd, string name);
3. int  s_get_length(int fd, string name);
4. void s_get_dir(int fd, *string dirName);
```

Examples 1 and 2 are sent as asynchronous messages. Example 3 is a synchronous RPC since the function returns a value. Example 4 is also synchronous since the string `dirName` is passed by reference, to be filled in by the server.

After an RPC is started, the BSM buffers all incoming messages until a response is received that matches the original RPC request. Once the RPC is complete, and all application code has finished executing, the buffered messages are dispatched in a first-in first-out (FIFO) manner. Each message contains a sequence number and a time stamp. The sequence number is used for consistency checking, and for synchronizing RPC return values. The time stamp is available to time-dependent applications and user interfaces.

Because of the symmetrical relation between applications and services, there are usually two specification files associated with each service. One file specifies the synchronous and asynchronous calls from the client to the server, the other specifies the asynchronous events that are sent from the server to the client.[4]

Compiling an RPC specification file[5] produces three code files (`.c`) and a header file (`.h`). Two of the code files represent the RPC stubs for the server and client respectively. The third code file is typically linked into the application, providing an improved programmatic interface to the application registered callbacks. Rather than enforcing predefined callback names on an application, the application registers interest in particular events, that in turn call local client-defined functions. This mechanism allows run-time, rather than compile-time, binding of functions to server generated events.

3.2 The Telnet Interface

Telnet [10, 16] can be used to interactively communicate with a server, allowing it to act as a command interpreter. This allows access to a server from any machine without having any code running on the remote machine. Figure 2 illustrates a sample telnet session for the audio server described in section 4.1. This ASCII string interface also allows clients to be written in environments where the C-oriented RPC generator cannot easily be used (e.g., from a lisp machine).

Server functions are called with the names (or shortened names) and arguments defined in the specification file. Function names and arguments are entered as strings, for example a function call of the form `s_start(fd, 8000)` would be entered in the telnet interface as "`s_start 8000`" or more simply as "`start 8000`".[6] Asynchronous events from the server, or the return value and arguments of synchronous RPC calls, are similarly sent and displayed as ASCII strings. The telnet interface also supports a crude form of help that is generated by the BSM compiler. A list of all functions supported by the server (with their arguments) is printed if a "?" is sent to the server.

This style of interface to a server was found to be extremely useful for debugging servers and applications. Commands for a server can be easily typed or pasted into a window with a telnet connection to a server. This allows commands or scripts to be interactively tested without compiling any code. In addition to the telnet interface, the BSM library can be run in a mode where all the data packets sent between peers are printed. This feature is useful in the often difficult task of debugging distributed programs.

4 SERVERS

This section details several audio and recognition servers that have been built using the SM and BSM. In addition, several other systems have been implemented including: a server for finding out where users are located [13] and a video snapshot server.

4.1 SparcStation-Based Audio Server

An existing high-level library interface to the audio capabilities of a Sun SPARCstation [20] was converted to run as an asynchronous server. Replacements for the old synchronous procedure calls fell into three categories:

1. Calls to obtain status information from the server (e.g., how much of the current sound file has been played) were converted into synchronous RPCs.
2. Calls to set parameters (e.g., setting playback gain) or perform simple operations (e.g., copy a sound file, start playing a sound), were replaced with asynchronous messages.
3. Asynchronous events are sent to the application when the server completes a time-based operation. Originally, the application would poll, using a library routine, to determine when playing or recording was finished.

The termination condition (e.g., maximum length reached, silence detected) of all time-based operations is passed into the application registered callback function. For example, when an audio recording has ended, an asynchronous "record done event" message is sent by the audio server indicating why the activity completed. Figure 2 shows a sample interactive telnet session with the audio server.

In addition, a variety of new asynchronous calls were created that were awkward to implement in a conventional library-based procedure call environment. For example, a callback can be registered to asynchronously receive energy values while recording, at intervals specified by the application. A mechanism was also created for asynchronously playing or recording from buffers managed by the application. To play from buffers, the client sends a message to the server including the first buffer of audio data. When the internal buffer in the server needs more data, an asynchronous message

[3] Note that any number of arguments can be used.

[4] A server typically does not make any RPCs to a client.

[5] The RPC stub generator is built using `lex` and `yacc`, standard compiler building tools.

[6] Since only one connection is possible with telnet, the file descriptor (the first argument) is not used. Function names of the form "`SoundStart`" can be shortened to "`Start`".

User input is in this font. `Server output is in this font.`	Comments
?	list server functions and arguments
`void s_halt`	halt current play or record action
`int  s_length (string)fileName`	get length of a sound file
`void s_pause_detect (int)trueOrFalse`	enable pause detection when recording
`void s_get_dir (string *)dirName`	get current sound directory
...	etc.
length greeting.snd	what is length of sound file greeting.snd?
`"reply" returnValue=3457`	length is 3457 milliseconds (3.457s)
get_dir	what is current sound directory?
`"reply" dirName="/sound/barons"`	directory name is returned
pause_detect 1	stop recording when user pauses
record test.snd	record sound file
`"s_record_begin_ev" data=0 handle=0`	record started event
`"s_record_done_ev" data=0 termination=5 handle=0 length=3604`	recording terminated after 3.604s
play test.snd	play sound file
`"s_play_begin_ev" data=0 handle=0`	play started
`"s_play_done_ev" data=0 termination=6 handle=0 position=3604`	play ended normally
play test.snd	play sound file
`"s_play_begin_ev" data=0 handle=0`	play started
halt	user interrupts playing
`"s_play_done_ev" data=0 termination=1 handle=0 position=926`	play interrupted after 0.926s

Figure 2: Sample output from interactive telnet session.

is sent to the client, executing the client-defined callback. In this callback, a new data buffer is typically sent back to the server asynchronously. This sequence of operations is similar in spirit to an RPC by the server to obtain data from the client. However, with this scheme neither the client nor server is ever blocked waiting for the reply to a synchronous remote procedure call.

Sounds can be played or recorded by the server from either files or buffers. Playing from files is easiest for an application, as the server internally handles all file manipulation and I/O. Using buffers provides more flexibility for an application, permitting, for example, sounds that are synthesized on-the-fly to be played in real-time. When using buffers, however, a greater burden is placed on the client to manage file I/O and send data to the server when it is requested.

Multi-client support is incorporated into the audio server. The current model of sharing of audio resources falls into the areas of client priorities, locking, and data interest events. In general, the X Window System philosophy is followed—provide flexible mechanisms in the server, rather than policies of how the server is to be used. However, some management of resources is necessary, and, for now, a sharing policy has been incorporated into the server. This allows multiple applications to use the server simultaneously, without having to write a separate audio manager (the equivalent of a window manager).

Priorities

Applications are classified by three priority levels (urgent, normal, background). Priorities are assigned on a per application basis, rather than on a per command basis—for the types of audio applications envisioned, this coarse priority structure appears sufficient.[7]

A higher priority application will preempt a lower priority application. For example, if a normal priority application is playing sound A, and a request arrives from a high priority application to play sound B, sound A is halted and sound B is played. When sound B is finished playing, sound A is resumed.[8] The server sends events to the lower priority client indicating the preemption and resumption of the play activity. After receiving a preemption event, the client can cancel the pending play request so that it is not automatically resumed. Requests of equal or lower priority that arrive from other clients are queued in a first-in first-out manner, and are executed when the audio play or record resources become available.

Recording priorities are handled in a similar manner, except that record actions are terminated on preemption, and are not automatically restarted. This behavior was chosen so that applications can provide appropriate prompting or feedback to the user before restarting a recording.

Locking

For some applications it is desirable to lock resources in the server for the exclusive use of the client. This ability is needed if an application must ensure an uninterrupted sequence of play or record actions. For example, one application could request to play sound file A, then sound file B. After A starts playing, a separate application with the same priority can request to play file C. However, because of the FIFO queue of play requests, the sounds may be played in A-C-B order instead of the desired A-B-C order. A similar intermixing of playing and recording is also possible as the server is capable of playing and recording simultaneously.

A client can request to lock all the server's resources for its exclusive use.[9] An example where this is needed is in a workstation-based telephone answering machine. The answering machine application

[7] Although it is possible to for an application to change its priority dynamically.

[8] The sound is restarted at a point several seconds before the interruption.

[9] The VOX audio server controlled a larger array of speech and audio devices and allowed locking of subsets of devices, rather than the entire server [4].

must lock the server before the telephone line is taken off-hook, since no other applications should be able to play or record over the phone until the call is completed.

Application writers are discouraged from locking the server unless absolutely necessary, and should release the server lock as soon as possible, because all other audio activity is blocked while a client has the server locked. Once the server is locked, the client has complete control of the server—a higher priority client cannot interrupt or preempt it.

Data and Interest Events

To allow for flexibility in the types of applications[10] that can be written, clients can register interest to receive events for a variety of server activities. A client, for example, can request an event whenever the server begins playing or recording for *any* client.

Multiple clients may need to simultaneously receive sound data during recording, however, only one process (the audio server) can read audio data from the recording hardware. For example, while one client is recording a telephone message, another client may want to receive a copy of the recorded data to detect the presence of DTMF tones[11], or to present a visual display of the record level (e.g., a VU meter). To enable such clients to work together, the server allows record data to be distributed to any number of interested clients.

Clients register the circumstances under which they wish to receive record data, as well as their intended use of the data. For example, a client can specify that it is going to perform speech recognition whenever another client is recording. This scheme is flexible, allowing clients to get record data under a variety of conditions, such as: (1) when any other process is recording, (2) when another process is performing a particular operation on the record data, such as DTMF detection, or (3) at all times, regardless of other audio server activity.

Data interests are considered "background" tasks, and multiple data interests are handled simultaneously. Priorities are only used with "primary" tasks, such as when the server is playing or recording from files.

The recorded data is distributed to interested clients as asynchronous events. The data is not broadcast or multicast, as the clients may have started recording at different times or may have requested to receive different length data packets. A process that captures background speech (section 5.1), for example, may want to receive large packets of audio at infrequent intervals, while a DTMF detection client may want small packets to reduce latency in interactive applications.

Queuing Requests in The Server

Most requests, such as halting a sound that is playing, are executed when received by the server. However, requests to the server are queued if they cannot be handled immediately. For example, while playing a sound file, any subsequent play requests from the client are queued until the previous requests are complete.

In some applications it is desirable to explicitly queue commands that would normally be handled immediately. The audio server allows the speed of a sound to be changed during playback (constant pitch time-compression), so, for example, a graphical slider can be used to interactively adjust playback speed. However, an application may want to play all of sound A, then play sound B at a different speed. Sequentially sending three requests to the server to (1) play A, (2) set the speed, and (3) play B would not have the desired effect, as the speed of sound A would change as soon as the second message arrived. The client could wait until receiving the done event from the first play request before sending the other messages, however, that would entail a round-trip between the server and client, possibly causing a discontinuity of the played sounds. To overcome this limitation, it is possible to explicitly turn command queuing on and off in the server. For example, the application can send commands to the server to (1) play A, (2) turn on queuing, (3) set the playback speed, and (4) play B. With this technique, the requests to change speed and play B are queued until the play A action is completed.

A similar approach to command queuing was taken in the VOX and DEC audio servers [1, 4]. This simple mechanism is sufficient for many applications, but is inadequate for others. Some applications may want to perform more sophisticated branching and logic in the server, based on termination conditions or user input. Rather than relying on this simple queuing mechanism, an audio server should ideally support an interpreted programming language (such as Tcl [15]) so that client defined procedures can be executed within the server. Such a programming and extension capability is provided by the Network extensible Window System (NeWS) [24].

Status Service

In addition to the primary play and record service provided by the audio server, a secondary status service is provided for debugging and logging purposes. The internal state of the server such as number of clients, current play/record status, error information, etc. is sent to all clients of the status service. This facility is more powerful than simply logging this information to a file for several reasons. First, the real-time operation of the server can be easily monitored simultaneously from a variety of locations. Second, when a status connection is established, the current internal state of the server is sent to the client in summary form—such information may be difficult to visually extract from a log file.

4.2 PC-Based Speech Recognition Server

An existing library interface to a PC-based speech recognizer was converted into an asynchronous server by adding several SM calls, and defining an RPC specification file. The library communicates to an external speaker-dependent isolated word speech recognizer by a serial interface [12]. During initialization of the server a file descriptor for the recognition hardware is registered with the SM.

Most of the server functions are accessed from the application by asynchronous messages. Once the application has requested that the server load a file of speech templates and has started recognition, the server asynchronously sends events to the application when a word is recognized. The events provided by the recognition server are more closely related to keyboard or mouse input than to those related to temporal media, such as provided by the audio server.

4.3 Software-Based Speech Recognition Server

A second speech recognition server with the same software interface has been built using a software-based recognizer [9] that runs entirely in a SPARCstation. This speaker-dependent, isolated word recognizer runs in real-time on the SPARC processor. The recognition server gets speech data from the audio server via data interest events, applies the recognition algorithm, and sends recognition events to interested clients.

A typical application using this recognition server requires a very distributed software architecture. Clients will often communicate with the audio server, the recognition server and a window server.

[10] And eventually, audio managers.

[11] TouchTones.

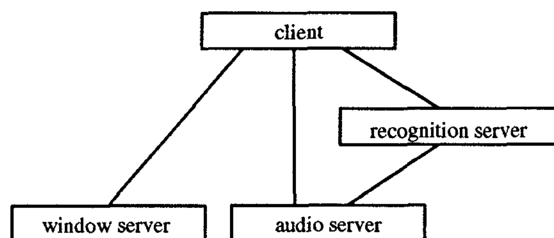

Figure 3: A client that uses audio, recognition, and window servers.

The recognition server, in turn, gets recorded audio data from the audio server. Such a configuration is shown in figure 3.

We are refining our application programming interface (API) to support recognizers with varying performance characteristics. We would like to support different classes of recognizers (speaker independent, continuous, etc.), as well as be able to handle recognizer-specific attributes such as differing training protocols.

5 APPLICATIONS

A variety of applications have been implemented by the Speech Research Group that use the audio and recognition servers. In addition to the two applications detailed in this section, several applications have been built that take advantage of multiple audio servers that execute in parallel on separate machines. We are in the process of migrating *all* of our speech and audio applications to use the SPARCstation-based audio server.

Such an environment can lead to a highly distributed configuration, allowing a variety of sound processing servers to be layered on top of the audio server (see figure 4). In addition to several applications and low-level servers, two "mid-level" servers that obtain data from the audio server are shown.

5.1 Capturing Background Speech

An application has been built using the audio server that continually records background audio into in-memory buffers for later playback. Both a graphical and a speech-only interface have been built to access the recorded audio information. The graphical interface uses the audio server, and an X Window Server for graphical output and mouse input [11]. The speech-only interface uses the recognition server for controlling the playback of the captured audio ([23] describes a similar system running in a Macintosh environment).

5.2 Voice Mail Message Collection

Voice mail in the Speech Research Group is gathered by a client application running on a Sun SPARCstation. This application communicates to a "phoneserver" process that monitors ISDN telephone activity. When a call is answered, messages are played and recorded using the audio server. Recording is controlled using real-time silence detection functionality provided by the audio server.

Information about the calling and called parties supplied by the phoneserver allows the messages to be delivered electronically. Recorded messages can be heard remotely over the telephone, or locally on a workstation using an interactive graphical interface [22]. This application and the audio server are an important part of the software infrastructure of the Speech Research Group—the voice mail application runs continuously, and answers dozens of telephone calls per day.

6 RELATED WORK

6.1 Hand-Built Servers

Several hand-crafted servers have been used in the Speech Research Group to provide information services to interactive applications (e.g., [13, 26]). Since our research environment contains diverse machine architectures, a least common denominator approach to network communications was usually taken: an ASCII string protocol was used on a predefined communication port. Servers were written to support either synchronous or asynchronous calls.

This arrangement had several drawbacks. First, the communications portions of the servers and clients evolved in an ad hoc, and unmaintainable, manner. Second, to access the server from a procedural interface required translating commands into properly formatted strings, and then parsing the output strings sent by the server—the serialization and deserialization of the data stream was coded by hand.

The ASCII telnet interface was found to be powerful enough to be required for all servers designed by the Speech Research Group. The tools described in this paper support both an ASCII interface and a procedural interface to a server, and provide high-level abstractions for establishing connections between clients and servers.

6.2 Remote Procedure Calls and Message Passing

Most remote procedure call systems built to date operate in a synchronous manner [8, 14]. One process sends a call message to the other process, and blocks until a reply is received—in such a model only one process is active at a given time. This master/slave relationship is sufficient for many kinds of distributed environments, but is not ideal for constructing interactive applications, or multi-modal user interfaces.

An RPC system is often used instead of asynchronous message passing because procedure calls are the primary control structure and data transfer mechanism in most programming languages. Message passing is better suited than RPCs for "dialog systems" where multiple conversing parties reside on different hosts, and in "mailing systems" where information is sent regardless of if it can be processed right away [21].

An RPC stub generator is a compiler that helps reduce development time by simplifying the coding and debugging of low-level routines [25]. RPC generators automate the process of serializing and deserializing data passed by a remote procedure call, often converting the data to into a machine independent format, and dispatching incoming calls to the appropriate routines.

The tools described in this paper allow the use of both RPC and message passing within a procedure call framework, and provide several data serialization formats.

6.3 X Window System and X Toolkit

Version 11 of the X Window System uses an asynchronous message passing protocol, however, for some requests the library acts like an RPC system and waits for an expected reply. The X Window System was intended to be completely independent of operating systems, network transport, and programming languages, thus instead of using an RPC generator, a protocol was defined for performance and portability reasons.

The X Toolkit provides a comprehensive event selection and dispatching mechanism to client applications [6]. Xt also allows applications to register additional devices and application specific timers to be handled in the main event loop. Unfortunately, these extensive

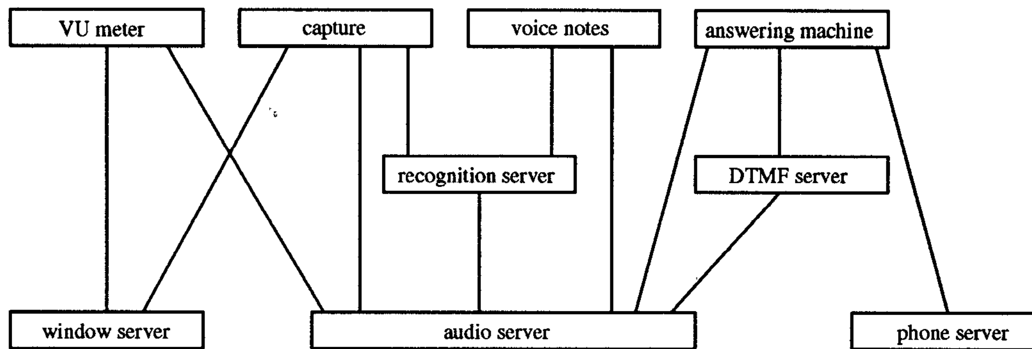

Figure 4: A multiple server environment.

facilities are only available to applications that use the toolkit and window system. Not all user interfaces require graphical displays or mice—some of the Speech Research Group's current work deals with hand-held and "speech-only interfaces" where a keyboard and display are not used [3, 23].

Early work on remote procedure call systems suggest that the procedure call was well understood by programmers, thus it was easy to adopt the RPC paradigm. The prevalence of the Xt mechanism for callback-based programming, now allows similar mechanisms to be used in a range of multimedia user interface applications.

Xt supports communication of window-related data between clients, but does not provide a general communication mechanism between clients and other types of servers. The tools described in this paper provide many of the low-level communication, event, and dispatching mechanisms provided by X and Xt without requiring the presence of a window system.

6.4 Previous Audio Servers

An early serial-controlled audio server developed in the Speech Research Group [18] relied on a hand-built client side library, and only supported a single client.

The remote procedure call compiler used in the VOX Audio Server [5] supported both synchronous RPCs and asynchronous messages, however, it was necessary to explicitly declare if the data was to be sent synchronously or asynchronously. VOX supported complex client-defined constructs in the server that represented audio routing configurations needed by the client. VOX, however, did not allow data to be exchanged between the server and a client.

The DEC audio server [1] was modeled closely on the X Window System server, and hence uses a protocol rather than an RPC generator. The server design was also based on the VOX Audio Server, and has complex server-side constructs. However, as a means of sharing resources, the DEC server mixes multiple sounds when they are played simultaneously.

The tools described in this paper use an RPC generator for rapid development, rather than hand-crafting code to support a protocol. The RPC specifications implicitly define whether calls should be synchronous or asynchronous. The Sparc-based server directly supports files, and also allows data buffers to be exchanged between the server and clients.

7 CONCLUSIONS

New user interface and software engineering paradigms are emerging because of the ubiquity of network-based services. The tools described in this paper provide a flexible and rapid prototyping environment for creating networked servers and applications. These tools allow servers to be built easily, while retaining the power and familiarity of remote procedure calls, and callback-style programming using asynchronous messages.

A simple specification file generates code for both synchronous RPCs and asynchronous messages. The procedure call generator, though currently limited to simple data types, supports several communication modes, including a telnet interface that allows direct interactive use of a server.

Future directions for the software include adding an interpreter into the server to reduce the traffic between the client and server, and to eliminate the queuing mechanism. Also under consideration is porting the system to use Apple Events under the Macintosh operating system.

It has recently been suggested [19] that RPC is evolving into the mainstream mode of communication in client-server models, and that it will be the key component in distributed applications and distributed operating systems. The tools described in this paper provide a different perspective that should be considered when designing future RPC systems, in particular, the usefulness of asynchronous messaging for multi-media applications and systems that manage temporal media [27].

These simple but powerful tools allow applications to easily incorporate new media and new interaction styles. Facilities such as these promote the development of new applications and user interface techniques, especially those involving temporal media such as speech, audio, and video.

8 ACKNOWLEDGMENTS

Chris Schmandt participated in early design discussions of the SM and BSM architecture. Ralph Swick provided information about X and Xt internals. Chris Schmandt, Lisa Stifelman, and Wayne Yamamoto reviewed earlier drafts of this paper. Jordan Slott is assisting in the development of the audio server.

This work was sponsored by Sun Microsystems and Apple Computer.

REFERENCES

[1] S. Angebranndt, R. L. Hyde, D. H. Luoung, D. Siravara, and C. Schmandt. Integrating audio and telephony in a distributed workstation environment. In *Proceedings of USENIX Summer '91*, pages 419–435. USENIX, 1991.

[2] B. Arons. The design of audio servers and toolkits for supporting speech in the user interface. *Journal of the American*

Voice I/O Society, 9:27–41, Mar. 1991.

[3] B. Arons. Hyperspeech: Navigating in speech-only hypermedia. In *Hypertext '91*, pages 133–146. ACM, 1991.

[4] B. Arons, C. Binding, K. Lantz, and C. Schmandt. A voice and audio server for multimedia workstations. In *Proceedings of Speech Tech '89*, pages 86–89, May 1989.

[5] B. Arons, W. Yamamoto, J. D. Northcutt, C. Binding, K. Lantz, and C. Schmandt. The VOX audio server, version 1.0. Technical report, Olivetti Research Center, Aug. 1988.

[6] P. J. Asente and R. R. Swick. *X Window System Toolkit*. Digital Press, 1990.

[7] C. Binding, C. Schmandt, K. Lantz, and B. Arons. Workstation audio and window-based graphics: Similarities and differences. In *Engineering for Human-Computer Interaction*. IFIP Working Group 2.7, Aug. 1989.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[9] G. Cockroft. Personal communication. Agog, Inc. (517-627-2186, greg@agog.com).

[10] D. E. Comer. *Internetworking with TCP/IP*, volume 1. Prentice Hall, Engelwood Cliffs, NJ, 1991.

[11] D. Hindus and C. Schmandt. Ubiquitous audio: Capturing spontaneous collaboration. In *CSCW '92*. ACM, 1992.

[12] S. Manandhar. Speech interface to a workstation window manager. M.I.T. Bachelors Thesis, Media Arts and Sciences Section, May 1989.

[13] S. Manandhar. Activity Server: You can run but you can't hide. In *Proceedings of the Summer 1991 USENIX Conference*. Usenix, 1991.

[14] C. Manson and K. Thurber. Remote control. *Byte*, pages 235–239, July 1989.

[15] J. K. Ousterhout. Tcl: An embeddable command language. In *Winter USENIX Conference Proceedings*, 1990.

[16] J. Postel and J. Reynolds. Telnet protocol specification, May 1983. Internet RFC854 (available by ftp from nic.ddn.mil).

[17] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, Apr. 1986.

[18] C. Schmandt and M. McKenna. An audio and telephone server for multi-media workstations. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 150–160. IEEE Computer Society, Mar. 1988.

[19] A. Sinha. Client-server computing. *Communications of the ACM*, 35(7):77–98, June 1992.

[20] Speech Research Group, MIT Media Lab. *Sound Library and Sound Server Documentation*, 1992.

[21] J. A. Stankovic. Software communication mechanisms: Procedure calls versus messages. *Computer*, pages 19–25, Apr. 1982.

[22] L. J. Stifelman. Not just another voice mail system. In *Proceedings of 1991 Conference*, pages 21–26. American Voice I/O Society, 1991.

[23] L. J. Stifelman. VoiceNotes: An application for a voice controlled hand-held computer. M.I.T. Masters Thesis, Media Arts and Sciences Section, May 1992.

[24] Sun Microsystems Inc., Mountain View, California. *NeWS: Technical Overview*, 1987.

[25] Sun Microsystems Inc., Mountain View, California. *Network Programming Guide*, 1990.

[26] C. C. Wong. Personal communications. M.I.T. Masters Thesis, Media Arts and Sciences Section, June 1991.

[27] W. K. Yamamoto and S. Stanton. Symmetric, asynchronous protocols for supporting workstation integrated teleservices. Submitted to Winter USENIX, 1993.