Ethercom: A study of audio processes and synchronization

Richard Rybacki Kay A. Robbins Steven Robbins
Division of Mathematics, Computer Science, and Statistics
The University of Texas at San Antonio

Abstract — Ethercom is a project which introduces many aspects of network communication, process synchronization, and scheduling at a level suitable for an undergraduate operating systems or networks course. The project requires a Unix workstation equipped with a microphone and speaker. The project is developed in stages starting from simple I/O to the audio device. The final development is two-way communication over a network using lightweight processes and asynchronous I/O. When they have completed the project, students can sit at their workstations and converse with counterparts at remote sites. Variations and enhancements for the basic project are suggested including a monitor implementation of communication. Libraries of simplified routines for I/O and network communication are also available via anonymous ftp.

1 Introduction

Continuous media applications (e.g. audio and video) are becoming feasible as network bandwidth and workstation power increase. The inclusion of an audio device on many popular workstations provides an opportunity for the development of a variety of interesting projects in operating systems and networks courses. These projects are not only of current interest, but they allow students to explore a range of concepts which are important to their understanding of the modern systems environment. The projects which are proposed in the following discussion explore audio technology, network technology, process scheduling, signal handling, synchronization and lightweight processes.

The basic project consists of the development of an Ethernet intercom system (Ethercom). When they have completed the final stage of the project, students are able to converse with counterparts at remote sites. The approach we describe allows the students to experiment with different system calls without writing large amounts of code. The project is realistic enough to engender enthusiasm, and the audio aspect of the project provides instant feedback.

2 Project Overview

The goal of the project is to develop a system for full duplex voice communication over a network. The project is developed in three stages. Each stage covers some basic concepts. Additional topics and enhancements are also suggested.

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.
ACM-24thCSE-2/93-IN,USA
© 1993 ACM 0-89791-566-6/93/0002/0218...$1.50

1. Reading from and writing to the audio device. (Basic concepts: devices and files, I/O control, read and write. Additional topics: filtering, thresholding, adjusting of volume.)
2. Sending audio over the network. (Basic concepts: network communication, client-server model, Additional topics: out-of-band control, TCP and UDP protocols, buffering.)
3. Developing an audio system for two-way communication. (Basic concepts: non-blocking I/O, signal handling. Additional topics: lightweight processes, process scheduling, monitors, producer-consumer synchronization.)

3 System Information

The project will be described specifically for Sun systems, however many other Unix systems (such as Next and SGI) have similar facilities. The programs were written for Sun SparcStation and SLC workstations running SunOS version 4.1.2. These systems are equipped with an AM79C30A digital subscriber controller chip which performs digital to analog and analog to digital conversion. The signal is sampled 8000 times per second with 12-bit precision. The samples are then compressed to 8-bits using a u-law encoding scheme. Inexpensive general purpose microphones (Radio Shack #33-1067) were purchased to provide the audio input.

4 Audio I/O

In the first assignment, students are asked to read from and write to the audio device. A discussion of files and devices fits very nicely into this assignment. A library of routines to access the audio device is shown in Figure 1. The audio device is represented by the file descriptor `fd`. It is considered to be an object which has the operations open, close, read, and write. The trade-off between uniform treatment of devices and access to device-specific control can also be illustrated. Only one process can open the audio device at a time, and the open in `open_audio_device` will hang if the device has already been opened unless the `O_NDELAY` flag is used. (The `ioctl` is called with `I_SETSIG` in order to cause a `SIGPOLL` signal to be generated when the device represented by `fd` is ready for input. It is used later when lightweight process library is introduced and is not necessary for basic communication.)

A first program to access the audio device is given in Figure 2. The program repeatedly reads a block of up to 1024 bytes of data and echoes it to the speaker. The performance is somewhat sensitive to the size of the blocks (`AUDIO_BUFSIZE`) that are read. For the Sun system, 1024

bytes seems to be the smallest chunk transferred from the audio device to the kernel. Delay seems to be minimized with this selection of block size. Most students picked a much larger block size in their initial experiments and ran into timing problems when trying to maintain continuous speech during transfer across the network. The timing problems were interesting in their own right, and many experiments on the effects of buffering and network protocols can be performed.

One of the difficulties which immediately becomes apparent with the microphone assignment is that once opened, the audio device is sampled until it is closed. This continuous sampling produces a prohibitive amount of data for transmission across the network. A filter should be provided to throw away audio packets which contain no voice. A simple method of filtering is to convert the u-law data to linear scale and reject packets which fall below a threshold.

Possible enhancements to the program at this point include providing a calibration routine which allows the threshold for voice detection to be adjusted based on the current value of the ambient room noise. More sophisticated filtering algorithms can also be explored. Another enhancement is to provide a window which dynamically displays the number of packets per second which have voice data. Students can also add signal handling to the simple routine in Figure 2 so that a summary of packets read and packets transmitted can be displayed when the program is interrupted. An audio tool is provided with the Sun operating system, and one can easily add code to fork and exec an audio tool to control the volume.

```
#include <stdio.h>
#include <fcntl.h>
#include <stropts.h>
#define DEVICE "/dev/audio"
static int fd = -1;          /* audio file descriptor */

int open_audio_device()
{            /* Open audio.  Return 0 if successful */
   if ((fd = open(DEVICE, O_NDELAY | O_RDWR)) < 0)
      return(-1);
                     /* used later to enable SIGPOLL */
   if (ioctl(fd, I_SETSIG, S_INPUT) < 0)
      return(-1);
   return(0);
}

void close_audio_device()
{                          /* Close the audio device. */
   close(fd);
}

int write_audio_device(buffer, length)
char *buffer;
int  length;
{        /* Write length bytes of buffer to audio. */
   return(write(fd, buffer, length));
}

int read_audio_device(buffer, maxcnt)
char   *buffer;
int    maxcnt;
{/* Read up to maxcnt bytes from audio into buffer. */
   return(read(fd, buffer, maxcnt));
}
```

Figure 1: The audio device object and its basic operations.

```
#include <stdio.h>
#define ETHERCOM_PORT     31270
#define AUDIO_BUFSIZE      1024
main()
{/* Read from microphone and echo to speaker
                 until interrupted or error. */
   unsigned char buffer[AUDIO_BUFSIZE];
   int bytes;

   if (open_audio_device() < 0) {
      perror("open_audio_device");
      exit(1);
   }
   while (1) {
      if ((bytes = read_audio_device
                (buffer, sizeof(buffer))) < 0) {
         perror("read_audio_device");
         exit(1);
      }
      if (write_audio_device(buffer, bytes) < 0) {
         perror("write_audio_device");
         exit(1);
      }
   }
}
```

Figure 2: Basic program to access the audio device.

5 Network Transmission

The second stage of the project is the establishment of one-way communication over a network (in our case Ethernet) using either sockets or TLI (transport layer interface). Communication follows a client-server model. A server program runs on a remote host, and a client program on the local machine seeks to establish communication. Communication takes place through ports which are associated with and accessed by ordinary file descriptors. The server opens a file descriptor and listens for connection requests on port number A which has been agreed upon in advance as shown in Figure 3. The client requests a connection to the server through port A. When the connection is made, the server transfers the client to another port for the session so that the server can continue to monitor port A for other calls as shown in Figure 4. This is analogous to operator assisted telephones at hotel switchboards. A patron calls the operator and requests to be connected to a certain room. The operator makes the connection (with patch cords in the old days) to the requested hotel room and steps out of the conversation.

One of the difficulties with assigning a project involving network communication in an undergraduate operating systems class is that there are a lot of details which are peripheral to the main content of the course. For this reason, we have provided a library of simplified routines which we call UICI (Universal Internet Communication Interface). We have implemented these routines (u_open, u_listen, u_connect, u_close, u_read, u_write, and u_error) using sockets and using TLI. The details of the implementation can be discussed in the classroom, and students in a networks course can be required to replace UICI with their own routines.

A client and server which do one way communication using these routines are shown in Figures 5 and 6. The client routine in Figure 5 is relatively simple. Once a connection has been established, the client simply reads from the audio

device and transmits the packet across the network. The client program is called with a command line argument specifying the name of the host machine on which the server is running. The server in Figure 6 opens well-known port ETHERCOM_PORT and listens for connections through file descriptor fd. When a connection is established, communication will take place through the port represented by file descriptor newfd. When either the client or the server is interrupted, the network connection is broken. The other participant detects that a network read or write has failed and terminates.

In a normal client-server program, the server would fork a child to handle the communication. The server could then resume listening for other connection requests. However, since only one process can open the audio device at a time, the server performs the communication directly.

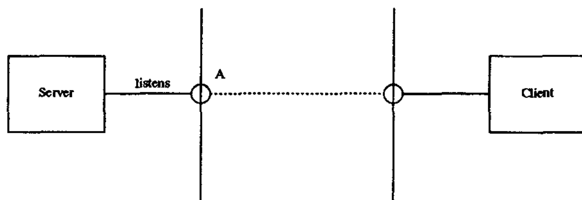

Figure 3: Client server model — connection request

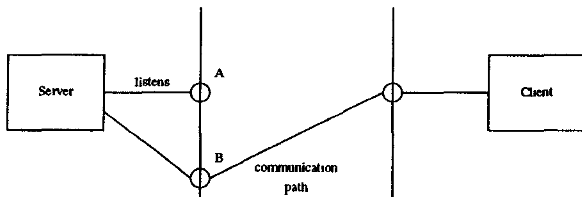

Figure 4: Client server model — connection established

6 Two-way Communication

The core of the project is the design of a two-way communication system. The program should monitor an audio input device and an input channel from a remote connection. If a packet comes in from the audio device, it should be filtered. If the packet data exceeds a voice transmission threshold, the packet should be transmitted to the remote socket. If a packet comes from the remote connection, it should be written to the audio output device.

The simplest approach is to use the Unix select or to use polling with nonblocking I/O. However, the implementation discussed here uses the lightweight process library (LWP) provided with SunOS 4.1.2. We define a lightweight process as an independent thread of control within an address space. Lightweight processes provide a transparent way of switching among paths of control without the overhead of context switching. We made the LWP choice because of the increasing move towards operating systems with fully-preemptive kernels running on multiprocessor systems. Most future operating systems will probably provide kernel support for lightweight processes along with facilities for user-scheduling of threads and real-time capabilities necessary for the effective handling of continuous media applications[1].

```
#include <stdio.h>
#define ETHERCOM_PORT     31270
#define AUDIO_BUFSIZE      1024
main(argc, argv)
int argc;
char **argv;
{/* Read from audio device and send to remote server. */
   int fd, bytes;
   unsigned char buffer[AUDIO_BUFSIZE];
   if (argc != 2) {/* server host is command line arg */
      fprintf(stderr, "Usage: %s <host>\n", argv[0]);
      exit(1);
   }
      /* request a connection with the remote server */
   if ((fd = u_connect(ETHERCOM_PORT, argv[1])) < 0) {
      u_error("u_connect");
      exit(1);
   }
                              /* get ready to send audio */
   if (open_audio_device() < 0) {
      perror("open_audio_device");
      exit(1);
   }
   while (1) {
      if ((bytes = read_audio_device
               (buffer, AUDIO_BUFSIZE)) <= 0) {
         perror("read_audio_device");
         exit(1);
      }
                         /* send audio to remote server */
      if (u_write(fd, buffer, bytes) <= 0) {
         u_error("u_write");
         exit(1);
      }
   }
}
```

Figure 5: Audio device reader client for one-way network communication using UICI.

```
#include <stdio.h>
#define ETHERCOM_PORT     31270
#define AUDIO_BUFSIZE      1024
main()
{       /* Receive from network and write to speaker. */
   int fd, newfd, bytes;
   unsigned char buffer[AUDIO_BUFSIZE];
         /* open the well-known port for listening */
   if ((fd = u_open(ETHERCOM_PORT)) < 0) {
      u_error("u_open");
      exit(1);
   }
                              /* listen for a connection */
   if ((newfd = u_listen(fd)) < 0) {
      u_error("u_listen");
      exit(1);
   }
   if (open_audio_device() < 0) {
      perror("open_audio_device");
      exit(1);
   }
   while (1) {
      if ((bytes =
           u_read(newfd, buffer, AUDIO_BUFSIZE)) <= 0) {
           u_error("u_read");
           exit(1);
      }
      if (write_audio_device(buffer, bytes) <= 0) {
           perror("write_audio_device");
           exit(1);
      }
   }
}
```

Figure 6: Audio device writer server for one-way network communication using UICI.

```
#include <lwp/lwp.h>
#include <stdio.h>
#define AUDIO_BUFSIZE       1024
#define THREAD_STACK_SIZE   1000
static thread_t reader_t;
static thread_t writer_t;
static int socket_fd;

static void shutdown()
{ /* Close devices and shutdown processes. */
   close(socket_fd);
   close_audio_device();
   lwp_destroy(reader_t);
   lwp_destroy(writer_t);
}
void reader()
{ /* /dev/audio -> network (fd)  until error.  */
   static char buffer[AUDIO_BUFSIZE];
   int bytes;

   while (1)  {
      if ((bytes = read_audio_device
                 (buffer, AUDIO_BUFSIZE)) <= 0) {
         perror("read_audio_device");
         shutdown();
      }
      if (u_write(socket_fd, buffer, bytes) <= 0)   {
         u_error("u_write");
         shutdown();
      }
   }
}
void writer()
{/* socket -> /dev/audio until error. */
    int bytes;
    static char buffer[AUDIO_BUFSIZE];

    while (1)  {
       if ((bytes = u_read
          (socket_fd, buffer, AUDIO_BUFSIZE)) <= 0) {
          u_error("u_read");
          shutdown();
       }
       if (write_audio_device(buffer, bytes) <= 0)  {
          perror("write_audio_device");
          shutdown();
       }
    }
}
int threads_init(fd)
int fd;
{/* Initialize the reader and writer LWP's. */

   lwp_setstkcache(THREAD_STACK_SIZE, 2);
   socket_fd = fd;
   lwp_create(&reader_t,reader,
              MINPRIO,0,lwp_newstk(),0);
   lwp_create(&writer_t,writer,
              MINPRIO,0,lwp_newstk(),0);
}
```

Figure 7: reader reads from the audio device and outputs to the network. writer reads from the network and writes to the audio device.

Two lightweight processes will be introduced—reader and writer. The reader routine reads from the audio device and outputs to the network, while the writer routine reads from the network and writes to the audio device. The reader and writer lightweight processes are shown in Figure 7. They are activated by the threads_init routine. Active threads attached to the reader and writer routines are created by calls to lwp_create. Once a thread is created, it is referenced by a thread id which has type thread_t. The **lwp_setstkcache** routine is called to allocate two stacks of 1000 bytes each. The **lwp_newstk()** gets a free stack when the **lwp_create** call creates a lightweight processes. These stacks are returned to the stack pool when the lightweight processes are destroyed. On the first call to the lightweight process library, the main program becomes a thread running at highest priority. The threads that it creates are blocked until that thread exits or lowers its priority.

Server and client programs implemented with lightweight processes are shown in Figures 8 and 9 respectively. The client and server are symmetric in the sense that they both run reader and writer lightweight processes. The reader lightweight process of the client is communicating with the writer lightweight process of the server.

```
#include <stdio.h>
#define ETHERCOM_PORT     31270

main()
{/* Server for two-way commun. using LWP  */
    int fd;
    int newfd;

    if ((fd = u_open(ETHERCOM_PORT)) < 0)   {
        u_error("u_open");
        exit(1);
    }
    if ((newfd = u_listen(fd)) < 0)  {
       u_error("u_listen");
       exit(1);
    }
    if (open_audio_device() < 0)  {
       perror("open_audio_device");
       exit(1);
    }
    threads_init(newfd);
}
```

Figure 8: Server for two-way communication using LWPs.

```
#include <stdio.h>
#define ETHERCOM_PORT     31270

main(argc, argv)
int argc;
char **argv;
{/* Client for two-way commun. using LWP */
   int fd;

   if (argc != 2)  {
      fprintf(stderr, "Usage: %s <host>\n", argv[0]);
      exit(1);
   }
   if ((fd = u_connect(ETHERCOM_PORT, argv[1])) < 0)  {
       u_error("u_connect");
       exit(1);
   }
   if (open_audio_device() < 0)  {
      perror("open_audio_device");
      exit(1);
   }
   threads_init(fd);
}
```

Figure 9: Client for two-way communication using LWPs.

The lightweight process implementation of two-way communication illustrates the elegance and simplicity of implementing parallel processes using threads. There are many possible extensions to this part of the project. One extension is to implement the communication of the reader as two threads. One thread reads the audio device and deposits a block in a ring buffer. A second thread removes a block from the ring buffer, tests it for the presence of voice and forwards it on the network if appropriate.

Producer-consumer and synchronization implementations are typically difficult to debug, and the audio aspect of this assignment provides an audible indication of the effectiveness of the algorithm. A deadlock free and efficient algorithm is necessary for smooth and continuous speech. In addition, the thread library provides a mechanism for the user to write a thread scheduler. The student can experiment with different scheduling constructs and *hear* the effectiveness of the scheduling discipline selected. When ordinary semaphores are used, students with buggy programs can fill up the system semaphore table causing all other processes which use semaphores to block. If the students don't clean up, their semaphores can persist until a reboot. The LWP library primitives are implemented at the user level, so they only persist as long as the program does—a definite advantage in teaching synchronization to inexperienced students.

7 Discussion

The Ethernet intercom project was initially given to a graduate operating system class with instructions to implement the basic system and then to enhance it in some way. Students pursued many different aspects of the project. One student did extensive voice analysis and developed sophisticated threshold and calibration techniques. Another student experimented with conference calling and combining incoming audio signals from several remote sources. Several students installed their intercom daemons on the machines of friends at other universities and tried remote communication.

The first undergraduate class to work on Ethercom implemented the basic two-way communication using `select`. The class then designed (in class) and implemented a conference calling system using lightweight processes. It was possible to introduce several advanced topics on group communication in the discussion and subsequent design process.

A number of problems were encountered in the first implementation of the Ethercom project. The Sun TLI interface does not seem to be completely debugged. Students who were not careful about the way that TLI calls were interrupted by signal handlers could cause the remote machine to reboot. The buffer size for reads and writes seemed to be critical for performance, and the lack of a true real-time facility for scheduling on the current SunOS (4.1.2) made performance at busy times less than perfect. The filtering and culling of packets not containing voice data was important to the overall performance of the network. The Ethernet in the section of the lab in which students were testing their audio projects went down several times during the course of the project.

We also encountered some problems when using the LWP library. Non-blocking I/O didn't seem to work with the lightweight processes unless the `SIGPOLL` was enabled for the audio device. Hence in Figure 1 we had to do an `ioctl` with the `I_SETSIG` argument. We also could not get the LWP library to work with our TLI version of UICI. We view these as short term problems since there will be kernel support for threads and lightweight processes in the next version of SunOS [2, 4].

In addition, a good mechanism for allowing the receiver to reject or accept calls should be incorporated in order to remove the audio nuisance factor since the person sitting at the workstation may not be a class member. Other enhancements include a busy signal when a client calls a server which is busy, or a call waiting mechanism which allows the server to switch calls. The `UICI` routines used the TCP protocol. We found that the sockets needed to be created with the `TCP_NODELAY` option or acknowledgments would be delayed causing speech to be discontinuous [3]. This detail can be hidden within `UICI` but is interesting from a network point of view.

Overall the Ethercom seems to be an effective project for introducing many important operating systems concepts without demanding an excessive programming effort. The students who were working on the audio project seemed to enjoy it. The project introduces concepts in communication, synchronization, signal handling and scheduling. It is satisfying because of the immediate feedback provided by the audio, and the simplicity of the I/O.

All of the programs and a makefile for compilation are available by anonymous ftp from `ringer.cs.utsa.edu` in the directory `/pub/sigcse92`.

Acknowledgments:

We would like to thank the following students for their work on Ethercom: L. Bishop, G. Butchee, R. Castaneda, S. Dykes, E. Grossenbacher, K. He, S. Kulkarni, C. Michaels, V. Randal, A. Shah and H. Yang. This work was supported by the National Science Foundation ILI Program, Grant USE-0950407.

References

[1] R. Govindan and D. P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media," *Proc 13th ACM Symposium on Operating Systems Principles*, (1991) pp. 68–80.

[2] S. Khanna, M. Sebree, J. Zolnowsky, "Realtime Scheduling in SunOS 5.0," *SunSoft Incorporated, preprint*, 1992.

[3] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of 4.3BSD Unix Operating System*, Addison Wesley, 1989.

[4] "SunOS 5.0 Multithread Architecture," *Sun Microsystems White Paper* (1991).

[5] W. R. Stevens, *Unix Network Programming*, Prentice Hall, 1990.

[6] D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System," *ACM Transactions on Computer Systems* 6 (1988) pp. 3–27.