Using Remote Logging for Teaching Concurrency

Steven Robbins
Department of Computer Science
University of Texas at San Antonio
srobbins@cs.utsa.edu

Abstract

Students often have difficulty visualizing, understanding and debugging concurrent programs. Programming assignments involving concurrency are also difficult to grade. The output alone is not sufficient because the print statements from cooperating threads or processes can interfere with each other and garble the results. The remote logging tool described here allows multiple processes or threads to atomically log messages to a local or remote host. Different views of the messages are available in real time through a graphical user interface (GUI). The tool consists of two parts, a library for adding logging commands to a user program and a GUI for presenting different views of the logged messages. A separate logging library is needed for each programming language (e.g., C, C++, Java), but a single GUI works with all of these. A C logging library and a general GUI are available on the web.

Categories & Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education - Computer Science Education.

General Terms

Concurrent Programs, Debugging Tools.

Keywords

Atomic, Logging, GUI.

1 Introduction

Concurrency can be introduced in many computer science courses including systems programming, operating systems and computer networks. Aids for the teaching of concurrency that have been recently reported include communication libraries [3], tools [4], Petri nets [2], instructional operating systems [1, 5], simulators [7] and process topology [6].

The tool described here is a remote logging facility that allows a program to log messages at various points in a concurrent program. The messages are sent to a GUI-based logging engine that allows the user to display different views of the messages along with information about the source and the time they were received. The GUI allows users to create printable log files that can be handed in to an instructor to aid in grading an assignment. The next section of the paper gives an overview of the logger design, followed by sections on the local interface and the logging engine. Sections 5 through 8 give examples of using the logger along with sample output. Section 9 talks about using the implementation of the logger as a teaching tool, while Section 10 gives conclusions.

2 Design

The logging facility has two parts, a local interface that is compiled and linked with the program doing the logging, and a logging engine with a graphical user interface (GUI) that receives the logging information and displays it in a user-modifiable format. The GUI may be run on a host other than the one running the logged program and is independent of the programming language of the logged program. This separation of local interface and logging engine provides flexibility when the logged host does not support a GUI or when the programs to be logged are written in different languages. It also removes the CPU-intensive GUI processing from the logged host, minimizing the interference with the program being logged, and it allows processes on multiple hosts to be logged.

The logging engine GUI allows students to save the output in HTML format and display it on their web page. Java was chosen for the language of the remote logger because its graphical library is almost universally available and the same code can be run on multiple machines. The code is distributed in jar files and can be run on any machine with a Java 1.1 or later runtime system. All code is available over the Internet.

A separate local interface is needed for each language and operating system type. The local interface to the logger for the C language under UNIX described here is based on the C I/O library, which makes learning to use the logging facility fairly simple. The logger works seamlessly in either a threaded or non-threaded environment. It can be used with programs written in C or C++. A different local interface would be needed for other languages such as Java.

3 Local Interface (C Implementation)

The logging facility uses a handle of type `LFILE` with functions `lopen`, `lclose` and `lprintf` that are analogous to the C I/O functions `fopen`, `fclose` and `fprintf`. The prototypes are:

SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

```
LFILE *lopen(char *host, int port);
int lclose(LFILE *mf);
int lprintf(LFILE *mf, char *fmt, ...);
```

The `lopen` takes two parameters, the name of the host that receives the logging information and the port number used to connect to that host. If `host` is NULL or the port is 0, environment variables are used for these values. The `lclose` and `lprintf` have syntax identical to `fclose` and `fprintf`, except that they return 0 on success and -1 on error. The `fmt` string in `lprintf` behaves exactly like the format string in `fprintf`, except that it supports one additional format specification, `%t`, for including the current time in the logged string. The design ensures that all logging is atomic.

The local interface consists of four files. Two files are for UICI [8] which handles the network communication. A header file contains a typedef for the `LFILE` structure and the prototypes for the interface routines given above. The last file contains the code. Each `lopen` makes a network connection to the remote log receiver. This connection stays open until all processes that have inherited it have closed the connection with `lclose` or have terminated.

The `lprintf` creates a string and sends it to the remote logger along with the process ID of the calling process. In a threaded environment, `lprintf` also sends the thread ID. The `lprintf` function does not do any buffering at the process and each call to `lprintf` sends its string atomically.

4 Remote Logging Engine (Java Implementation)

The remote logging engine is a network server that waits for connections on a given port. The server will accept any number of connections on this port and log information coming from all connections. Each `lopen` from the local interface starts a new connection with a new connection number. When the server starts, it displays a control window that allows access to the three main views: Output, Connections and Generators. It also allows the information shown in any of these views to be put is a log file [9].

The **Output Frame** in Figure 1 shows all of the output sent to the remote logger ordered by arrival time. For each string logged, it gives the message number, the connection number, the generator, the time the message was received and the message. The generator is the process ID of the process that executed the `lprintf`. In a multithreading environment this is followed by a period and the thread ID of the calling thread as shown in Figure 2. By default, times are measured in seconds since the first connection was made. Three decimal places are shown, giving millisecond precision, the precision available in the base Java system. Time can also be displayed as wall clock time.

The **Connections Frame** shows information about each connection. A connection is made from the C interface with the `lopen` function. Connections are inherited by forked processes and threads, so many processes and threads can use the same connection. Each connection is given a connection number by the remote logger starting with connection 0. For each connection, the **Connections Frame** shows the connection number, the time the connection was opened (from the point of view of the GUI logging engine), the number of messages sent over that connection, the time the connection was closed, and the number of seconds that the connection was open.

Clicking on a line in the **Connections Frame** pops up a **Connection Frame** window giving detailed information about a particular connection. This gives similar information to that displayed in the **Output Frame**, but only for the messages from a given connection.

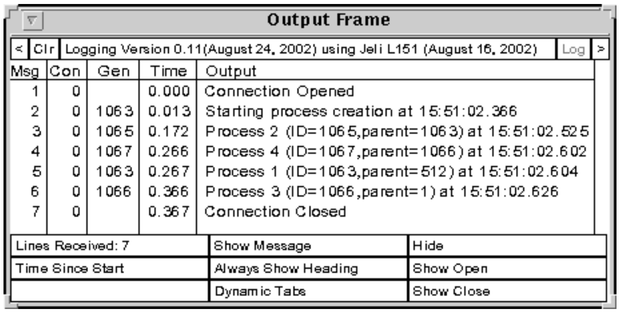
The **Generators Frame** shows one line of information for each generator. A generator is determined by the process ID (or process ID and thread ID in a multithreaded environment). A single process can have more than one connection to the logging engine and so each generator line may correspond to a number of lines in the **Connections Frame**. Clicking on a line in the **Generators Frame** pops up a **Generator Frame** showing only those messages having that generator.

5 Example: Process Chain

In a chain of `n` processes, each process except the last has another of the processes as its parent. Process chains are useful in teaching about inheritance of resources and concurrent operation. The order in which the processes in a chain execute is nondeterministic, affected by the scheduling algorithm and the other processes on the system. When I first started using the process chain as an example, I would have each process of the chain output a message. The messages would appear in a different order on subsequent runs. On faster, modern systems, the output of this type of program is almost always the same and it may take hundreds of runs before the nondeterministic behavior is apparent. For testing, each process can call `waste_time` to simulate doing a random amount of CPU bound processing before printing the message. This makes the nondeterministic nature more apparent.

The program below creates a chain of 4 processes and logs a message from each process. Each process calls `waste_time(n)`, which wastes a random number of microseconds of CPU time between 1 and n. Figure 1 shows the **Output Frame** for this program. For each run, the messages may appear in a different order.

```
int main(void) {
   int   i;
   LFILE *mf;

   mf = lopen(NULL,0);
   lprintf(mf,"Starting process creation at %t");
   for (i = 1; i < 4;  i++)
      if (fork() != 0)
         break;
   waste_time(100000);
   lprintf(mf, "Process %d (ID=%d, parent=%d) at %t",
           i, (int)getpid(), (int)getppid());
   return 0;
}
```



Logging Version 0.11(August 24, 2002) using Jeli L151 (August 16, 2002)

Msg	Con	Gen	Time	Output
1	0		0.000	Connection Opened
2	0	1063	0.013	Starting process creation at 15:51:02.366
3	0	1065	0.172	Process 2 (ID=1065,parent=1063) at 15:51:02.525
4	0	1067	0.266	Process 4 (ID=1067,parent=1066) at 15:51:02.602
5	0	1063	0.267	Process 1 (ID=1063,parent=512) at 15:51:02.604
6	0	1066	0.366	Process 3 (ID=1066,parent=1) at 15:51:02.626
7	0		0.367	Connection Closed

Lines Received: 7

Time Since Start

Figure 1: Output from a chain of 4 processes.

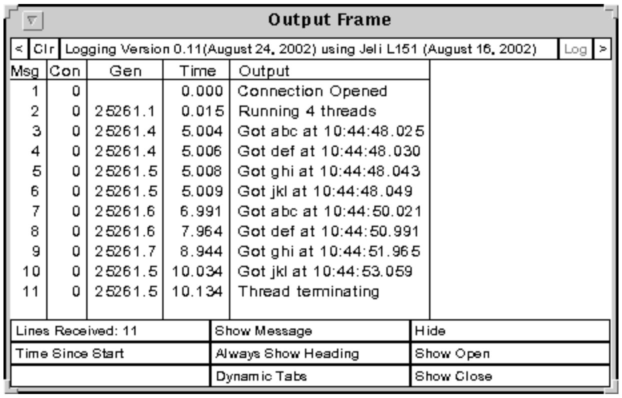
6 Example: Threads

The following program creates 4 threads that each monitor standard input. After getting input each thread does a short calculation of random length to simulate the processing of the input.

```
static void *do_it(void *mf) {
   char buf[80];
   int i;

   for (i = 0; i < 3; i++) {
      scanf("%79s", buf);
      lprintf((LFILE *)mf, "Got %s", buf);
   }
   lprintf((LFILE *)mf, "Thread terminating");
   waste_time(100000);
   return NULL;
}

int main(void) {
   int i;
   pthread_t tid[4];
   LFILE *mf;

   mf = lopen(NULL, 0);
   lprintf(mf, "Running 4 threads");
   for (i = 0; i < 4; i++)
      pthread_create(tid+i, NULL, do_it, mf);
   for (i = 0; i < 4; i++)
      pthread_join(tid[i], NULL);
   lprintf(mf, "Main program exit");
   return 0;
}
```

Figure 2 shows the **Output Frame** after the line “abc def ghi jkl” was typed at the keyboard followed by “abc”, “def”, “ghi”, and “jkl” on separate lines. The **Output Frame** shows the process ID and the thread ID in the **Gen** column. In this run of the program the threads that receive the input are 4, 4, 5, 5, 6, 6, 7 and 5. The result may be different on each run.

This simple example illustrates the asynchronous behavior of multiple threads. It is not possible to predict which thread receives each input.



Logging Version 0.11(August 24, 2002) using Jeli L151 (August 16, 2002)

Msg	Con	Gen	Time	Output
1	0		0.000	Connection Opened
2	0	25261.1	0.015	Running 4 threads
3	0	25261.4	5.004	Got abc at 10:44:48.025
4	0	25261.4	5.006	Got def at 10:44:48.030
5	0	25261.5	5.008	Got ghi at 10:44:48.043
6	0	25261.5	5.009	Got jkl at 10:44:48.049
7	0	25261.6	6.991	Got abc at 10:44:50.021
8	0	25261.6	7.964	Got def at 10:44:50.991
9	0	25261.7	8.944	Got ghi at 10:44:51.965
10	0	25261.5	10.034	Got jkl at 10:44:53.059
11	0	25261.5	10.134	Thread terminating

Lines Received: 11

Figure 2: Output from a program with 4 threads.

7 Example: Request-Reply Protocol

In a request-reply protocol, one host sends a request to another, and the second host sends a reply message back to the first host. Consider a time server that sends the current time as a string each time it receives a byte from a network connection. The time server waits for a connection request. Each time the server reads a byte it sends the current time to the remote host in the form of a string. When the connection is closed, the server waits for another connection request. Figures 3 and 4 show the result of logging this request-reply protocol by the client and the server under different circumstances. In each case the client makes two requests. This example shows some of the strengths and weaknesses of the logger.

In Figure 3 the server, client, and logging host are the same machine. The **Time** column in the **Output Frame** indicates the time the message was received. Normally, this is not related to any times generated by the hosts doing the logging, since the clocks on different hosts are not synchronized. The default is to display the time the message was received relative to the time of the first connection to the logger. However, in this case everything is running on the same machine, so we show the absolute time at which the messages are received. The messages with connection number 0 or generator 5739 come from the time server. Messages with connection number 1 or generator 5741 come from the client.

The order that the messages from a given host are received is the order in which they are sent. However, messages from the server and client are interleaved in a way that does not represent the order in which they were sent. For example, message 7 was generated by the server at about 788 milliseconds after 2:33:26 pm and logged 89 ms after it was sent. Message 10 was sent by the client 1 ms before message 7 was sent by the server, but message 10 was processed by the logging engine 12 ms after message 7.

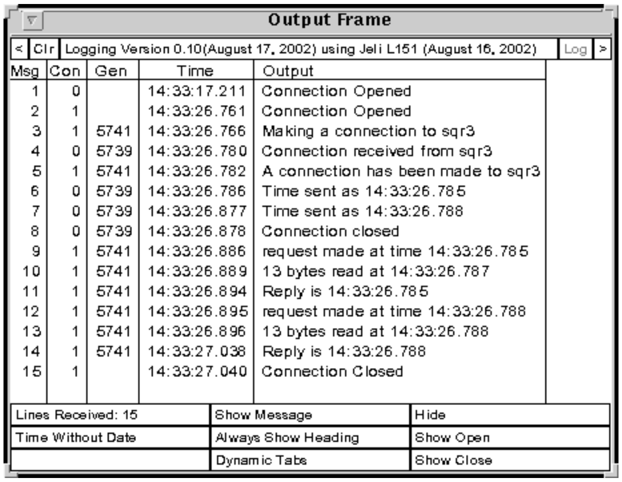
In other words, the second reply was logged before the first request, even though the second reply could not have been generated until the first request was made, its reply was received and the second request was made.

Messages 10 and 11 come from the client and are generated by consecutive statements in the client. They are logged 5 ms apart. Messages 13 and 14 are the corresponding messages from the second request from the client. Notice that although these also were generated by consecutive statements, they were received 142 ms apart. Most likely a context switch occurred on the client or the logger.

Because everything is on one machine, the communication between client and server is very fast. Only 3 ms separates the time between the two replies by the server as illustrated by the output columns for messages 6 and 7. This example also shows the latency between the sending and receiving of messages. The latency of message 6 is only 1 ms, while the latency of message 7 is 89 ms. In general the logger will not be able to reveal anything about this latency, except when the logger is run on the same host as the one being logged and the message sent contains the time the message was generated.

In Figure 4 the client and the server are on different machines, and so the logger is set to just show times relative to the first connection. Messages 8 and 12 show that the difference in time between the two replies is 73 ms, and messages 7 and 11 show that the time between the requests is 110 ms. Messages from different hosts can be received out of order. For example, message 14 from the server indicating when the reply was sent is received at the logger after message 11 from the client indicating that the reply was received. That is, the reply message was logged before the corresponding request.

For an assignment such as the request-reply protocol, students could hand in either a screen shot of the **Output Frame**, or a printout



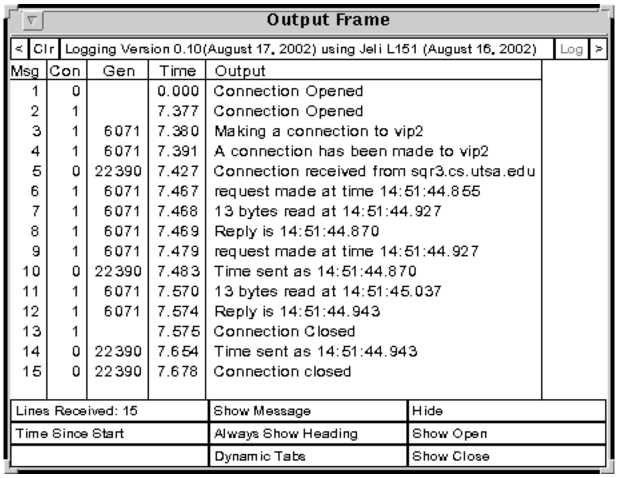
Logging Version 0.10(August 17, 2002) using Jeli L151 (August 16, 2002)

Msg	Con	Gen	Time	Output
1	0		14:33:17.211	Connection Opened
2	1		14:33:26.761	Connection Opened
3	1	5741	14:33:26.766	Making a connection to sqr3
4	0	5739	14:33:26.780	Connection received from sqr3
5	1	5741	14:33:26.782	A connection has been made to sqr3
6	0	5739	14:33:26.786	Time sent as 14:33:26.785
7	0	5739	14:33:26.877	Time sent as 14:33:26.788
8	0	5739	14:33:26.878	Connection closed
9	1	5741	14:33:26.886	request made at time 14:33:26.785
10	1	5741	14:33:26.889	13 bytes read at 14:33:26.787
11	1	5741	14:33:26.894	Reply is 14:33:26.785
12	1	5741	14:33:26.895	request made at time 14:33:26.788
13	1	5741	14:33:26.896	13 bytes read at 14:33:26.788
14	1	5741	14:33:27.038	Reply is 14:33:26.788
15	1		14:33:27.040	Connection Closed

Lines Received: 15

Time Without Date

Figure 3: Output from a request-reply protocol with clients and server on the same host.



Logging Version 0.10(August 17, 2002) using Jeli L151 (August 16, 2002)

Msg	Con	Gen	Time	Output
1	0		0.000	Connection Opened
2	1		7.377	Connection Opened
3	1	6071	7.380	Making a connection to vip2
4	1	6071	7.391	A connection has been made to vip2
5	0	22390	7.427	Connection received from sqr3.cs.utsa.edu
6	1	6071	7.467	request made at time 14:51:44.855
7	1	6071	7.468	13 bytes read at 14:51:44.927
8	1	6071	7.469	Reply is 14:51:44.870
9	1	6071	7.479	request made at time 14:51:44.927
10	0	22390	7.483	Time sent as 14:51:44.870
11	1	6071	7.570	13 bytes read at 14:51:45.037
12	1	6071	7.574	Reply is 14:51:44.943
13	1		7.575	Connection Closed
14	0	22390	7.654	Time sent as 14:51:44.943
15	0	22390	7.678	Connection closed

Lines Received: 15

Time Since Start

Figure 4: Output from a request-reply protocol with clients and server on different hosts.

of the log file. Alternatively, the students could put their log files (which are in HTML format) on their web sites, along with a commentary explaining the output. This gives the instructor valuable information about the operation of the program without the need for detailed analysis of the code. It can also be used to pinpoint errors.

8 Example: Tunnel Monitor

A tunnel is a web utility that passes web traffic to a fixed destination without any interpretation or modification. A web browser running on host A makes a request to host B for a resource that actually resides on host C. A tunnel server running on host B creates a tunnel process that makes a connection to host C. All information that comes to the tunnel from host A is sent to host C, and all information that comes from host C is sent to host A. Host A thinks that it is communicating only with host B and knows nothing of host C which may be behind a firewall. This hides and protects host C from the rest of the network. Tunnels are commonly used to support web servers behind firewalls.

A simple implementation of a tunnel uses two processes (or threads), one that reads only from host A and one that reads only from host C. Each of these executes code in a loop similar to:

```
bytes_read = read(fromfd, buf, BLKSIZE);
write(tofd, buf, bytes_read);
```

This is somewhat oversimpilfied in that the error checking is not shown and it assumes that the `write` outputs all of the bytes requested. To monitor the number of bytes read on each iteration of the loop by each process, open the remote log file before creating the second process and insert the following line in the loop:

```
lprintf(mf, "Bytes:  %d", bytes_read);
```

In the HTTP protocol, a request consists of an initial request line followed by additional header lines, an empty line and possibly a resource. The initial request and the header lines contain only printing characters and line terminators. The resource can be arbitrary binary information. The response has a similar format. Monitoring the initial requests that come through a tunnel requires reading and logging the initial request and then transferring bytes in either direction until the connection is closed. Reading the initial request typically requires reading a byte at a time until a line feed is found. Monitoring and logging the header lines requires reading and logging lines until a blank line is found, and then transferring the binary resource. This parsing of the transmission adds considerable complication to the tunnel.

The remote logging facility has features that allow monitoring the full headers without parsing the transmission. When a message is logged, all non-printing characters are ignored except for the line feed. A line feed imbedded in a message causes the message to be logged on separate lines, each with the same message number. The logging can be accomplished by inserting the following line into the tunnel loop for each process:

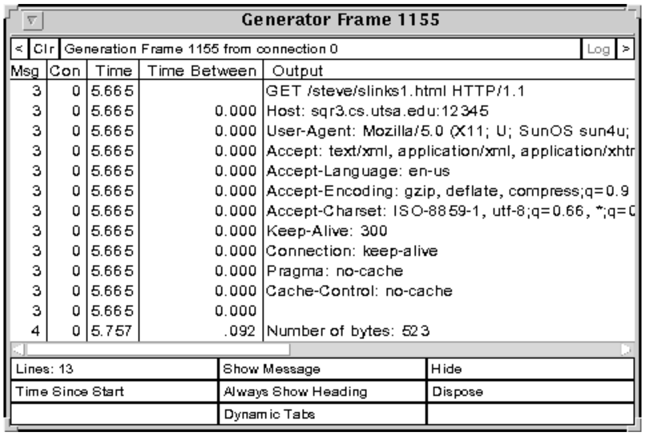
```
lprintf(mf, "%.*s", bytes_read, buf);
```

The use of the variable precision format specification limits the number of bytes output to `bytes_read`. This is necessary because the header lines are not strings. All header lines will be correctly logged. If the resource contains only printing characters, it will be logged also. Otherwise, some, all, or none of the resource will be logged. Figures 5 and 6 show the **Generator Frame**s for the two processes of a tunnel handling two consecutive requests for the same short resource. In Figure 5 a simple GET request is made by the browser. The request contains several header lines, all of which were logged by a single `lprintf` having message number 3. Figure 6 show the reply, a simple one line resource.

9 The Implementation as a Teaching Tool

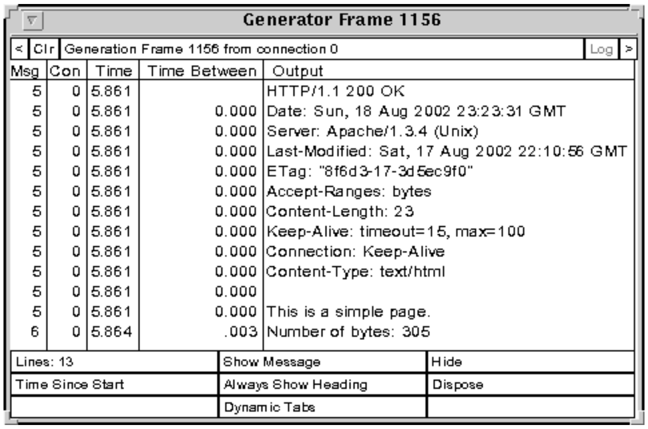
The implementation of the C logging library gives examples of techniques that could be discussed in class. Some of these include the use of conditional compilation to handle threaded and nonthreaded applications, the careful use of allocated memory to avoid memory leaks and the use of functions with a variable number of parameters.

Most interesting is the method used to guarantee that the messages are logged atomically. It is not sufficient to send a message to the remote logger with a single call to `write`. When writing to a network, it is not unusual for a `write` to return with fewer bytes written than requested. Even if it does always write the entire amount requested, there is no guarantee that writes to the same connection from multiple processes will not be interleaved.



Msg	Con	Time	Time Between	Output
3	0	5.665		GET /steve/slinks1.html HTTP/1.1
3	0	5.665	0.000	Host: sqr3.cs.utsa.edu:12345
3	0	5.665	0.000	User-Agent: Mozilla/5.0 (X11; U; SunOS sun4u;
3	0	5.665	0.000	Accept: text/xml, application/xml, application/xhtr
3	0	5.665	0.000	Accept-Language: en-us
3	0	5.665	0.000	Accept-Encoding: gzip, deflate, compress;q=0.9
3	0	5.665	0.000	Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=
3	0	5.665	0.000	Keep-Alive: 300
3	0	5.665	0.000	Connection: keep-alive
3	0	5.665	0.000	Pragma: no-cache
3	0	5.665	0.000	Cache-Control: no-cache
3	0	5.665	0.000	
4	0	5.757	.092	Number of bytes: 523

Figure 5: The output for a tunnel request.



Msg	Con	Time	Time Between	Output
5	0	5.861		HTTP/1.1 200 OK
5	0	5.861	0.000	Date: Sun, 18 Aug 2002 23:23:31 GMT
5	0	5.861	0.000	Server: Apache/1.3.4 (Unix)
5	0	5.861	0.000	Last-Modified: Sat, 17 Aug 2002 22:10:56 GMT
5	0	5.861	0.000	ETag: "8f6d3-17-3d5ec9f0"
5	0	5.861	0.000	Accept-Ranges: bytes
5	0	5.861	0.000	Content-Length: 23
5	0	5.861	0.000	Keep-Alive: timeout=15, max=100
5	0	5.861	0.000	Connection: Keep-Alive
5	0	5.861	0.000	Content-Type: text/html
5	0	5.861	0.000	
5	0	5.861	0.000	This is a simple page.
6	0	5.864	.003	Number of bytes: 305

Figure 6: The output for a tunnel response.

The interaction is handled by sending all output to a pipe. Pipes have the property that any write of size less than `PIPE_BUF` is guaranteed to be atomic. The writes will not be interleaved. The `lopen` function creates a pipe and all writes are done to this pipe. It also creates a child process that reads from the pipe and sends the result to the network. Since there is only one processes writing to the network for each connection, the correct sequence of bytes is maintained.

10 Conclusions

The logging facility described here consists of two parts, a local logging library that is linked to the user program being logged and a GUI program that displays the logged information. The logging facility can aid students in understanding the behavior of concurrent programs. A few minutes of explanation is sufficient to allow students to use the facility since the interface is similar to the standard C I/O interface. It can be used in an environment with multiple processes or threads on multiple machines, and the complexity of the analysis and display by the remote GUI does not perturb the programs that are being logged.

The user can dynamically customize the views of the logged data to show the aspects that are most relevant. The logs can be saved in HTML format and printed from a standard browser. By examining the printed logs, a student or instructor can gain insight into the details of the running of concurrent programs that would otherwise be hidden. The student can use this information to debug programs and this same information can aid the instructor in grading. The implementation of the local logging facility can be used as a case study to illustrate how writes from multiple sources can be made atomic.

Source code for the C local logging library, a jar file for the Java GUI and a users guide are available on the web [10].

11 Acknowledgments

This work has been supported by an NSF grant: *A Course in Experimental Techniques for Computer Science Majors: Proof of Concept*, DUE-0088769. It was also supported by a UTSA Faculty Research Award.

References

[1] Atkin, B. and Sirer, E. G., "PortOS: an educational operating system for the Post-PC environment," *Proc. 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 116-120.

[2] Barros, J. P., "Advance CS courses: Specific proposals for the use of petri nets in a concurrent programming course," *Proc. 7th Annual Conference on Innovation and Technology in Computer Science Education*, 2002, pp. 165-167.

[3] Carr, S., Feng, T. J., Mayo, J. and Ching-Kuang, S., "A communication library to support concurrent programming courses," *Proc. 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 360–364.

[4] Exton, C., "Elucidate: A tool to aid comprehension of concurrent object oriented execution," *Proc. 5th Annual SIGCSE/SIGQUE conference on Innovation and Technology in Computer Science Education*, 2000, pp. 33–36.

[5] Holland, D. A., Lim, A. T. and Seltzer, M. I., "A new instructional operating system," *Proc. 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 111-115.

[6] McDonald, C. and Kazemi, K., "Teaching parallel algorithm with process topologies," *Proc. 32rd SIGCSE Technical Symposium on Computer Science Education*, 2001, pp. 70-74.

[7] Robbins, S., "Exploration of process interaction in operating systems: a pipe-fork simulator," *Proc. 33rd SIGCSE Technical Symposium on Computer Science Education*, 2002, pp. 351-355.

[8] Robbins, K. and Robbins, S., *Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading*, Prentice Hall, 1996.

[9] Robbins, S., "Remote logging in Java using Jeli: A facility to enhance development of accessible educational software," *Proc. 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, pp. 114-118.

[10] Robbins, S., Logging Facility, 2002. Online. Internet. Available WWW: **http://vip.cs.utsa.edu/nsf/logging/**