An interaction network monitor for Amoeba

Paul Ashton
Department of Computer Science
University of Canterbury

TR-COSC 10/95, Oct 1995

An interaction network monitor for Amoeba

Paul Ashton*

Department of Computer Science
University of Canterbury

TR COSC 10/95, October 1995

Abstract

The interaction network has been proposed as a way of representing interactive processing in a distributed system, and an interaction network monitor for SunOS was developed in earlier work. This paper describes development of an interaction network monitor for Amoeba, the first interaction network monitor for a distributed operating system. Case studies are used to show some of the types of information the monitor can capture, and to show that the monitor is a valuable tool for monitoring complex modern systems.

1 Introduction

The interaction network has been proposed as a way of representing the *system reaction* to a *user action* (any user input). We define a system reaction to be all processing (process execution and message passing) that is the direct result of a single user input to any type of computer system [3]. The interaction network is particularly well suited for representing **interactive** processing in distributed systems, because for each input it shows exactly what happened during the response time for that input. The first interaction network monitor was developed for the SunOS operating system. Some interesting results were obtained using the monitor, but by the time it was completed the Sun 3 hardware and SunOS 4.0 software were basically obsolete.

An interaction network monitor has been developed for the Amoeba distributed operating system [6]. Amoeba was seen as a better development environment than SunOS for two main reasons. First, Amoeba was designed from scratch as a distributed operating system. Second, the source code of Amoeba is readily available.

Our main aims in this paper are:

1. To report on experiences with instrumenting a microkernel operating system (Amoeba) to record interaction networks.

2. To show, through examples, how recording of interaction networks can lead to increased understanding of the operation of such a system, and how interaction networks have been used in some small performance studies.

*e-mail: paul@cosc.canterbury.ac.nz

The report begins with a brief introduction to interaction networks, and the remaining sections address the aims listed above.

Note: the figures in this document are in colour, and are best viewed in colour. The figures can still be understood if the document is viewed in monochrome, but viewing in colour is definitely preferable.

2 The interaction network

Each interaction network represents the system reaction to a single user action (such as a keystroke or mouse event). A system reaction is very different to the execution of a single program because the system reaction involves complete and partial execution of many programs. We regard reactions as being performed by communicating threads that are distributed over one or more machines.

An interaction network is an acyclic digraph. Each vertex in the graph represents an event and each edge represents a period of thread execution, or message progression, between two events. To date, we have developed two ways of analysing interaction networks. One analysis method is to provide a browser that produces a display of an interaction network. The other analysis method involves producing reports that contain summaries of the times spent by threads and messages in various states during the system reaction, and counts of activities of various types. It is up to the user to decide how states are defined. One possibility is to define states in terms of use of hardware and software resources.

The interaction network concept was developed originally to allow analysis of interactive performance. The reports on resource-use produced by the second analysis method contain information intended for use in performance analyses of this type. We have, however, found interaction networks to be useful in **any** situation in which increased understanding of interactive processing is required, with interaction network displays being especially useful.

3 Overview of the Amoeba monitor

The design and implementation of the Amoeba interaction network monitor draws heavily on the work done on the SunOS monitor [1, 3]. A set of probes has been added to the Amoeba kernel to record the occurrence of various events, and to maintain the identifiers used to group event records into interaction networks. Each probe invokes a local event recorder to have an event record created, and ultimately stored to disk. Analysis tools extract interaction networks from a set of related log files, and provide textual and graphical views of interaction networks.

3.1 Probes added

A small set of probes has been added to the Amoeba kernel. Most record "structural events"—the events that must be recorded to show the extent of a system reaction, where the extent of a reaction is the messages and periods of thread execution that resulted from a given user input. Probes for recording resource (CPU, disk, network, and so on) use have yet to be added.

The main probes can be classified into three groups:

1. Detection of user input. One probe has been added to the terminal server to record when user input occurs on a serial line. Mouse and console keyboard input have yet to be instrumented, so all interaction networks presented resulted from keyboard input through a character terminal interface.

2. Thread life-cycle[1]. Probes have been added to detect thread creation and termination, and when a process is assigned a new name.

3. Communication. Most communication in Amoeba is done using remote procedure calls. Probes have been added to record the sending and receiving of remote procedure call and return messages.

In addition to making a call to the event recorder, each probe is responsible for maintaining identifiers used to associate each event record with a system reaction. The techniques developed for this purpose in the SunOS interaction network monitor (described in detail in [1]) have been used successfully in the Amoeba monitor. The SunOS techniques are based on a number of assumptions about the spread of a system reaction, the most important of which are: that when a thread T sends a message or creates a new thread, then the message or new thread are part of same system reaction as the one that thread T is currently associated with; and that when a thread receives a message it becomes associated with the same system reaction as that of the message.

3.2 The event recorder

During monitoring, event records are logged to disk by an event recorder that is divided into client and server components. Each probe logs event records to the server in its local kernel. Event records are double buffered in the server in two 30000 byte buffers. Clients are responsible for making RPCs to servers to retrieve full buffers.

Because the server has an RPC interface, the event recorder client need not run on the same machine as the server. In the case studies reported here, clients executed on a host running SunOS to minimise the monitoring overhead on the Amoeba system.

3.3 Clock synchronisation

A Sparcstation 1 has two microsecond timers. One is used by the kernel to provide standard timing services. The second has been taken over for use by the event recorder. The interrupt frequency of this second timer is very low (one interrupt every two seconds), so the chance of losing an interrupt is extremely small.

No attempt is made to synchronise clocks during event recording. Once monitoring is complete, there is one file of event records for each machine that was monitored. A clock correction program has been written that takes two files of event records and corrects the timestamps in one to be consistent with the timestamps in the other. Several algorithms have been implemented, including those described by Duda *et. al.* [4], enhanced versions of Duda's algorithms, and some new algorithms. Research into the effectiveness of the various algorithms is continuing, with results to date showing that very good synchronisation can be achieved.

[1] In Amoeba, a process is basically an address space. The kernel supports multiple threads of execution within each process. In the remainder of this paper, the Amoeba definitions for "process" and "thread" are assumed.

3.4 Analysis tools

The following analysis tools have been developed:

1. `insplit`, a program that takes a collection of log files recorded at the same time on a collection of machines, and from them produces a collection of log files each of which contains the event records of a single interaction network.

2. `logdump`, a program that produces a textual description of every event record in a log file.

3. `browser`, a program for display and inspection of an interaction network.

4 Case studies

Three case studies are presented to demonstrate the capabilities of the Amoeba interaction network monitor. The first involves execution of the `ls` command from the Bourne shell. It is presented to show the way in which Amoeba executes user programs. The second and third case studies show use of the interaction network monitor in two small performance studies. The first of these was done to determine reasons for losing clock interrupts, and the second to find reasons for the interactive performance of the `elvis` editor.

The case studies are illustrated with displays produced by the interaction network browser. Before presenting the case studies, we give some information on the display format used, and on the configuration of the Amoeba system on which the case studies were performed.

4.1 Amoeba background

The Amoeba system used in the case studies consisted of five Sparcstation 1 machines (scooter, ralph, piggy, gonzo, grover) connected by an ethernet segment that was bridged off the department's network. All five machines ran Amoeba 5.2, with patches up to patch 5 applied. One machine, scooter, was the file server.

In Amoeba, many servers are involved in supporting interactive processing. Some servers run within the kernel; others run as user processes. Some servers are present within all kernels; others exist on one machine only. For each server, one or more threads is created to service calls to the server. Use of the following servers occurs in the case studies:

- tty. Each kernel has a tty server that provides access to local terminals. In all of the case studies presented here, a terminal connected to scooter was is use.

- soap. The directory service (mapping names to capabilities) is performed by soap. There is one soap server for the whole system located on scooter, the file server.

- bullet. There is one bullet file server for the whole system located on scooter.

- session. Unix emulation in Amoeba is provided by Ajax, which consists of library functions and the session server. The session server, for example, simulates the existence of Unix process ids and maps them to Amoeba process identifiers. There is one session server per logged in user.

- run. The run server makes load balancing decisions. All five machines were considered by the run server whenever it was asked to select a machine on which a new process should be started. There is one run server, and it runs on piggy.
- process/segment. Management of processes and memory segments is carried out within each kernel by the process/segment server.

4.2 Interaction network displays

In an interaction network display, the vertices (each representing an event) are arranged in columns, with one column for each thread that performed processing in response to the user input. Time increases as you go down the network, with the Y-coordinate of each vertex in direct proportion to the time at which the event represented by the vertex occurred. The length of an edge in the Y direction shows, therefore, the duration of the activity that it represents. All of the vertical edges represent activities carried out by threads. Each non-vertical edge represents a message, or the initial activity performed by a newly-created thread.

The browser uses colour to highlight execution on different machines, using green for scooter, light blue for piggy, red for ralph, purple for grover and grey for gonzo. All vertices and edges associated with a thread have the colour of the machine on which the thread ran. Vertices and edges associated with messages local to a machine have the colour of that machine. Vertices and edges associated with messages that were sent across the network are coloured black.

4.3 Case 1—`ls`

The interaction network in Figure 1 shows execution of the `ls` command. The user action was entry of the newline character at the end of a command line on which the user had previously entered `l` and `s`. A total of 11 threads, executing in 8 different processes spread over 4 machines were involved in the system reaction. The system reaction took 0.569 seconds, and involved 36 RPCs, and 191 ethernet packets containing a total of 182.7Kb of data. The ethernet utilisation averaged over the period of the reaction was, therefore, 26.3%.

In the figure, each thread is identified by machine (given by the colour of the thread's vertices) and two numbers at the top of the thread's column. The first number is a process id, and the second a thread id. Both are unique within a machine. The kernel of each machine executes as process 0. A summary of threads involved in the reaction, in the order in which they appear in the figure, is given in Table 1.

The interaction network and small pieces of source code were used in preparation of the following description of what is happening in Figure 1.

The vertex representing input of the newline is in the top left corner of the figure, in the tty server column. The message from the tty server to the shell contains the newline character, and is a reply to an RPC request that the shell had made earlier. The shell runs the `ls` command before prompting for the next command. Doing this involves it making a total of sixteen RPCs. The first four RPCs are to the soap server, and are made while looking for `ls` in the directories along the search path. On the fourth attempt `ls` is located in `/bin`. An RPC is then made to the session server to register the fact that the shell is about to create a new process.

Next the shell makes an RPC to the soap server to read the process descriptor from the executable. `/bin/ls` turns out to be a directory, so an RPC is made to soap to retrieve the

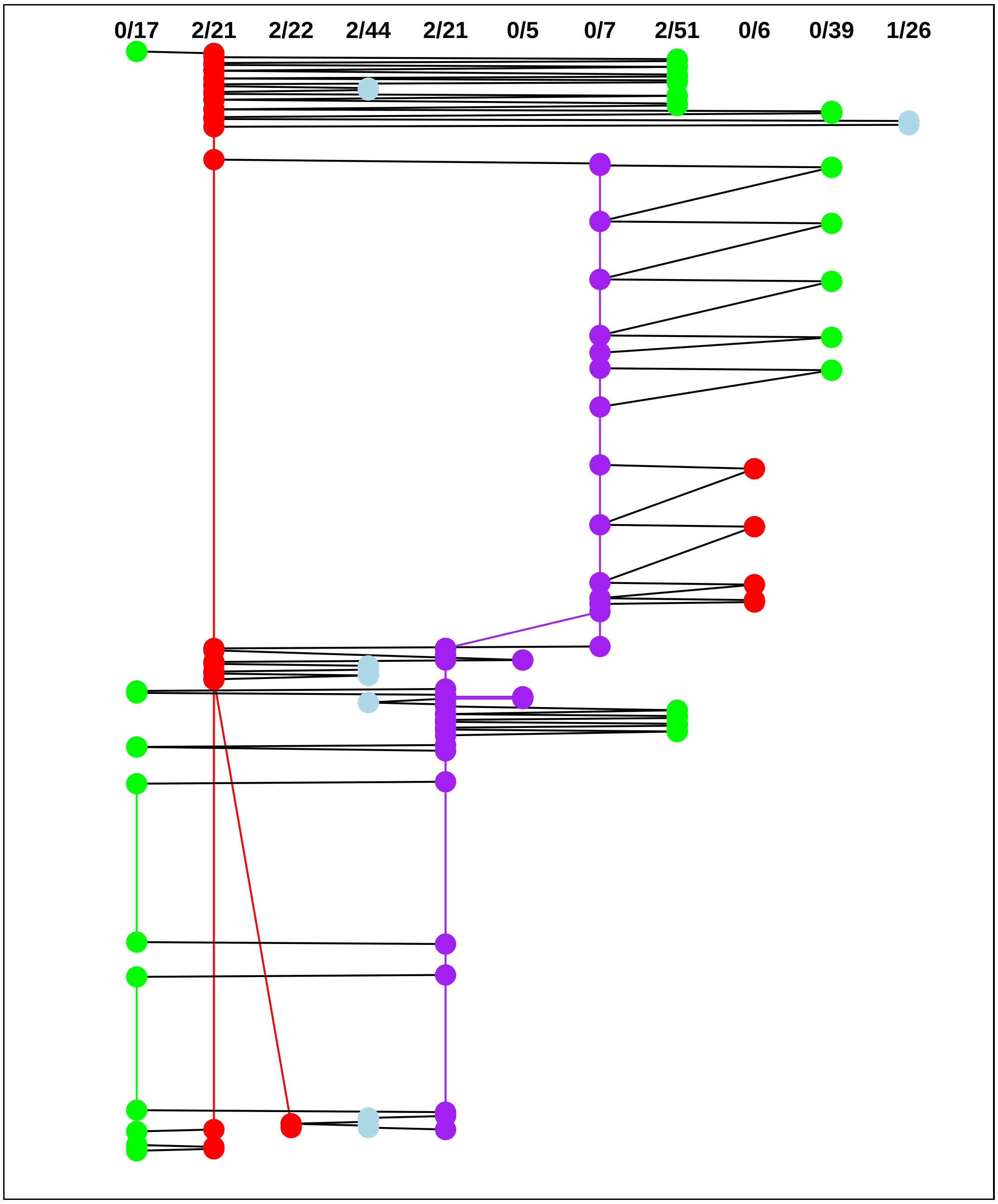

Figure 1: Interaction network showing system reaction to a newline input to the Bourne shell to request execution of `ls`.

thread id	machine	description
0/17	scooter	tty server
2/21	ralph	shell (main thread)
2/22	ralph	shell (waiting thread)
2/44	piggy	session server
2/21	grover	ls
0/5	grover	process/segment server
0/7	grover	process/segment server
2/51	scooter	soap (directory server)
0/6	ralph	process/segment server
0/39	scooter	bullet (file server)
1/26	piggy	run server

Table 1: Key to the threads shown in Figure 1.

contents of the directory. An RPC is made to the file server to read the process descriptor from the executable `/bin/ls/pd.sparc`. The process descriptor contains details of how the executable can be run, including memory segments and threads that must be created before execution can begin. The process descriptor is sent to the run server, which decides that the new process will be run on grover. The shell then creates a stack segment for the new process, before making an RPC to the process/segment server on grover to create the new process. When that RPC (eventually) finishes, the shell makes another RPC to the process/segment server to give a name to the new process. It then makes an RPC to the session server to confirm that the process has been created successfully. Another RPC is made to the session server to check for processes that have terminated. There are none, so the shell creates a new thread to wait for a process to terminate, and then waits. When the new thread finally awakens the main shell thread, it makes an RPC to the terminal server to print the prompt for the next command line, before sending a request to the terminal server for further input.

In creating the `ls` process, the process/segment server on grover makes a total of nine RPCs. The first eight of these are used to copy three segments: the executable's text segment (four RPCs to the file server), the executable's initialised data segment (one RPC to the file server) and the stack segment created by the shell (3 RPCs to the process/segment server on ralph). The final RPC instructs the process/segment server on ralph to delete the stack segment. With the three segments loaded the new thread is created, with the first activity in the thread represented by the diagonal edge from thread 0/7 to thread 2/21. Creation of the new thread completes the RPC to create the `ls` process, so the process/segment server replies to the shell. The RPC from the shell to the process/segment server on grover takes 0.253 seconds, 45% of the total response time.

`ls` makes a total of eleven RPCs. It begins by making a "get attributes" call to the terminal server through a library call to determine whether standard output is a tty. This is done during setting of flags. In setting of another flag, `ls` needs to find the effective user id; something that must be retrieved from the session server. An RPC to the local process/segment server to get the session server's capability is needed before the session server can be accessed. Two RPCs are made to the soap server to get status information on the directory that is to be listed by `ls`. These are followed by an RPC to the soap server to get the capability for the directory to be listed, and an RPC to read its contents. Another "isatty" RPC is made to the session server by the stdio library during allocation of a buffer, followed by two RPCs to write the two lines of output produced by `ls`. The terminal server takes a considerable time to reply to each of these calls. In each case the delay reflects the time taken to write the bytes supplied at a rate of 9600 baud.

Just before the `ls` thread terminates, it makes a call to the session server to inform it that `ls` is about to exit. The session server makes an RPC to the shell thread created to wait for `ls` to terminate. This shell thread then wakes up the main shell thread before terminating.

The processing that resulted from entry of newline is much more than the processing performed by `ls`, thread 2/21 on grover. This highlights the difference between interaction monitoring and program monitoring. All of the processing involved in setting up the `ls` process is shown, as well as the processing involved in tidying up afterwards. All processing done by servers, whether invoked by the shell, `ls` or other servers, is also shown. Only a small proportion of response time is accounted for by execution time spent in `ls` itself, with most time spent taken up by the shell, servers and network delays.

Space precludes discussion of other interesting aspects of the network.

4.4 Case 2—Finding a bug in the MMU context flushing code

Early in the development of the monitor, clock ticks were being lost even though the interrupt frequency (at that stage) was one interrupt per second. The interaction network in Figure 2 was recorded before this problem had been resolved. The user action was to type newline at the end of a simple shell command line (in this case the command executed by the shell was the `date` command).

Two lengthy periods of inactivity can be seen from the interaction network. Both occur just after creation of a new process. It seemed likely that these periods (each of about 1.08 seconds, out of a total response time of 2.8 seconds) were periods during which interrupts were disabled. Addition of new probes and investigation of the source code to determine code executed with interrupts disabled helped to track down the problem. There was a bug in the code for invalidating an MMU context, that meant that many more addresses than necessary were accessed during context invalidation. The bug was duly fixed, and the problem of lost clock ticks disappeared. Flushing an MMU context now takes a little under 4ms.

4.5 Case 3—Performance of `elvis`

Amoeba is used in an operating systems course in the School of Electrical Engineering at University of Technology, Sydney. In discussions with David Holmes from UTS, it became apparent that they had been having performance problems with `elvis`, a version of the `vi` editor distributed with Amoeba. Some interaction networks that resulted from inputs to `elvis` were subsequently recorded. One experiment is described below, and two interaction networks are then presented.

4.5.1 Overview of the `elvis` experiment

In this experiment, a few characters were entered into an input buffer, which was then saved to disk. In all, eighteen keystrokes were made. The characters `ibar` then escape put `elvis` into input mode, added "bar" to the buffer, then put `elvis` back into command mode. The characters `^ifoo` then escape moved the cursor to the beginning of the line, put `elvis` into

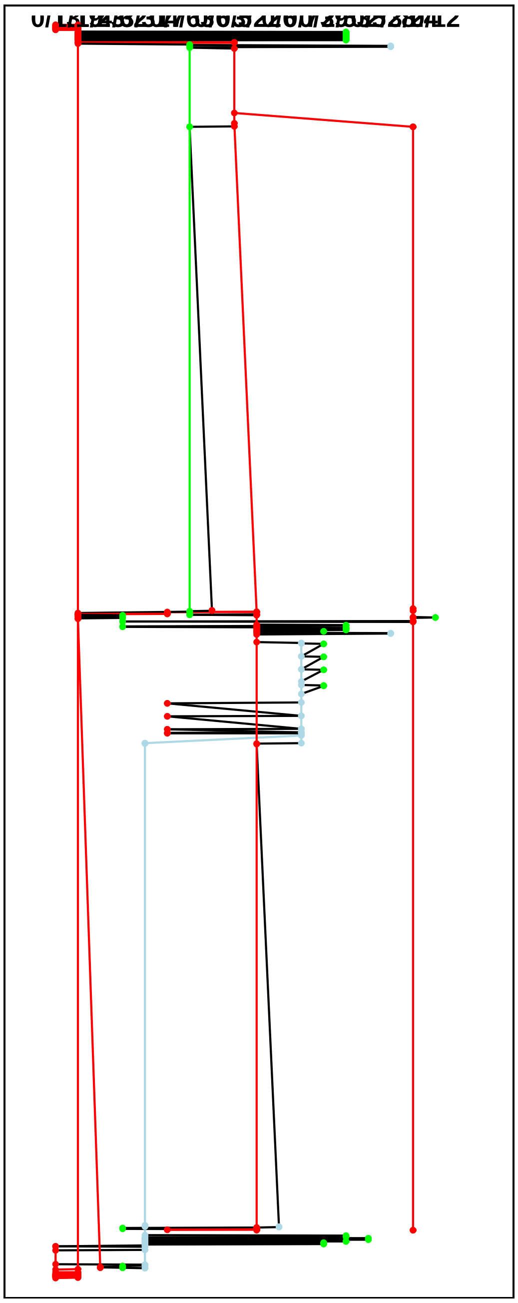

Figure 2: Interaction network showing execution of a small program before the context flushing bug was fixed.

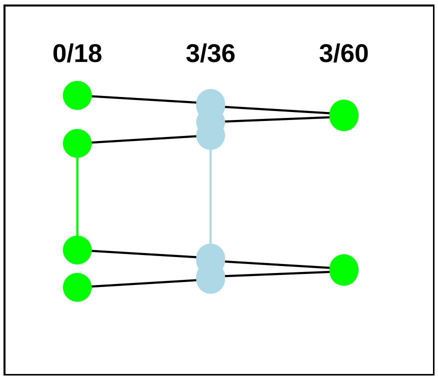

Figure 3: Interaction network showing system reaction to a single character inserted into an `elvis` buffer.

thread id	machine	description
0/18	scooter	tty server
3/36	piggy	elvis
3/60	scooter	session server

Table 2: Key to the threads shown in Figure 3.

input mode, added "foo" to the buffer, then put `elvis` back into command mode. The characters `:w foo` then return wrote the buffer to a file called "foo".

4.5.2 Insertion of a single character into the buffer

The first interaction network resulted from input of the third character of the input string, the 'a' in "bar". All `elvis` had to do in response was to add the 'a' to the input buffer. The interaction network recorded is shown in Figure 3. Three threads from three processes on two machines are involved, as summarised in Table 2. The response time is 33ms.

The terminal server sends a reply to an earlier RPC made by `elvis` to return the character input. `elvis` then makes an RPC to the session server, before making an RPC to the terminal server to perform output. After that, `elvis` makes a second RPC to the session server, before sending a request to the terminal server to ask for more input.

Nineteen characters are written by `elvis`, and this is reflected by the length of the edge representing the bytes being output by the terminal server. In the future we may investigate why nineteen characters were written, rather than the one just input. So far we have not done so because the output of eighteen additional characters places little load on the system as a whole, and the 17ms it adds to the response time is not likely to be of concern at the speed most users type.

The most puzzling aspect of the interaction network is the two RPCs made to the session server. Inspection of the call type recorded for the RPCs revealed that they resulted from calls to the `alarm` function. In Ajax, Amoeba's Unix emulation package, alarms are managed by the session server. Inspection of the `elvis` source code revealed that an alarm is set just before a `read` call is made to get terminal input, and is reset immediately after the call to `read` returns. The first RPC to the session server shown in Figure 3 is to reset the alarm after a character read has completed. The second is to set an alarm before requesting the next character.

One way of eliminating the RPCs to the session server is to change Ajax so that the alarm handling for a process is performed by a thread within the process. Further investigation of the `elvis` source revealed, however, that most of the alarm RPCs could be eliminated by a minor change to `elvis`. The timeout to use is a parameter to the elvis `ttyread` function. In most calls to `ttyread` the value of timeout is 0, which makes both calls to `alarm` unnecessary.

The `ttyread` function was changed so that `alarm` was called only if the timeout parameter was non-zero. This change eliminated most of the alarm RPCs to the session server. The sequence of 18 characters described above in Section 4.5.1 was input into the before and after versions of `elvis`. In the before version, 90 RPCs were recorded in the 18 interaction networks, whereas in the modified version the total was only 58.

The changes to alarm setting reduce input mode character handling to four RPCs in most cases. This is a very worthwhile change because the network is known to be a bottleneck for Amoeba, and this change has halved the network load for input mode character handling. There seems little scope to reduce further the number of RPCs without resorting to a combined write-buffer-then-read type of call. Network traffic could be further reduced by running `elvis` on the same machine as the tty server. Currently the Amoeba run server does not take probable message patterns into account when performing load balancing.

4.5.3 Input of `i` in command mode

The second `elvis` interaction network shows in Figure 4 the reaction to input of the second `i` command, the seventh character in the input sequence. This command causes `elvis` to enter input mode, in which characters entered are inserted into the buffer being edited. The response time is 44ms.

Four threads from three processes on two machines are involved. Three of the threads were also involved in the interaction network discussed in Section 4.5.2. The fourth is thread 0/40 running on scooter. It is one of the kernel threads created to run in the file server.

The message passing in Figure 3 is evident at the top and the bottom of Figure 4. In between, `elvis` makes three RPCs to the file server. A puzzling aspect of this figure is why entering input mode should require file server accesses.

Investigation of the `elvis` source code revealed that the file server RPCs were part of processing done to save to a temporary file information that would be needed if the insert command was subsequently undone. In fact, a total of eight RPCs are made to the file server in response to simple editing commands in the experiment. Three are made during processing of each of the `i` commands, as described above. One write RPC is made during processing of each of the `^` and `:` commands to flush the contents of the buffer to the tmp file. Changing `elvis` to make more use of memory data structures for working storage and less use of its tmp file is a more substantial change than the one needed to reduce the number of alarm RPCs, and has not yet been attempted.

5 Summary of experiences to date

Based on the experience we have had with the monitor to date, we can report that:

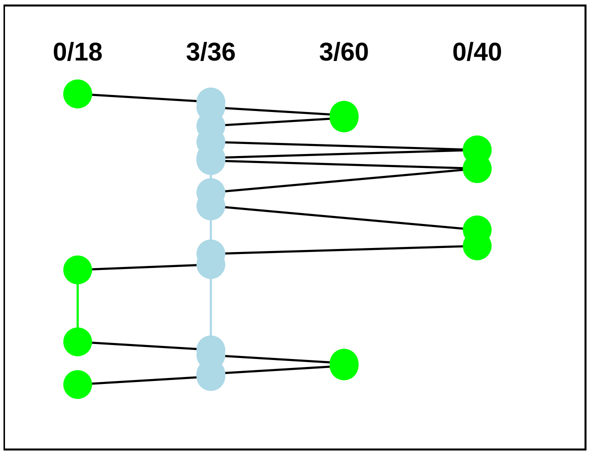

Figure 4: Interaction network showing system reaction to input to `elvis` of an `i` to put `elvis` into input mode.

- Although SunOS and Amoeba are very different operating systems, there were no difficulties in applying the interaction network concept, as well as the design and much of the source code of the SunOS monitor, to Amoeba. This was true even at the probe level.

- Propagation of interaction identifiers through remote procedure calls has been sufficient to allow recording of all processing for a wide variety of commands. To make the monitor more general, other communication methods (such as Amoeba's group message passing primitives) will have to be instrumented in time, but the current instrumentation is sufficient for most Amoeba applications.

- Two strengths of the interaction network are that it is oriented towards interactive processing, and that it records processing performed by servers, whether invoked directly or indirectly. Interaction networks recorded for inputs made through X windows (which will be possible once further probes have been added) will be even better at illustrating this than the examples presented here.

- Experience with the current tools has lead to several ideas about how they can be improved. Briefly, these ideas include: better support for relating interaction networks to the input stream that produced them, better support for relating events in interaction networks to source code statements, improved browsing of large networks, and development of a program to produce summaries of activities recorded in a set of one or more interaction networks.

- A feature of modern systems is a tendency towards use of many hardware and software components that interact in complex ways, and that are arranged into many layers of abstraction. Inefficiencies can arise because of the way in which the various layers interact, and such inefficiencies are often difficult for a programmer working with an individual layer to foresee. Also, new systems often provide traditional interfaces so as to support existing software, but the costs of the various operations provided by the interface may be very different to those found in traditional implementations of the same interface. This was illustrated in the `elvis` case study, where the costs for alarm setting and file access were higher than in the systems `elvis` was originally written for.

Interaction network monitoring is useful for such systems as the time consumed by components, and the interactions between components can be captured in interaction networks. In the case of Amoeba, the network is known to be a bottleneck. Interaction networks show clearly use of remote servers that results in network communication.

- With all event monitoring clients running under SunOS, the predominant source of monitoring overhead is the cost of sending buffers of event records across the network. The event records in the interaction network used in the `ls` case study occupy about 13Kb, and were generated over a period of 0.569 seconds. Given that each server uses 30000 byte event record buffers, it is likely that no network traffic due to monitoring occurred during recording of the `ls` interaction network.

  In monitoring over longer periods, however, and with the addition of further probes, event record buffers will be sent during monitoring, and will affect the information recorded by extending message delays. The pattern of server accesses will be the same though, and in many situations the extension of some message delays will be of little concern to the monitor's users. Also, a range of techniques can be used to minimise network load imposed by the monitor, including buffer compression, a second network dedicated to the monitor and local secondary storage on each machine.

6 Related work

Several program monitors exist that represent program execution using directed acyclic graphs. Examples include the program activity graph (PAG) [5], the integrated process-level execution log (IPEL) [7] and the causality graph [8]. A browser even exists for viewing causality graphs.

The key difference between our work and other work on monitoring of distributed systems is that the interaction network is based on monitoring *interactions*, not programs, machines, or entire systems. Note that it is possible to determine what happened in a program, machine, or system by combining information from several interaction networks. With current program monitors, however, it is not possible to determine what happened in individual interactions from a collection of program logs.

7 Summary

The Amoeba monitor is the first interaction network monitor for a distributed operating system. Implementing and using the monitor has led us to the following conclusions about interaction networks, and monitoring tools for distributed operating systems.

In implementing the monitor, we found that the interaction network concept could be readily applied to Amoeba. In fact, the design and much of the source code of the SunOS monitor have been used in the Amoeba monitor. The process of deciding on the location of probe points in a new kernel is never easy, but the experience of adding probes to the SunOS kernel made addition of the Amoeba probes a lot easier. The relative ease of the port to

Amoeba, and of a recent port to Minix [2], support our claim that the interaction network concept can be widely applied.

The case studies presented have shown some of the capabilities of the monitor. Recording of interaction networks has led to identification of a bug in the Amoeba code for flushing a Sparcstation MMU context, and of sources of inefficiency in the `elvis` editor. The `ls` interaction network provides considerable insights into how Amoeba performs interactive processing.

To understand causes of response time delays in complex modern systems you need support from appropriate tools. The examples presented highlight two strengths of the interaction recording approach: its emphasis on interactive processing, and its ability to cross program boundaries to capture **all** processing associated with a user action. Because of this the interaction network is well suited for use in monitoring of modern systems that consist of many interacting components that may be distributed over many machines.

References

[1] Paul Ashton. *The Interaction Network: a Performance Measurement and Evaluation Tool for Loosely-Coupled Distributed Systems.* PhD thesis, Department of Computer Science, University of Canterbury, Christchurch, New Zealand, March 1992.

[2] Paul Ashton, Carl Cerecke, Craig McGeachie, and Stuart Yeates. Use of interaction networks in teaching Minix. Technical Report TR-COSC08/95, University of Canterbury, Department of Computer Science, September 1995. URL: `http://www.cosc.canterbury.ac.nz/~paul/tr-cosc.08.95.ps.gz`.

[3] Paul Ashton and John Penny. A tool for visualising the execution of interactions on a loosely-coupled distributed system. *Software—Practice and Experience*, (accepted for publication).

[4] A. Duda, G. Harrus, Y. Haddad, and G. Bernard. Estimating global time in distributed systems. In *7th International Conference on Distributed Computing Systems*, pages 299–306. IEEE, September 1987.

[5] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, Sek-See Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[6] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[7] J. J. P. Tsai, Kwang-Ya Fang, Hong-Yuan Chen, and Yao-Dong Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering*, 16(8):897–915, August 1990.

[8] Dror Zernik, Marc Snir, and Dalia Malki. Using visualization tools to understand concurrency. *IEEE Software*, 9(3):87–92, May 1992. Their notation has many similarities to the interaction network.