M.I.T. Laboratory for Computer Science
Computer Systems Research Division

May 29, 1980
Request for Comments No. 187

Debugging in a Distributed System

by Robert Schiffenbauer

The following is a Master's thesis proposal.

Brief Statement of the Problem:

Interprocess communication via message passing has become a central concept in distributed computer systems. A software package is proposed to enable user monitoring and debugging of conversations between executing processes at separate nodes. It is intended that such a package will complement conventional subsystems that merely permit debugging of the process executions themselves.

1. Introduction

The growth of distributed computing has created a new way of looking at programming systems. Rather than a collection of tasks running on a single CPU sharing the same real address space, a particular job may now be implemented as a group of processes executing concurrently at remote sites, maintaining cooperation through the use of message passing and acknowledgement.

Distributing the computation in this fashion complicates the task of monitoring and control of a particular job. No longer does interprocess coordination consist of merely setting semaphores via P and V operations or leaving data in specified memory areas by producer processes for later use by consumer processes. Now processes must take a much more active role in coordinating themselves. This leads to the creation of an entirely new area for possible programming errors, and, hence, a new area that must be monitored and debugged by the programmer. For complicated systems, correct interprocess message communications may be more important for obtaining good results than the correct execution of any particular process.

It seems appropriate, then, to develop new system tools which are tailored to a distributed environment by aiding the programmer's understanding and usage of message passing. In particular, a program debugger ought to include facilities for monitoring the communications between executing processes in the same way that conventional debuggers monitor the course of the executions themselves. Such a system would allow the capability of switching back and forth between the monitoring of the execution of a particular process (or group of processes if the debugger is sophisticated enough) and the monitoring of the intercommunications between several processes.

Unfortunately, we know of no existing debuggers that permit both intraprocess and interprocess monitoring. We hope to rectify that situation by the time this thesis is completed.

2. The Project

The debugger will be written in the *Mesa* language and will exist as a process, invokable at any time by the user, running on a particular *Alto* personal computer. The communications to be monitored are those sent as messages (called *PARC Universal Packets* - or *PUP*'s) over the *Ethernet*, a coaxial cable connecting the various *Alto*'s in the Laboratory for Computer Science. This is the only hardware equipment needed for this project. Since the *Ethernet* is currently up and running, no problems are foreseen in this area. As far as software equipment goes, packet generating processes are needed to test the debugger. There is no lack of such processes currently running on the *Ethernet* (file transfer protocols for example). Anything else that is needed will be written in *Mesa* by the author.

A good point of departure for this discussion is the view that debuggers depend on three main concepts: dumping, tracing and breakpointing [1]. Dumping of core gives the user a view of the system state at a particular point of execution. Tracing reveals to the user the particular sequence of operations performed by the system during a particular interval of time. Breakpointing is the mechanism whereby the user interrupts the execution of his program in order to hand control over to the debugger so that tracing and/or dumping can be accomplished.

It is believed that these three features are just as important for communications monitoring as they are for execution monitoring and ought to be included in any distributed system message debugger. In particular, a user ought to be able to trace the flow of messages through the system as processes execute; he ought to be able to examine at his leisure the status of incoming and outgoing information at any node he chooses (for example, he ought to be able to determine what fraction of a particular file has been received by node Y from node X at the current time); and he should be able to set breakpoints to suspend interprocess communications to allow traces and dumps to be performed.

Tracing - The user ought to be able to specify exactly how global a view he desires for tracing. In the ideal case, the entire *Ethernet* would be available to the user at a glance. He could focus his concentration on any particular nodes he likes simply by examining a particular part of the screen. However, this flexibility may be neither wise nor necessary. It is probably unwise because the *Alto* screen is by no means large enough to clearly display the entire *Ethernet*. Screen crowding is seen as a constant adversary throughout this project. It is probably unnecessary because (hopefully) few jobs will need to monitor the communications from all (or even most) nodes in the net. Thus we allow the user to specify what nodes he is directly interested in and place only those named on the screen for monitoring.

Furthermore, we allow the user, if he so desires, to specify the course of a message trace by supplying him with the power to indicate which packet (or group of packets) is to be sent next (in other words, the user can decide what the next step in the trace will be). Obviously, this is a

capability that cannot exist in conventional process execution debuggers. Alternatively, the user can elect to allow the system to decide the order of the trace. The system algorithm for determining the "next step" (i.e. the next message to be passed) in a particular trace has yet to be worked out. A time ordering scheme would need to be constructed. Perhaps something based on Reed's pseudo-temporal environments might prove useful.

Dumping - The user ought to be able to determine the progress of a particular conversation between two or more nodes. He should have the power to examine in detail any *PUP* in the system at any time. We believe that the user will most often desire to see the next *PUP* to be sent to a particular node in order to determine, in a limited way, how much information that node has thus far received (e.g. how much of a file has been transferred at a particular point in time). However, the user may examine any *PUP* he wishes.

We acknowledge that this is a roundabout way of determining the "state" of interprocess communications, and admit that it really is not the same concept as exemplified by memory dumps in current debuggers. However, the alternative, which involves maintaining indicators at each node to reflect the communication state for that node at a particular point in time would appear to be well beyond the scope of this thesis. Indeed, it might make a good master's thesis topic in itself.

Breakpointing - We intend to give the user the capability of stepping through his debugging at any pace he desires. A "single-step" trace capability will be made available, which would have the same effect as physically setting breakpoints after each occurrence of a *Send PUP* command in any process. "Slow-step" traces will also be possible, with the user allowed to break in at any time to make adjustments (e.g. altering packet contents, changing the order in which packets are sent, originating new packets artificially, changing packet destinations, etc.) or simply examine packets. Finally, the user may choose not to set breakpoints and simply sit back and watch the communication flow.

3. The Particulars

The goal is to make monitoring and debugging as easy as possible for the user. To this end we see two major subgoals. However, in all probability, and unless the implementation of one or the other of these proves unexpectedly easy, only one of the subgoals will be met.

User Interface - We view the debugger as providing a clear and coherent picture of the intercommunications process. As such, our first subgoal is to implement a powerful interface that presents a graphic representation of what (we assume) is a natural conception of that process. Specifically, we theorize that the interface ought to look not unlike an air-traffic controller's screen, with markings indicating both packets and *Ethernet* nodes. The debugger-user will employ the *mouse* to point at specific screen areas and, in conjunction with this, issue simple one or two character commands. For example, the user may select a particular *PUP* on the screen (represented, say, as a filled in rectangle) and then a particular node (a circle, say) and issue a *Send* command by hitting a *carriage return*. The effect of this within the system will be to actually send the designated *PUP* to the designated node and to make some adjustments in debugger controlled node clocks for each node involved in the transaction. This will be manifested on the screen by the rectangle flowing (at some predetermined speed, perhaps signified by the user) from its source node circle to its destination node circle. We hope that the commands will not be burdensome to the user, and that the screen will be interesting to watch, sufficiently so that users will be willing to make use of the system in the first place.

Above, we alluded to making some adjustments to node clocks. It is quite apparent that, at this time, *PUP*'s flow too swiftly through the system to allow 100% interception and processing by the debugger package (for some commands such interception will be necessary). Thus this project involves the creation and management by the debugger of virtual node clocks whose rates of flow can be altered (by the debugger) at will. The ability to control time allows a *PUP* to be sent on its way only when the debugger (or perhaps the user) sees fit.

One problem which we have considered and do not yet have a solution for is the nature of the switching back and forth between the monitoring of process executions and the monitoring of the communication lines. During a trace, do we switch to the "air-traffic" screen every time a *PUP* is to be sent (this would lead to an inordinate amount of switching and user annoyance) or do we let process execution continue until a number of *PUP*'s are to be sent, switching only then to the message screen so that a series of *PUP* deliveries are seen in a continuous manner (thereby giving the user a false view of the sequence of instruction executions in the trace). This needs more thought.

High Level PUP *Representation* - A probable user desire will be to examine in detail the structure of a particular *PUP* before it is sent through the net to its destination. Currently existing programs (such as *Etherwatch*), display packets in a fashion that leaves much to be desired. *Etherwatch*

simply prints out decimal values of particular packet bytes. We feel (as our second subgoal) that a useful debugger ought to display packets at the level at which the user is thinking about his programs, namely, as constructs of the language being programmed in (perhaps *Mesa* or *ECLU*). Single packets, or multiple packets configured upon the screen as the user dictates, ought to be displayable. Such representations will doubtlessly make life much easier for human users.

We expect that the user will interface to this capability by pointing at a particular *PUP* and typing *d, carriage return* (*d* for *display*). The *PUP* contents will be blown-up to cover some part of the screen (or perhaps all of it). The user can examine the *PUP* contents at his leisure and return to the "air-traffic" screen whenever he desires.

As previously stated, in all probability only one of the two subgoals will be attacked. If we manage to implement the "air-traffic" screen, then we will simply allow *PUP* contents to be displayed in the *Etherwatch* fashion, and leave the user to fathom the bits. If, on the other hand, we obtain a good high level *PUP* representation, then the "air-traffic" screen will be replaced by a simpler view. Perhaps the screen can be divided into partitions, one for each node. All *PUP*s at a particular node (waiting to be sent) will be shown in a high level representation.

4. Statistics Gathering

We recognize that a debugging package with all the capabilities described might also provide the basis for a powerful statistics gathering facility. Statistics gathering packages have been used extensively in the past to monitor traffic flow over the *Ethernet*, so there is nothing new about this concept. Such a package is currently absent from our system, however, and we hope to implement a statistics gatherer in parallel with, and indeed drawing from, the debugging package. We see this as representing perhaps 25% to 35% of the thesis work.

5. Background

We are not aware of any work being done along the lines described here. Certainly, the literature on debugging is not sparse, and [1] presents a good summary of past and current trends in the area. [2] provides detailed documentation concerning the state of the current resident *Alto* (*Mesa*) debugger. However, we believe that we are the first to attack the problems of interprocess communication debugging in the way described.

Powerful user interfaces have been constructed, along the lines of presenting to the user a higher-level, more global view of a complex system (see [1], [3], [4] and [5]). However, these studies did not attack the particular area we have been discussing.

Facilities for presenting information in terms of the language that the user is programming in is discussed in [1] and [3]. As previously mentioned, this ought to prove useful for displaying of *PUP* contents for user examination.

Virtual timing facilities, and the ability to control system clocks to present virtual time environments to applications programs are not new concepts. They are discussed extensively in [6]. However, the Virtual Machine Emulator presented there is oriented to the conventional single CPU, single program concept. We hope to implement a more advanced version, which handles control over and synchronization of multiple virtual timers at multiple nodes.

Nor, as mentioned, is there anything new about statistics gathering packages for measurements of interprocess communications over the *Ethernet*. Detailed discussions of these may be found in [7] and [8].

6. Schedule of tasks

April - Decide what the detailed capabilities of a debugger and a statistics gatherer should be. Either complete, or finish majority of, main debugger module to intercept and display packets.

May - Decide on which road to take - Do we want an "air-traffic" screen or high level representation of *PUP*'s? Begin programming one or the other.

June - Assessment - Is it possible to do both of the above tasks? If not, should be at least half-way through the implementation of one of them. Should have most of statistics gatherer out of the way.

July - Begin writing thesis. Wrap up implementation and begin intensive debugging on typical applications programs that use the *Ethernet.*

August - Put finishing touches on project. Finish writing thesis.

References

[1] Monitoring System Behavior in a Complex Computational Environment; Model; Xerox PARC; 1979.

[2] *Mesa* Debugger Documentation; Xerox PARC; 1979.

[3] Displaying Data Structures for Interactive Debugging; Myers; M.I.T. Master's Thesis; To Appear.

[4] RIG, Rochester's Intelligent Gateway: System Overview; Ball, Feldman, Low, Rashid, Rovner; University of Rochester Department of Computer Science; 1976.

[5] VTMS: A Virtual Terminal Management System for RIG; Lantz, Rashid; University of Rochester Department of Computer Science; 1979.

[6] A Virtual Machine Emulator for Performance Evaluation; Canon,Fritz, Howard, Howell, Mitoma, Rodriguez-Rosell; IBM Research Laboratory.

[7] Performance of an *Ethernet* Local Network - A Preliminary Report; Shoch, Hupp; Xerox PARC; 1980.

[8] METRIC - A Kernel Instrumentation System for Distributed Environments; McDaniel; Xerox PARC; 1977.

[9] *Ethernet*: Distributed Packet Switching for Local Computer Networks; Metcalfe, Boggs; Xerox PARC; 1976.

[10] *Mesa* Language Manual Version 5.0; Mitchell, Maybury, Sweet; Xerox PARC; 1979.

[11] *PUP*: An Internetwork Architecture; Boggs, Shoch, Taft, Metcalfe; Xerox PARC; 1979.

[12] *Mesa PUP* Package Functional Specification; Xerox PARC; 1979.

[13] A Field Guide to *Alto*-Land; Levin; Xerox PARC; 1979.