Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

I.J.P. Elshoff

A distributed debugger for Amoeba

Computer Science/Department of Algorithmics & Architecture Report CS-R8828 July

69C24, 69D25, 69D55

A Distributed Debugger for Amoeba

I.J.P. Elshoff *
Centrum voor Wiskunde en Informatica
Amsterdam

5 May, 1988

Abstract

We describe a debugger that is being developed for distributed programs in Amoeba. A major goal in our work is to make the debugger independent of the Amoeba kernel. Our design integrates many facilities found in other debuggers, such as execution replay, breakpointing, and an event-based view of the execution of the target program. This paper discusses the influence of Amoeba's architecture on the attainability of our goals and the desired functionality of the debugger. We also consider such problems as how to deal with timeouts and interactions between the target program and its environment.

CR Categories: C.2.4, D.2.5, D.4.5

Keywords & Phrases: Amoeba, distributed debugging, program monitoring, execution replay, timeouts.

Note: This paper was presented at the ACM Workshop on Parallel and Distributed Debugging, 5-6 May, 1988, Madison Wisconsin, U.S.A.

1 Introduction

Amoeba is a capability-based, distributed operating system being developed at the Centre for Mathematics and Computer Science (CWI) and the Free University (VU) in Amsterdam [8,9]. It is far enough along to need tools such as a debugger so that Amoeba programmers have some help when things go wrong. This paper describes such a debugger; however, many of the lessons can be applied to distributed debugging in general.

One of the primary goals for the debugger is that it is totally independent of the implementation of the Amoeba kernel. The Amoeba kernel is a small, monolithic part of the system with a well-defined user interface. Adding hooks for a debugger would complicate the kernel and make the two components heavily dependent on each other. It is too tempting to build in intimate knowledge about the implementation of the kernel if the debugger has access to its internal data structures. In addition, since Amoeba is still in the development stage, we would like to be able to completely reimplement the kernel without having to rewrite support tools such as the debugger (provided, of course, that the semantics and interface of the kernel do not change).

*The author's present address is: Department of Computer Science, University of Arizona, Tucson AZ 85721, U.S.A. (`irv@arizona.edu`) +1-602-621-6612.

Report CS-R8828
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

The Amoeba debugger takes an event-based approach [2,10] and views the user's (target) program as a generator of a stream of events. Events correspond to things like sending a message, creating a task, hitting a breakpoint, and dividing by zero. Since the number of events generated can become quite large, the debugger provides filtering and abstraction mechanisms to make the event stream more manageable.

However, a purely event-based approach is not enough; some access to the state of the program must also be provided. To this end the Amoeba debugger provides many of the facilities found in sequential debuggers: the ability to examine and alter memory, obtain a stack trace of procedure calls, set breakpoints, and to deal with the program on a source-level basis (i.e., refer to statements, user-defined types, variables, etc.). Nonetheless, the interesting aspects of the debugger are those that deal with the distributed nature of the target program; in this paper we will mostly ignore the sequential aspects.

The debugger attempts to provide an execution replay facility along the lines of Instant Replay [7]. Instant Replay is a program monitoring facility that faithfully reproduces a distributed program's execution sequence. It does this by remembering the program's interactions with its environment and the order in which significant events occur. Unfortunately, the existence of an asynchronous primitive in Amoeba — cluster signaling — spoils the chance to use their techniques to construct a comprehensive replay facility. At best we can only replay the execution sequence of the target program if cluster signaling is not used.

While the user may not always be able to replay the exact execution sequence, he can manually control the execution order by choosing which process to run next when doing a context switch. This gives him fine grain control over scheduling, but unfortunately does not scale well to large numbers of processes (doing frequent context switches).

It is our philosophy that debugging is a specific phase of the program *development* cycle. We do not support the constant background monitoring of believed-to-be-correct ("production") programs. If such a program aborts during its execution, a separate postmortem analysis tool can be used to shed enough light on the nature of the problem so that the user can go back with the debugger and explore the conditions that caused the failure. This tool can analyze the state of the program at the time of the crash, but does not provide any event history information. Even the best monitoring mechanisms incur some overhead. We would like production programs to be as fast as possible, so *no* debugger support is added (this is like turning off run-time subscript checking in languages that support it).

The debugger must know what state a target program is in at all times. There are two approaches to this: it can cooperate with the operating system to share this information, or it can maintain its own idea of the state of a target program. Since the debugger is kernel independent we must use the latter approach. There is a performance penalty to be paid for this decision but it is not really important. The whole point of a debugger is to take things slowly and at a human pace so that the execution can be scrutinized; lightning fast execution is not really the goal. (This does not mean that the debugger should be inefficient; as with any utility, a reasonable amount of effort should go into making it as efficient as possible given the available system services and desired functionality.)

2 An Overview of Amoeba

In Amoeba, a *program* is a collection of clusters which interact with their environment through messages. A *cluster* is an address space along with a collection of one or more threads, called *tasks*. An address space consists of a collection of *segments*, which are contiguous blocks of memory with a particular set of access rights. The environment of a program is the set of standard services that it uses (e.g., a file server, directory (yellow pages) server, and time server). The debugger is also considered such a service — it is not part of the target program or the Amoeba kernel. Each service is also a program, although from the debugger's perspective it is just a black box that generates and responds to requests.

Amoeba is a capability-based system. Knowledge of a *capability* is prima facie evidence that the holder has the right to access the object represented by it. The sparsity of the name space makes it arbitrarily hard to guess capabilities (but not impossible). There are capabilities for system objects, such as clusters and segments, as well as higher-level objects, such as files and programs. However, Amoeba does not give tasks capabilities,[1] or any other type of name for that matter. And since there is no system-wide standard for naming tasks, the debugger must make them up as it goes (see Section 3).

Tasks can communicate in several ways, and communication is what makes distributed debugging an interesting problem. Any two tasks can exchange messages. Tasks within the same cluster can communicate through shared memory or by synchronizing with binary semaphores and condition variables. Lastly, a task can signal any cluster it has a capability for. A signal is an asynchronous event that interrupts a cluster by inducing an artificial exception in it.

Message communication is based on transactions (request/reply pairs) and goes through ports; a *port* is identified by a string of 48 bits.[2] When a task sends a message (with the **trans** system call), it will be blocked until the message has been received and a reply returned. Many tasks can listen to the same port, but a message will be delivered to at most one task. When a task wants to receive a message (using **getreq**), it blocks until one arrives. When it wakes up, it can process the message, but must send a reply (with **putrep**) before it can ask for the next message. Amoeba does not have a non-blocking message passing mechanism; the need for this is subsumed by the existence of light-weight tasks.

When a cluster receives a *signal* or causes an *exception*, all of the tasks in it are frozen and a cluster descriptor (containing segment capabilities and task states) is sent to a *handler.* The descriptor is delivered to the handler as an ordinary message. The handler can examine and modify the cluster's segments and the states of its tasks. It can also remove tasks and segments, or add new ones. Once the handler is finished, it can return a (possibly modified) cluster descriptor to the kernel, which will reactivate the cluster. The handler can also hand back a different descriptor, in which case a new cluster will be started in place of the old. Using this mechanism, the debugger can implement a checkpoint and restart facility. Normally, the debugger will be the handler for a cluster, but in the **signal** call it is possible to specify a different handler. For example, the shell can use this mechanism to perform job control.

An important attribute of the current implementation of Amoeba is that tasks *within* a cluster are scheduled non-preemptively and with mutual exclusion. This means that context switches occur only during system calls that block, and tasks have exclusive access the cluster's address space while they are running, so shared memory need only be protected if there are blocking system calls within a critical region. Tasks in *different* clusters are scheduled preemptively on a round-robin, time-slicing basis. And, of course, tasks in clusters on different machines execute with real parallelism, so our scheduling assumption does not reduce the problem to debugging pure coroutines (a sequential situation).

A version of Amoeba is being developed for a multiprocessor where tasks within a cluster can execute with true parallelism. In such an environment, truly concurrent access to shared memory

[1]Tasks are intended to be very light-weight objects. Giving them a capability would only weigh them down.

[2]For the purposes of this paper we will ignore the fact that in Amoeba a port has two components: a *put port* and a *get port*. This is a security mechanism to ensure that when a task gives out a port it can do so without also giving away the right to read from that port.

causes problems since such access can occur completely without kernel intervention. Furthermore, we cannot even assume that the user will properly protect access to shared memory. The lack of such protection may, in fact, be the bug the user is looking for. One of the consequences of this problem is that it will be impossible to implement an execution replay facility for a multiprocessor version of Amoeba.

3 The Debugger

The Amoeba debugger can be broken down into three levels. At the bottom is the *monitor*, which monitors the behavior of the target program and generates events when appropriate. Above that comes a layer of tasks, one for each cluster in the target. Each such task is called a *correspondent*; it receives the stream of events from the corresponding cluster in the target program. Finally, at the top is the user interface, which is window-based and allows the user to examine and control individual clusters and tasks, as well as deal with the target program as a whole (e.g., to make checkpoints or initiate replay).

As advocated by Joyce [6], the monitoring aspect of the debugger has been separated from the analysis and control aspects. The monitoring component is very simple; it is realized by a special library, linked in with the target program, that replaces the standard stubs for all system calls. Whenever the target invokes a system service, the stub sends a message to the correspondent announcing the event. In most cases an immediate reply is returned. If the user wants to delay the task, the debugger will hold off sending the reply. (In this case it is possible that Amoeba will schedule another task in the same cluster, which might change the state of the stopped task.)

Once the correspondent replies to the event message, it is said to have *released* the event, and the target program goes on to invoke the system service. If the call is potentially blocking, another event is generated after it completes. The correspondent uses this second event to update its scheduling information (a context switch may have occurred); it is also visible to the user so that he can tell exactly when each blocking call wakes up.

An *event* is a 4-tuple with the following components: an event type, the identity of the task generating the event, the capability of the cluster containing the task, and the arguments of the event. These arguments depend on the event type. For events related to message passing the argument is the header and body of the message. For semaphore events it is the name of the semaphore. For segment management events it is the capability of the segment in question, and so on.

A task's identity is assigned by the debugger when it is created and has no meaning outside of the cluster (correspondent) containing the task. Beyond that, a task is specified by a (cluster capability, task name) pair. A task has the name "*proc–i*" if it is the *i*th task to start in the procedure named *proc*. When a new program is started the root task is called "`main-0`."[3] These names may not always be appropriate, so the debugger allows the user to rename tasks with more meaningful names.

Even though Amoeba has names for clusters, capabilities are most often a seemingly random collection of bits and hence do not have a meaningful representation for the user. For this reason, the debugger also allows the user to give symbolic names to cluster capabilities. These can be used wherever capabilities are called for.

In Amoeba, when a task wants to read a message from a port, and there is more than one potential sender, it does not know in advance which sender's message it will receive. One type of bug might be that the wrong two tasks rendezvous during a message exchange. Since Amoeba does not have task names, it cannot provide the identity of the sender — it is up to the sender to do this. The

[3] The debugger is targeted for C programs, which begin executing in the function called `main`.

debugger does, however, keep track of which send corresponds to a receive. In fact, it goes much further by maintaining cause-and-effect links between all appropriate events. For example, the event representing the unblocking of a **P** is linked to the **V** that made it possible. This enables the user to see exactly why certain events occur when they do. Such information is normally not available to the Amoeba programmer. We describe how this is implemented in Section 3.6.

3.1 Filters and Recognizers

The stream of events generated by the target program can be quite voluminous. This creates two problems: it may not be possible to store all of these events in a log, and we need a way to select and analyze them for the user. Presenting a raw stream of events to the user is akin to giving him a memory dump — there is a wealth of information, but interpreting it is very difficult.

We provide two mechanisms to overcome these problems: filters and recognizers. Filters make it possible to weed out events that are of no interest to the user; these front-line filters are called *global filters*. In any program being debugged, every system call always generates an event so that the debugger can maintain state and scheduling information. However, only filtered events get passed on to the subsequent stages of the debugger, which include the recognizers and user display routines.

A *recognizer* searches for a sequence of events. It consists of a *local filter*, a finite state machine (FSM), and a command list. When an event has passed through the global filters it is given to each recognizer. If it also passes through the local filter, it is used as input to the FSM. If it causes a move and the new state is a final state, the associated command list is executed. If no move is possible, the recognizer silently resets by moving to the initial state. The FSM does not tolerate "noise" — it depends on the local filter to get rid of irrelevant events.

The finite state machine implements an extended form of regular expression, whose primitives are patterns for individual events. The operators are concatenation, alternation, repetition, and permutation. Permutation is like concatenation, except that the constituent subsequences can appear in any order. For example, "`[abc]`" means that `a`, `b`, and `c` can appear in any order, and is equivalent to "`(abc|acb|bac|bca|cab|cba)`". Unlike concatenation, permutation is an n-ary operator. Since it can be expressed in terms of concatenation and alternation, it is merely syntactic sugar, but quite convenient.

There are other possible ways to specify sequences: simple subsequences, context-free grammars, and interval logic [5] are but a few. Simple subsequences are not powerful enough, and context-free grammars, while being very expressive, are too cumbersome to specify. Regular expressions come somewhere in between. Furthermore, many statements in interval logic are hard to evaluate on-line. We can use well-known and efficient techniques for searching for sequences described by regular expressions.

Once a recognizer has found an event sequence, a list of debugger commands is executed. The command list may contain any of the commands to the user interface, including one to insert a new event into the stream. In this way the events in the sequence can be combined into a single higher-level event, and a hierarchy of event abstractions can be built up. Higher-level events are treated identically to lower-level ones by the debugger; they are only higher-level in the user's mind. More than one recognizer can be triggered by a given event. The command lists are executed one-by-one, and the order can be defined by the user. Only when all eligible recognizers have executed their command lists will the event that triggered the eligibility be released.

Several distributed debuggers, such as IDD [5] and MuTEAM [1], constantly check to see that no user-specified assertion about the target program is violated. If they detect a violation, the user is notified. We provide this by giving the user a special form of recognizer, called a *verifier*. Verifiers are like recognizers, except that when their FSM's are not in their initial states and they see an event that is not expected, they execute the command list rather than silently going back to the initial state. Only when they reach a final state do they reset. A typical command list would stop the target and print a message for the user. Using this mechanism we can ensure that sequences of events in which we are interested are well-formed; this is often easier than trying to recognize all possible bad behaviors.

Both filters and recognizers use patterns to select events. A *pattern* matches zero or more events and is constructed from logical combinations of primitive patterns (i.e., using conjunction, disjunction, and negation). Primitive patterns are 4-tuples, like events, but in addition to allowing fixed values in their fields, they can have wildcards of various types. The event type field can contain a name of an individual event type (e.g., create a cluster), or the name of a class of event types (e.g., any cluster-related event). The task and cluster identification fields can contain lists of tasks or clusters. The argument field for non-message-related events can contain a list of values to match (e.g., a list of semaphores for **P** events). For message-related events, the argument field can contain a source-level expression referring to the header and body of the message (or reply) in question. Finally, each field of a primitive pattern may also contain a universal wildcard, which matches any value in the corresponding field of an event. A primitive pattern matches an event if and only if each field of the event is matched by the corresponding field of the pattern.

Filters, recognizers, and patterns can be read from (or written to) a file[4] as well as entered directly by the user. This makes it easy to reuse them between debugging sessions or make minor modifications with a text editor. All debugger objects have user-assigned character-string names for easy reference. Furthermore, filters and recognizers can be turned on or off at will with a simple command. This gives the user detailed control over the debugger; e.g., a recognizer can disable itself or automatically turn on a filter at a particular point in the execution. Finally, it is possible to write the event log to a file for later examination with a text editor or other tool.

3.2 Execution Replay

Ignoring for the moment Amoeba's signaling mechanism, context switches within a cluster can only take place during blocking system calls. Furthermore, the debugger is notified before and after every such call, so it can collect enough information to replay the execution of a program *provided the environment reacts exactly the same.* This last requirement is not trivial to satisfy. Since the debugger knows nothing about the behavior of the environment, it cannot assume that it will indeed react the same. The debugger must keep track of all message traffic involving the outside world as well as the internal scheduling sequence.

An execution replay facility obviously has two phases: recording and replaying. Because one cluster can only affect another through a system call, we can do the recording locally at each cluster. Furthermore, inter-cluster context switches (which occur without the knowledge of the debugger) do not cause problems. During the recording phase, we adopt the approach of Instant Replay and only note the scheduling order of events for each cluster, not the data (e.g., message bodies) associated with them. These events are maintained in a sequential log.[5] During replay, the debugger knows which event to expect next. If that event occurs, it is immediately released. However, if a different event occurs, the debugger holds off releasing it. Later, when the event is supposed to happen, it will be released. In the mean time, Amoeba will schedule another task in the cluster. This process

[4]In general, the input to the user interface can be temporarily attached to a file so that frequently executed (lists of) commands need not be entered every time. Other paradigms (e.g., graphical) for referring to oft-used objects are also being investigated.

[5]Because of the non-preemptive nature of the scheduler, events within a cluster are totally ordered.

continues until a task is scheduled that generates the expected event.[6]

Part of the recording phase is deciding which events are internal and which are interactions with the environment. The only possible interactions with the environment are through messages,[7] so we need only consider those events dealing with messages. When an event representing the sending or receiving of a message occurs (specifically, after the `trans` or `getreq` completes) the debugger looks for a matching event in another task. If one cannot be found the event involves the environment. In this case the entire message or reply buffer is saved in the event log. Later, when this event is encountered during replay, the environment is simulated using the stored data instead of actually starting a transaction or asking for a request.

The presence of the cluster signaling mechanism in Amoeba foils attempts at making a comprehensive replay facility. With the exception of signals, an Amoeba program interacts with its environment actively (i.e., only when *it* wants to) and with the knowledge of the debugger. This makes it possible to record the whole scheduling sequence and play it back at the right moments. But signals are asynchronous events that can occur at any time; the target cluster reacts passively. In order to replay them correctly, we need to know exactly when they occur with respect to the execution of the cluster receiving them. This is not possible in a debugger without extensive kernel support.

There is still not much practical experience with signaling in Amoeba. Signals are part of the system primarily to support migration, checkpointing, job control, and debugging, usually all at the behest of the shell. It may be the case that most programs do not use signals; if so, it may be acceptable for the replay facility not to be comprehensive, as long as the debugger warns the user whenever it cannot correctly satisfy his request.

3.3 Checkpointing and Rollback

In many cases the user does not want to have to replay the entire execution once the point at which a bug occurs has been narrowed down. We provide a mechanism to save the intermediate state of a program (checkpointing) and reinstate it later (rollback). Rather than replaying everything, which might take quite a long time if the bug does not manifest itself immediately, the user can roll back to a recently checkpointed state and start the replay from there. This is also useful if the user wants to return to an earlier state so that he can explore a *different* execution sequence. Amoeba provides for the checkpointing of individual clusters using the signal mechanism. When a cluster is signaled, it is frozen at once and as a whole.[8] Simultaneously freezing a collection of clusters is not possible, however.

Checkpointing an Amoeba program is really an instance of the distributed snapshot problem. However, since Amoeba does not buffer messages (i.e., inter-cluster communication is synchronous), it is possible to make a consistent snapshot without using a marker-type algorithm [3]. The debugger can simply signal the clusters of the target program one by one. If a running cluster attempts a transaction with a frozen one it will simply block (which might have happened anyway). If a cluster is servicing a request when it is signaled, the client cluster will just continue to block until it too is signaled. If a running cluster replies to a frozen one, the reply will be remembered in the client cluster's descriptor so that when it comes back it will be able to proceed. Once all of the clusters have been frozen, their images are written to a file for storage and a marker is placed in the event log indicating where the checkpoint occurred. Finally, the debugger can resume each cluster; they will

[6] If no such event is generated, the replay is unsuccessful. This is not a bug in the debugger — it is an indication that a cluster was signaled.

[7] And cluster signals, but signaling a cluster in the environment is assumed not to happen. Since the user has the capability for a cluster in the environment if he can signal it, he can also "move" it into the target program.

[8] Tasks executing nonblocking system calls are allowed to finish the call. Tasks in a blocking call are allowed to reach an appropriate point within the call before being frozen.

continue as if uninterrupted.

To do a rollback the debugger first destroys the existing clusters in the target program. Then it creates new ones by loading in the previously stored images, and sets them running. The event logs are also rolled back, although the "future" is kept if a replay is desired. This whole scheme works because in Amoeba the image of a cluster (i.e., its descriptor and copies of all segments) contains the entire state of the cluster while it is frozen. Nothing is remembered by the kernel.

There are two problems with this mechanism: timeouts and the environment. If the signaling process is not fast enough, it is possible that a running cluster times out because a frozen cluster does not act in time. See Section 3.5 for more discussion on timeouts. If the target program is checkpointed while engaged in a transaction with the environment it will not be able to roll back to that state since the debugger has no control over the environment. Therefore, the debugger will not allow a checkpoint to be done if any task in the target program is blocked in a **trans** with a server in the environment or servicing a request from the environment (i.e., executing between a **getreq** and the corresponding **putrep**). Because the debugger will not know that a **trans** does not involve the environment until it sees a matching **getreq** in another task, a checkpoint will be prohibited any time there is a **trans** that is not being serviced by another part of the target program. However, since it is most likely that servers in Amoeba will do a **getreq** before (or soon after) their clients do a **trans**, this limitation should not stand in the user's way.

3.4 Breakpointing

Breakpoints are implemented using the conventional technique of replacing the instruction at the breakpoint address with a trap instruction. However, breakpointing in Amoeba is much more complicated than in the sequential world. Firstly, there is a fundamental problem with halting only part of a distributed program — see the following Section for a thorough discussion. Secondly, tasks in a cluster can share code; yet we may not want every task that runs into a trap to be breakpointed. Perhaps we are only interested in one member of a task family. When the trap is executed an exception occurs, freezing the cluster. Do we want the whole cluster to stop on a breakpoint, or just the task that hit it? There are arguments for both, so we provide both. The user can either set breakpoints on whole clusters, families of tasks, or individual tasks.

Implementing the breakpointing of an entire cluster is done as follows: When a trap has been executed and the running task has a breakpoint at that address, the cluster is immediately frozen. When the user is ready to let it continue, the debugger replaces the original instruction and turns on the trace bit. Then the cluster is resumed; the trapped task proceeds to execute the original instruction,[9] which causes another exception because of the trace bit. The debugger catches this, puts back the trap instruction, turns off the trace bit, and lets the cluster continue.

When a trap has been executed, but we do not want to stop the whole cluster, the debugger removes the breakpointed task from the cluster descriptor during the handling of the exception caused by the trap. Then it immediately lets the rest of the cluster continue executing. When the user is ready to have the stopped task proceed, the debugger signals the task's cluster, reinserts the task in the cluster descriptor, and uses the method outlined above to step it over the trap without letting other tasks sneak up and pass through as well.

From the user's point of view, when a breakpoint is reached an event is generated. If the user wants to stop the task (or cluster), he can install a single-event recognizer with a "`stop`" or "`stop cluster`" command. However, he may only want to examine a variable, or just know that the breakpoint was reached (i.e., *trace* the program). The command list for the recognizer can be used

[9] A critical assumption is that no context switch occurs in the target cluster during the handling of an exception.

to tailor the actual response to a breakpoint. A predefined recognizer to stop at all breakpoints can be turned on, freeing the user from having to give one explicitly. Since the breakpoint event is not released until the command list terminates, the cluster is frozen while the commands are carried out.

3.5 Halting the Target Program

Being able to halt the target program is an important part of debugging. In the sequential world the whole target is either running or halted. When the target is halted the debugger has control, and vice-versa; the two are coroutines. In a distributed domain, the debugger runs concurrently with the target, and some parts of the target may be running while others are halted. Each task can be made runnable or halted independently. A task is runnable when it is actually running or busy with a (possibly blocking) system call. It is halted when the user has interrupted its execution; e.g., with a breakpoint or while single stepping. In addition, whole clusters can become halted if a constituent task causes an exception; e.g., by dividing by zero or attempting to reference nonexistent memory.

The fact that a program can be partially halted creates some special problems when timeouts are involved. If one task is halted, but another that is waiting on it times out because the first task does not respond, an abnormal execution path may be taken. The problem is compounded by the fact that the debugger has no way of knowing what a task might be waiting for when it gets a timeout. Amoeba is based on a client-server paradigm; in general, Amoeba clients do not know who their servers will be.

Remote procedure call (RPC) based systems do not have this problem. The caller (client) always knows where the procedure (server) can be found, and servers never wait for (and hence timeout on) clients. Given these constraints, it is possible for a debugger for an RPC-based system to tell if a timeout is due to a halted task. Pilgrim is a good example [4]. If a timeout is due to a halted task, Pilgrim will automatically adjust the timeout period to compensate for the time the task was (is) halted. It does so by maintaining a logical clock for each task in such a way that they cannot see the period in which they are halted.

Since we cannot predict the recipient of a message in Amoeba, the debugger has no way of knowing whether a timeout was meant to happen — logical clocks are no help. And even if we could solve the problem *within* a target program, we still have to worry about timeouts in tasks *outside* of the program, i.e., somewhere in the environment. If one occurs in a file server, for example, the execution future may be irrevocably altered. The debugger has absolutely no control over the environment,[10] so it cannot guarantee that it will not react adversely to a client being debugged.

Given all of these fundamental problems, how *can* timeouts be dealt with sanely? We leave it up to the user, since he knows much more about his program than the debugger. When an event is generated and the system call it represents has a timeout set, the user can be notified and given the opportunity to change or abandon the timeout. When a timeout actually occurs, another event is generated. The user can accept the timeout, in which case the timed-out task goes on, or reject it, in which case the task repeats the system call. If he rejects it, a new timeout can specified, including no timeout. If the user intends to do the same for a wide range of tasks, he can use the recognizer facility to automatically apply his desires. The user interface makes it easy to disable every timeout, or change the duration of a timeout in a group tasks. Unfortunately, there is no way to deal with timeouts occurring in the environment.

[10] We consider Pilgrim's approach of modifying servers to handle clients that may be debugged to be unacceptable. Servers should not have to handle every possible condition in which their clients could find themselves (e.g., being debugged). Unfortunately, we do not have a viable alternative.

3.6 Cause-and-Effect Links

As mentioned earlier, the debugger maintains cause-and-effect links between related pairs of events. There are four such type of links:[11] (1) The event representing the unblocking of a **getreq** call is linked to the event for the start of a **trans**. (2) The event representing the unblocking of a **trans** call is linked to the **putrep** representing the sending of the reply. (3) The unblocking of a **P** is linked to the **V** that made it possible. And (4) the event representing the awakening of a task that was sleeping on a condition variable is linked to the event for the wakeup call. More precisely, when an event B occurs that was directly caused by an earlier event A, two links are established: an *effect* link from A to B and a *caused-by* link from B to A. The remainder of this section describes how the debugger makes these links.

The simplest case is that for condition variables. A task can wait on a condition variable by invoking the **sleep** system call. When another task calls the **wakeup** primitive on the same condition variable, all tasks sleeping on it are woken up. If no such task exists, the **wakeup** call has no effect. When an event representing the unblocking of a **sleep** occurs the debugger scans the event log in reverse to find the oldest unmatched **wakeup** on the same condition variable. These two events are linked. Because one **wakeup** can cause many sleeping tasks to awaken, a **wakeup** event can have many effect links.

Next we consider how to link **P**'s and **V**'s on binary semaphores. Recall that semaphores can only be used by tasks in the same cluster and that the event sub-log for each cluster is totally ordered. The debugger keeps track of the value of a semaphore and which tasks (if any) are blocked on it. If a **V** occurs on a semaphore with a value of one, it has no effect and is marked as unmatchable. If the value is zero and there are no waiting tasks, the value becomes one and the **V** event is given a null effect link. Otherwise, the task blocked on the semaphore the longest will eventually be allowed to proceed. When it does, the debugger will link the **V** and the event representing the unblocking of the **P**. If a **P** is done on a semaphore with value one, the task can proceed immediately and the subsequent "**P**-unblocking" event will be given a null caused-by link.

Finally we discuss how the various events related to message passing are linked. When a **getreq** call becomes unblocked, the event log is searched for an event representing the start of a transaction involving the same port and having the same message contents. If one is found the two events are linked. (If there are more than one, it does not matter which we choose.) If none is found, the debugger can conclude that the request came from the environment and the "**getreq**-unblocking" event is given a null caused-by link. When a **trans** call becomes unblocked and the associated "**trans**-start" has a non-null effect link, the "**trans**-unblocked" event is linked to the **putrep** event corresponding to the "**getreq**-unblocked" linked to the "**trans**-start". If the "**trans**-start" still has a null effect link at this point, the **trans** was serviced by a task in the environment.

4 Final Comments

One disadvantage of using a special library to implement the monitor is that it resides in the same address space as the target. Amoeba does not distinguish between tasks in a cluster, so any task can modify any writable memory segment. This means than an errant target program (not an uncommon situation for a debugger) can corrupt the data belonging to the event monitor. We can mitigate the problem by minimizing the number of variables used by the monitor, but it impossible to completely eliminate them.

In many distributed systems the ability to debug programs (particularly production programs)

[11] Actually, there are a few more, but they are of little pedagogical interest and are omitted for the sake of brevity.

results in a security problem. This is not the case in Amoeba. In order to debug a program, the user must have the capabilities for every cluster in it. System servers are part of the external environment and cannot be debugged (and hence compromised), since the user only has capabilities for the objects maintained by the server, not for the clusters that implement the service.

The fact that the debugger uses the same communication mechanism as the target program to report events (and the fact that it has no control over the scheduler) means that it is unavoidable that it could influence the execution of the target program. Some debuggers such as MuTEAM and Pilgrim go to great lengths to have no influence on the target's execution.[12] The cost of this is transparency is extensive cooperation with the language run-time support or kernel. Since our debugger is kernel independent, we must live with the idea that it can cause a different execution sequence. We only require that the debugger not cause executions that would otherwise not be possible given the nondeterminism introduced by network delays and arbitrary scheduling decisions. Of course, there is still the problem with timeouts causing an abnormal execution if part of the target program is halted for too long.

Nonblocking system calls are made blocking by the use of regular message passing primitives to generate events. Therefore, we must ensure that no context switches visible to the target program are done during these "nonblocking" calls. This can be accomplished with a semaphore that is released only when the target program itself invokes a blocking system call, and is reacquired after such a call returns.[13]

It should be mentioned that many consider debugging to be an impure activity. They believe that one's time is best spent producing a provably correct program (implementation) from a formal specification of the problem. For the most part, we agree with this philosophy. However, bugs can permeate all levels of the programming process — modeling, specification, implementation, and coding. Only when the actual behavior of a program differs from the *expected* behavior does the user know something is amiss. It is at this point that a good debugger is invaluable for determining exactly what went wrong.

Distributed debugging is a complicated activity. Parallelism and the lack of global control means that many of the things we would like to provide are either very difficult or impossible. Nonetheless, we believe that it is possible to construct a useful tool for distributed debugging, albeit not a utopian one. Debugging, distributed or otherwise, still requires the skill and insight of the user to be successful. A debugger is only a tool to make his job easier — it is not a panacea.

Acknowledgements

Thanks are due to Sape Mullender and Guido van Rossum for many interesting discussions on the present and future versions of Amoeba, as well as for comments on the debugger's design. Jack Jansen was instrumental in making the implementation effort possible. Greg Andrews provided many useful comments on an earlier version of this paper.

[12] Cooper admits that in Pilgrim it is not possible to completely isolate the program from seeing a different execution while being debugged. All he can hope do is make it very unlikely.

[13] The assumption that at most one task can be executing per cluster is critical.

References

[1] F. Baiardi, N. De Francisco, E. Matteoli, S. Stefanini, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *SIGPLAN Notices*, **18** (8), 98-106 (August 1983).

[2] Peter C. Bates and Jack C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *Journal of Systems & Software*, **3** (4), 255-264 (1983).

[3] K. Mani Chandy and Leslie Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, **3** (1), 63-75 (February 1985).

[4] Robert Cooper, "Pilgrim: A Debugger for Distributed Systems," *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin BRD, 458-465 (21-25 September 1987).

[5] Paul K. Harter, Dennis Heimbigner, and Roger King, "IDD: An Interactive Distributed Debugger," *Proceedings of the 5th International Conference on Distributed Computing Systems*, Denver Colorado, 498-506 (13-17 May 1985).

[6] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Programs," *ACM Transactions on Computer Systems*, **5** (2), 121-150 (May 1987).

[7] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, **36** (4), 471-482 (April 1987).

[8] Sape J. Mullender and A.S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal*, **29** (4), 289-300 (March 1986).

[9] Sape J. Mullender, et al, "Amoeba Kernel 4.0 Specification," Internal CWI document, (16 December 1987).

[10] Edward T. Smith, "A Debugger for Message-Based Processes," *Software Practice & Experience*, **15** (11), 1073-1086 (November 1985).