Technical Report

UCAM-CL-TR-119
ISSN 1476-2986

Number 119

Pilgrim: a debugger for distributed systems

Robert Cooper

July 1987

ISSN 1476-2986

Pilgrim: A Debugger for Distributed Systems*

Robert Cooper
Cambridge University Computer Laboratory
Corn Exchange St., Cambridge, CB2 3QG, England
31 July 1987

Pilgrim is a source-level debugger for Concurrent CLU programs which execute in a distributed environment. It integrates conventional debugging facilities with features for debugging remote procedure calls and critical region based process interactions. Pilgrim is unusual in that it functions on programs in the target environment under conditions of actual use. This has caused a trade-off between providing rich and detailed information to the programmer and avoiding any unwanted alteration to the computation being debugged. Another complication is debugging one client of a network server while avoiding interference with the server's other clients. A successful methodology for this case requires assistance from the server itself.

I am designing and implementing a debugger in order to investigate the debugging of concurrent and distributed programs. The kinds of programs I wish to debug are written in a high level language with type-checked facilities for process interaction and remote operations. They will execute on the nodes of a local computer network and interact with the other programs and services which exist on such a network.

I begin by presenting the important ideas behind the debugger and describing the environment in which it is to operate. After describing the debugger in outline, I present a more detailed analysis of the aspects of debugging concerned with concurrency and distribution and how they are handled by the debugger.

1. Motivation.

There are three themes motivating the design of the debugger. The first is the employment of a high level language. While the hardware technology of distributed computing has existed for a decade, the software technology has been much slower to develop. The rise of language level

*Presented at the 7th International Conference on Distributed Computing Systems, Berlin, 21-25 September 1987.

remote procedure call (RPC) was the first important step. The next ought to be source level debugging of distributed programs. By this I mean that all components of the program's state should be accessible, and where appropriate modifiable, in source language terms. Besides the obvious convenience of debugging at the source level, many of the techniques for concurrent and distributed debugging described in this paper would be impossible without information about the types of variables and objects in the program.

The second theme is that of debugging distributed programs in their target environment under real conditions. Despite the best efforts during program design and development, many bugs will remain even after significant and systematic testing in a debugging environment. Hence the need for debugging tools which operate on programs under conditions of actual use, and perhaps after those programs have gone into service. This is especially so with distributed programming where the surrounding environment may be large, complex and (regrettably) not fully specified and thus impossible to simulate in all its detail. Consequently a program should not have to be recompiled, relinked or restarted in some special "debug mode" in order to debug it. The debugger must be able to operate on the normal code generated by the compiler and any debugging support included in the object program must not adversely affect the program's performance when it is not under control of the debugger. It is the generally unacceptable performance of debugging support which encourages programmers to leave out that support once "all the bugs are out".

The third theme is the maintenance of time consistency while debugging. The programmer wishes to slow down or interrupt the execution of the program to examine its workings more closely. But if as a result the program itself perceives time progressing faster or different processes execute at different rates, a faithful execution of the program will not occur. The correctness of a concurrent program should rarely depend on its rate of execution but faulty programs can be expected to contain timing errors of just this kind. The debugger in concert with other elements in the distributed environment must maintain a consistent and preferably unaltered time scale for the user program. Otherwise the symptoms of the bug under study may disappear or another bug may reveal itself and confuse matters.

The ideas of this section are embodied in the design of the Pilgrim debugger. Implementation of Pilgrim is nearly complete. Its use will test the efficacy of this approach to debugging.

2. The Mayflower environment and Concurrent CLU.

Pilgrim is a debugger for Concurrent CLU programs. CLU [Liskov 81] is a sequential language which provides very good support for user defined abstract data types and modular programming. It has been extended at Cambridge with light-weight processes and RPC to produce Concurrent CLU. (Concurrent CLU should not be confused with the Argus language, developed at MIT and based on CLU, which supports user defined, atomic, distributed objects.) In Concurrent CLU the processes at one node of a distributed program share memory. Process interactions are mediated by monitors, critical regions [Cooper 85], and semaphores, although the language does not enforce mutually exclusive access to memory.

Communication between nodes takes place via RPC. The RPC mechanism [Hamilton 84] is fully type-checked and permits arbitrarily complex objects of user defined type to be transmitted between nodes. Two RPC protocols are supported: the *exactly-once* protocol provides reliable communication in the absence of node failures while the faster, less reliable *maybe* protocol allows the programmer to handle both transient errors and failures with retry strategies appropriate to the application at hand.

Each node of a Concurrent CLU program runs on a single processor 8MHz MC68000 system connected to a Cambridge Ring, and executes under the Mayflower supervisor, a small operating system which supports multiple light-weight processes. Mayflower makes use of many of the servers which comprise the Cambridge Distributed Computing System [Needham 82].

3. Overview of debugger structure.

Pilgrim is itself a distributed program. Every node of a user program has a piece of debugging support code, called the agent, included in it by the linker. The agents remain dormant until the debugger proper, running on another node, connects to the user program to begin a debugging session. A debugger can be connected to a node at any time to investigate a suspected bug. At the end of a debugging session the debugger may be disconnected and the node continue executing. This is usually unwise if changes have been made to the contents of variables etc. because correctness of the node's operation is easily compromised.

To fulfill the goals described in Section 1 the agent code must be small and impose little or no overhead on the program when not connected to the debugger. This is one of the factors in deciding whether to implement some function in the agent or the debugger itself. For instance all activities involving the user interface, type-checking, and access to the source-to-object mapping information produced by the compiler and linker are performed in the debugger proper.

Some functions are logically required in the agent. These include memory access and the lowest level handling of hardware traps which are necessary in any debugging mechanism. In Pilgrim the agent must also handle the initial connection to the debugger and recover from failures of the connection or of the debugger itself. A session identifier (a unique but guessable number) is generated at the beginning of a debugging session and must be supplied upon all interactions between debugger and agent. The agent uses no timeouts when communicating with the debugger. Instead it is possible for a second debugger to forcibly connect to the agent – a process which results in the original session being abandoned and all breakpoints etc. cleared.

The functions provided by the agent would constitute a large security hole in any system. Authentication of the debugger and encryption of debugger-agent communication have been ignored in Pilgrim but should be addressed in any debugging system in a non-research environment.

Some functions while not necessary in the agent, would involve much more effort to implement in the debugger and would be very inefficient. For instance breakpoints could be achieved with many invocations of a low-level memory access primitive and some means of freezing all the processes on a node. Instead three less primitive operations are defined, one to set a breakpoint at a machine address, one to clear a breakpoint, and one to step a process over a breakpoint it has encountered. Similarly another agent function will invoke a procedure in the user program and return pointers to any results it returns. The dominant cost in most of the functions provided by the agent is the round-trip delay in communicating with the debugger. Expressing each logical request from the debugger as a single network interaction improves the overall performance.

A less obvious function provided by the agent is the display of objects in the user program. CLU encourages programmers to write print operations for their user defined types. The built-in types such as int and record also have print operations. These print operations are what the debugger uses to display the contents of variables etc. Since the print operations will be different for different programs the print operations must reside in the user program and be invoked by the agent. In Pilgrim the agent's procedure invocation mechanism is used, combined with a special output stream implementation which redirects output strings to the debugger.

Support for RPC and concurrent processes are two important parts of the agent. These are covered in the following sections which look at debugging issues specific to distribution and concurrency.

4. Debugging Remote Procedure Calls.

This section focuses on the facilities for debugging remote procedure calls, and how they are implemented in Pilgrim.

4.1 Debugging requirements of RPC.

One aim of the RPC support is to preserve the impression of RPCs as very similar to local procedure calls. This is achieved by providing stack backtraces which cross node boundaries, and avoiding any "mode-switches" when the user accesses code or data on multiple nodes. RPC is different from local procedure call because failures and retries may be visible to the programmer.

When using the debugger the programmer may wish to know why one RPC failed, or how many retries have been made while performing another. The failure of a call performed with the *maybe* RPC protocol could be due to either the call or reply packet being lost. The debugger ought to allow the programmer to find out which is the case.

The requirements of distributed debugging, as just described, are relatively straightforward; the main difficulty lies in the implementation. The debugger requires information about in-progress and recently completed RPCs. A lot of mechanism is needed to provide this and the implementation must not compromise the speed of RPC when the debugging support is not being used.

4.2 A first attempt at RPC support.

I was keen to decouple as much as possible the agent code from the rest of the Concurrent CLU runtime library to make it less susceptible to changes in the library. One way to achieve this, for the RPC support, was to monitor all RPC packets through a hook in the network device driver. A state machine would be maintained for each in-progress RPC and this would provide the information needed by the debugger. Since the RPC protocol itself is quite stable, and less likely to change than the library code which implements it, this approach looked promising. It became clear however that the work performed in the RPC debugging support would be of the same order as that in the RPC implementation itself. Thus RPCs might take twice as long when under control of the debugger. This was unacceptable.

4.3 The final Pilgrim implementation.

I decided instead to modify the RPC system directly. In the Mayflower RPC implementation most of the information needed by the debugger is already collected in the data structures maintained by the RPC runtime system. For instance the call identifiers (which uniquely name a particular invocation of a remote procedure) and the client process issuing the call are associated by a table in the client RPC mechanism. A similar table in the server associates the server process handling the call with the call identifier.

Some necessary information can be found only in local variables of a procedure stack frame of the RPC mechanism. This frame will either be at the top of the client process's stack, or at the bottom of the server process's stack (see Figure 1). I added an extra variable to these procedures in a known position in the stack frame. This variable points to an information block containing the process identifier, the remote procedure name, the call identifier, and an enumeration giving the current state of the protocol.

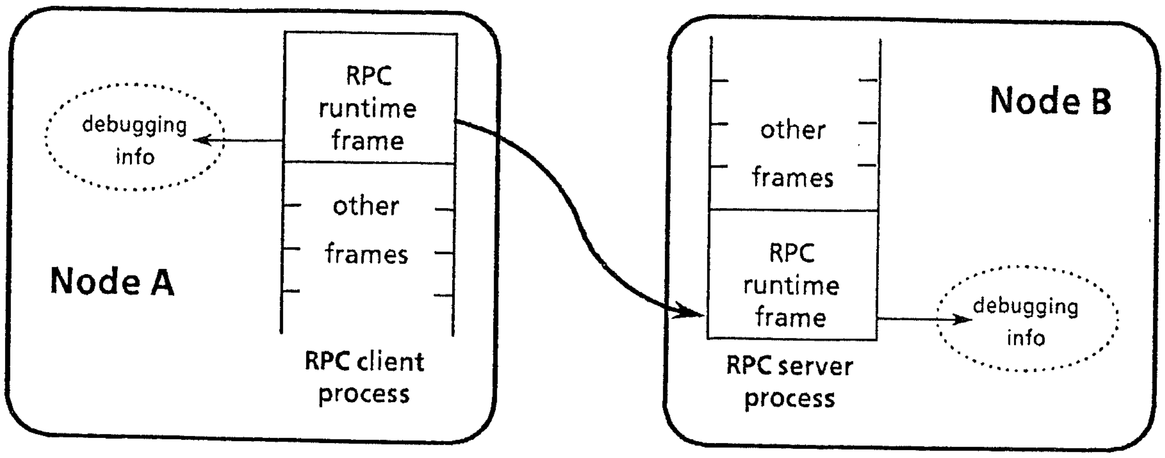

Figure 1. Finding RPC debugging information.

The data described so far concerns only in-progress calls. To aid in the debugging of recently completed or failed calls I added a ten-slot cyclic buffer describing the outcome of ten most recent RPCs. The only information maintained is the call identifier and whether the call failed or succeeded.

The effect of these changes to the RPC mechanism is to increase the time for an RPC by 400μs. For a null RPC (calling a remote procedure which has no arguments or results) this represents a slow-down by 2.5%. On more typical RPCs the slow-down is much less.

Similar techniques to those used here will be applicable to other RPC systems such as the Birrell-Nelson RPC system for Cedar [Birrell 84], and indeed should be easier to realize. In general the more failure recovery and orphan detection provided by an RPC system, the more information useful for debugging will be maintained by the RPC implementation and the less extra work will be necessary by the debugger. The main difficulty for Pilgrim was to find information about maybe RPCs for which the RPC mechanism maintains little state information.

5. Debugging concurrent processes.

This section examines the debugging of multiple concurrent processes. The most important issue here is the maintenance of time consistency between those processes when a breakpoint occurs. In addition, this section describes how the concurrent environment complicates the implementation of many conventional debugger features. The presence of other concurrent users in a distributed system is addressed in the next section.

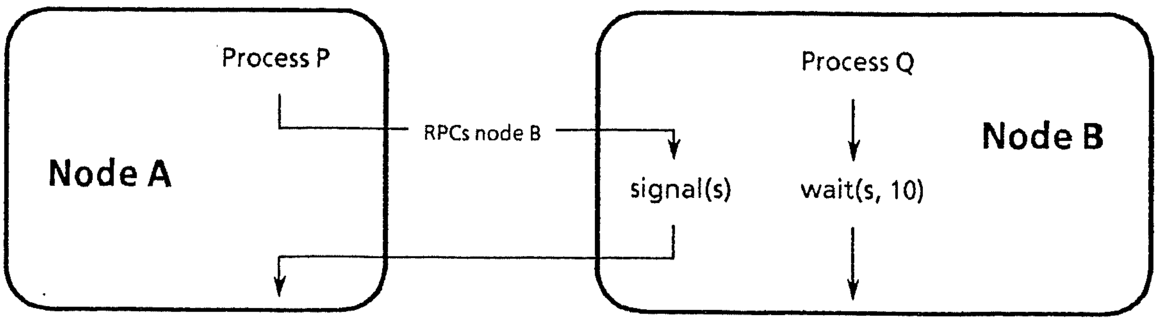

Figure 2. Breakpointing with remote processes.

5.1 Breakpointing a concurrent program.

A fundamental operation in most sequential debuggers is the breakpointing of the program to allow its state to be examined. Here I discuss what the appropriate counterpart of this operation is for concurrent and distributed programs. In the following I refer to breakpointing, but the same arguments apply any time the programmer wishes to interrupt the execution of the program to examine its state such as after an execution error.

Ideally the programmer wishes to instantaneously suspend all of the processes in the program. A number of problems may arise if some processes continue executing when a breakpoint occurs. Firstly the still running processes may alter the state which is being observed. Secondly, the running processes, and the breakpointed ones after they are resumed, may carry out a different computation to the one that would have occurred had the breakpoint not happened. Thirdly, if the programmer wishes to alter the state of the program or cause a different computation to occur, he or she may be unable to do so if some processes are still running.

Consider the example in Figure 2. Process *Q* on node *B* is waiting on the semaphore *s* with a timeout of 10 seconds. Also on node *B* is the body of a remote procedure which signals *s*. Process *P* on node *A* calls this remote procedure. If a breakpoint occurs and the processes on node *A* are halted before those on node *B* then there is a chance that *Q* will "see" that *P* has halted. That is, its semaphore wait may timeout whereas if the breakpoint hadn't occurred it may have been signalled by *P* first.

Strictly speaking a process need not be halted at the instant a breakpoint occurs. Rather it can continue executing until it either notices that some processes are halted, or alters some state accessible to them.

The arguments in Lamport's paper on distributed time consistency [Lamport 78] are relevant here. The communications between processes and the events internal to each process together impose a partial order on the events in the entire system and the implementor of a distributed system is free to choose any total order which is consistent with that partial order. When a breakpoint occurs, any process may continue running only so long as it does not upset the partial ordering of events which would have occurred had the breakpoint not happened. This will be referred to as *transparently halting* a process.

In practice, transparent halting of multiple distributed processes is difficult to achieve. Where it is impossible a weaker requirement should still be met, namely that a "typical computation" occurs following a breakpoint. A typical computation is one that could reasonably have occurred in the absence of the debugger. The execution of a distributed program is subject to the nondeterminism of processor scheduling, network delay and packet loss. As long as the effects of the debugger on the ordering and timing of program events are similar to those caused by these other sources of nondeterminism we can say a typical computation has occurred. If a debugger causes atypical computations bugs which occur in the normal execution of a program might never be able to be reproduced while under control of the debugger.

It is useful to review the ways in which processes that are still running can observe or affect halted processes. Processes on the same node of a computer network can communicate through common memory. This may take place via data objects designed for inter-process sharing such as semaphores, monitor locks, or critical regions. These objects may in turn protect access to sequential data objects like arrays and records. Interaction may occur through undisciplined or unsafe concurrent access to data. It is important to consider this possibility since the programs which the debugger must cope with probably contain bugs of this kind. Processes on different nodes may communicate by sending and receiving messages. Processes may interact by not sending a message or not updating memory before some time limit has expired. (The action of reading a clock, or timing out from a semaphore or message wait can be modelled as a message exchange with some clock process.) Another kind of interaction which is often overlooked is communication via third parties such as network servers or long term shared memory such as files.

5.2 Distributed breakpointing in Pilgrim.

Pilgrim maintains a logical clock at each node of the program. When a breakpoint occurs the processes and clocks on each node are halted. The processes executing on the node which encountered the breakpoint are halted immediately. Those executing on other nodes are halted as soon as possible subject to communication delays. A significant feature of the technique is that programs under control of the debugger do not execute more slowly because of this breakpointing mechanism.

The clock delta.

The logical clock is implemented by computing the difference, or *delta*, from the real time clock value maintained by the Mayflower supervisor. The real time clocks at each node are assumed to be synchronized correctly. The delta is subtracted from all date and time values read by the user program. When the program is executing normally, a copy of the delta is held in each node. While the program is halted at a breakpoint the delta is found by:

current time – time of breakpoint + previous time delta .

At the end of a debugging session the logical clock is reset to real time. The effects of this may be unpredictable thus adding to the arguments against letting the program continue executing after the debugger has been disconnected.

Halting processes on a single processor.

It is relatively easy to transparently halt processes which are time sliced on a single processor. A primitive was added to the Mayflower supervisor allowing one process to place other selected processes on a special wait queue. Any processes in this queue which were waiting on semaphores have their timeouts temporarily frozen. The agent code uses this primitive not only when a breakpoint is hit but upon hardware exceptions and user program failures as well. It could also be used during any long debugging actions which might upset the user program's timing. (No such actions exist yet.)

There are processes which must not be halted upon debugging. These include some processes in the runtime support library and the agent code itself. A bit was added for this purpose to the supervisor's process data structure specifying whether or not the process it describes should be halted.

Halting distributed processes.

To halt remote processes, messages are sent by the agent on the node which encountered the breakpoint to the agents on other nodes that are under control of the debugger. On receiving this message each agent will halt the processes on its node.

To achieve transparent halting it would be necessary to guarantee to send messages to every affected node in less time than it would take a single RPC to reach any of them. This is to ensure each node is halted before any of its processes can timeout because of non-receipt of a message from another halted node. Something approaching this can be achieved on a single broadcast network such as Ethernet. On a Cambridge Ring a number of messages must be sent serially because although the ring uses a broadcast medium it does not provide a broadcast facility at the data-link layer. Even on an Ethernet the requirement for a reliable broadcast is not met by a single broadcast or multicast packet.

A number of protocols are available which do achieve the necessary reliability [Mockapetris 83]. The particular protocol used in Pilgrim is closest to a negative acknowledgement scheme. A sequence of messages is sent, one to each node under control of the debugger. On a Cambridge Ring the transmitting hardware is informed if the packet just sent was not received by the destination network interface. This information can be obtained by the agent and will cause a retransmission. This guarantees all the destinations receive the message into their network buffers. It is assumed that either the agent software in those nodes is functioning correctly and will process the message, or the entire node has crashed.

Effects on program execution of the halting of distributed processes.

Under this broadcasting scheme a breakpoint and the subsequent halting of processes may affect the following computation if some nodes are unable to be contacted soon enough. Under the Mayflower system nodes communicate by RPCs, the minimum latency time for which is about 8 ms. Unfortunately, because of the efficiency and light-weight semantics of the Mayflower RPC system, this is close to the 3.5 ms required for a small Basic Block message (which is the lowest level protocol generally available). Thus we could be confident of contacting only two nodes in the time available for halting remote processes.

In such cases the strict requirements of transparent halting may not always be fulfilled, although a typical computation will still occur. "Well-behaved" programs, which use timeouts appropriate to the reliability and delay of the network, will be handled correctly by the debugger. But highly timing-sensitive faults in programs comprising more than three nodes may be difficult to debug.

5.3 Other approaches to breakpointing.

A number of more elaborate schemes for breakpointing were seriously considered but rejected as unsuitable for target environment debugging.

One scheme would be to ensure no other nodes had halted before allowing a process to receive a message, resume from a semaphore wait, or claim a monitor lock. Thus there would be no possibility of processes running on after a breakpoint and altering the future of the computation. Unfortunately determining if other nodes had halted requires a network interaction so the program would now execute at considerably reduced speed. Even the claiming of a monitor lock, which occurs very frequently and experiences little contention, would probably result in network traffic. Such poor performance is not suitable for a target environment debugger. Further, unsafe process communication via shared variables could not be handled by this scheme.

Reversible or replayable execution.

An approach taken in some debugging proposals [Jefferson 85, Di Maio 85] is to avoid interrupting the execution of the program at all when debugging. Instead a modified version of the program is generated which saves transcripts of all inter-node communication and records process states (by checkpointing or by maintaining history lists of variable contents). This highly instrumented program is then run to completion or failure. The impression of reversible execution is achieved by restarting a node from a particular checkpoint and then replaying the external communication it received. Unless subtle effects such as the exact timing of process time slicing can be deterministically reproduced this impression will fail to be convincing. The checkpointing and transcripting operations will take time and distort the original computation if care is not taken. A lot of space is needed for the retained information and the overheads will be severe unless most of this information is already being maintained for other reasons, as is the case with Chiu's proposal for debugging atomic actions [Chiu 84].

5.4 Accessing and modifying process information in Pilgrim.

Besides maintaining time consistency, the debugger should allow access to and where possible modification of all the concurrency constructs in the language. Thus the execution states of processes (running, waiting on a semaphore, waiting on a monitor lock, etc.) should be visible, and it should be possible to transfer a process between these states from the debugger. The procedure call stacks of all processes should be visible, and their variables and the objects they refer to modifiable.

The implementation of process management in Mayflower is split between the supervisor and the Concurrent CLU runtime library. For debugging purposes a new supervisor primitive was added to obtain those components of a process's state which are known only by the supervisor. These are: whether the process is runnable or waiting; if runnable, the register set; if waiting, the semaphore or monitor queue it is waiting on; and the process priority.

Changes to the runtime library modules were mainly restricted to hooks in the process creation and deletion code to call procedures in the agent. The agent must know of the existence of every process and maintain information which assists in mapping the supervisor's view of process state to the Concurrent CLU language view. A process must be able to determine its process identifier for many debugging activities. Although a function existed to do this, it was extremely slow and had to be re-implemented.

5.5 Implementation difficulties in a concurrent environment.

In a concurrent execution environment some of the implementation details of conventional debugger features become difficult. Mayflower uses a time slicing scheduler and the possibility of a process switch at any moment underlies many of the difficulties described below.

Interpreting the top of stack.

Pilgrim allows procedure call stacks to be examined at any time, not just when the process that owns the stack has hit a breakpoint. This complicates the task of interpreting the words at the very top of the stack. The stack may be left, temporarily, in an unusual state during some assembly language procedures in the runtime library and during procedure entry and exit sequences. This is compounded in Concurrent CLU's case by the multiplicity of code sequences used which depend on the number of arguments and results of the procedure. Very similar problems are reported by the designers of the Blit debugger [Cargill 85].

The two pieces of information required from the top of stack are a pointer to an object code location and a pointer to a stack frame for the highest well formed frame on the stack. The compiler and assembler were modified to generate tables describing, for a given program counter value, where these values could be found (e.g. in a register, or at a fixed offset from the top of stack pointer). Assembly language code had to abide by some extra conventions for this scheme to function, and the few assembly routines which didn't were modified.

Critical library routines.

Some pieces of important library code must not be interrupted at all. Most important is the heap allocator which is implemented as a critical region. Since the agent allocates objects off the heap, any process which is interrupted within the allocator must be allowed to exit before halting.

Setting and clearing breakpoints.

Pilgrim sets a breakpoint by the familiar technique of replacing the object code at the desired location by a trap instruction which causes an internal interrupt. A process is stepped over a breakpoint by replacing the original code and executing that process in trace mode for one instruction, before replacing the trap. Trace mode is a 68000 debugging feature which generates an internal interrupt after every instruction. Care must be taken since the object code is shared by multiple processes. While the breakpoint is thus temporarily removed, other processes must be halted to prevent them from executing through the breakpointed location without trapping.

6. Debugging in the presence of shared services.

Until now it has been assumed that the user program existed in isolation on a distributed system. But a characteristic of distributed programs is that they use public servers shared with other users and programs on the system. When a program is breakpointed the programmer might wish processes on any servers it is using to halt, in addition to processes in the program itself. Processes which remain running on the servers can affect the state of the halted program. A server may notice lack of response in a client and cancel the client's session or abort the transaction currently in progress. A server may grant a resource to a client to be reclaimed after a predetermined time period, or a client and server may exchange date/time values as data. In all cases the client's and server's different views of time while the client is being debugged will cause problems.

The processes on the file servers, name servers, print servers and so on cannot be halted since other users would be denied service for unreasonable periods. Equally it is not always convenient to use private copies of these services because of the expense and the difficulty of entirely reconstructing the state in each duplicated service. The dynamics of load provided by other concurrent users of a distributed system is especially hard to reproduce.

It is not possible, in general, for the debugger to insulate the server from the effects of client processes halting, or to insulate the halted processes from the actions the server might take. The debugger could send periodic messages to a server to maintain a deadman's handle or an idle handshake for well-known protocols but quite complex application level protocols can be built on top of RPC and the debugger could not be expected to understand all of these. If the client is breakpointed while holding locks on important shared resources, it may be wrong to deny service to other clients by refreshing those locks.

Therefore it seems important for servers to handle the possibility of their clients being breakpointed while they themselves continue running and take appropriate action. If debugging is considered in the design of shared services from the outset a number of advantages may result. If a client is breakpointed, the server may schedule work for that client in the background, or release some of the resources held by that client (as long as they can be regained later).

It is also useful for some local processes, on the client nodes, to remain running. For instance it may be easier for the process in the client which handles all communication with the file server to remain running than to complicate the code of the file servers themselves so that they can handle debugging. The term "server" is used in the following but the facilities provided and the examples given also apply to local processes which continue running.

6.1 Support for debugging clients of shared servers.

The debugging system must give support to servers to enable them to maintain some degree of time consistency for a client that is being debugged. Pilgrim provides two procedures for this purpose, one implemented by the agents on all nodes of a program being debugged, the other by the debugger itself.

The agent provides the procedure:

```
    get_debuggee_status = proc () returns (network_address, date)
```

The first result is the network address of the debugger to which this node is connected. A special value signifies that the node is not currently under control of a debugger. The second result is the value of the node's logical clock. The logical times at each node of a program being debugged should be almost the same and the time reported for a node which is not being debugged should equal real time – within the tolerance of the distributed clock synchronization algorithm used.

The debugger maintains a log of the breakpoints which have occurred and for each how long the program's execution was interrupted. The sum of these values will be almost the same as the logical time deltas at all nodes of the program. This breakpoint log is used to implement the procedure:

```
    convert_debuggee_time = proc (date) returns (date)
```

which takes a date/time value for some point in the past and returns the equivalent client logical date/time. Date/time values maintained by the server referring to past events can be converted into the client's time scale by calling the convert_debuggee_time procedure. The network address provided by the agent gives the correct destination for the call.

6.2 Examples of use.

To understand how the above support is useful some examples follow. The servers used in the examples are taken from the Cambridge Distributed Computing System.

Ignoring long timeouts.

The simplest way a server can use this debugging information is to determine if the client is under control of a debugger. If so the server can extend indefinitely any timeouts relating to the client. For instance the Resource Manager allocates machines to users and programs. These resources are reclaimed by the manager after long timeouts (typically three hours) have expired. Extending the timeouts on a client's resources, at least until the end of the debugging session, will satisfy almost all situations.

Even if there is some client code which handles the expiration of these timeouts, this is unlikely to be debugged by waiting three hours for a timeout to actually expire. Rather the programmer will build a small test harness which allows particular timeout situations to be created.

Precisely extending time intervals.

With more sophisticated use of the debugging information a server can extend timeouts by precisely the amount necessary to match the client's logical time scale.

For instance the authentication manager, AOTMan, issues *temporary unique identifiers* or TUIDs which are capability-like objects describing rights of access or service. TUIDs must be continually refreshed before their timeouts, typically two to five minutes long, expire. Finding a bug in a client, such as accidentally omitting to refresh a TUID, would be much easier if AOTMan extended timeouts by the correct amount when the client was under control of the debugger.

With the mechanisms above there are two ways for a server to correctly extend a timeout. An algorithm for the first method is shown in Figure 3. The server obtains the client's logical time just before the timeout begins. If the timeout expires the time is obtained again from the client. If the client has not been breakpointed in the interim the difference between these two times should equal the length of the timeout in real time. If a breakpoint has occurred the period remaining in the timeout can be calculated and used as the new timeout value as another iteration of the loop is performed.

This scheme has the disadvantage that an invocation of get_debuggee_status on the client is required at the start of every timeout, even when that client is not being debugged, and even when the timeout will not in fact expire. The second method, shown in Figure 4, avoids this work unless the timeout does expire. However it then involves a call to both get_debuggee_status and convert_debuggee_time.

```
timeout: = original timeout
client: = network address of client
client_start, debugger_address: = call get_debuggee_status () at client
                                              % This is a remote procedure call.

keep_waiting: = true
while keep_waiting do
    keep_waiting: = false

    semaphore_wait (sem, timeout)
    except when timed_out:
        client_now, debugger_address: = call get_debuggee_status () at client

        if now() > client_now + clock_tolerance then
            % Client logical time is slow: client may have been breakpointed
            % during timeout.

            % Compute how much of the timeout remains.
            time_left: = timeout - (client_now - client_start)
            if time_left > clock_tolerance then
                timeout: = time_left
                client_start: = client_now
                keep_waiting: = true
            end
        end
    end
end
```

Figure 3. Extending timeouts using only get_debuggee_status.

```
timeout: = original timeout
client: = network address of client

keep_waiting: = true
while keep_waiting do
    keep_waiting: = false

    semaphore_wait (sem, timeout)
    except when timed_out:
        client_now, debugger: = call get_debuggee_status () at client
        real_now: = now()

        if real_now > client_now + clock_tolerance then
            % Client logical time is slow: client may have been breakpointed
            % during timeout.

            % Compute how much of the timeout remains.
            client_start: = call convert_debuggee_time (real_now - timeout) at debugger

            time_left: = timeout - (client_now - client_start)
            if time_left > clock_tolerance then
                timeout: = time_left
                keep_waiting: = true
            end
        end
    end
end
```

Figure 4. Extending timeouts using get_debuggee_status and convert_debuggee_time.

Resource contention with other users.

The methods just described are applicable to a server which manages some set of shared resources for which many clients contend. When a resource is allocated to a client a timeout on the allocation is usually imposed. However extending that timeout when debugging may be wrong if the resource is very scarce and other clients require it. A decision must be made between delaying these other clients or reclaiming the resource (and thus upsetting a client's debugging session).

The server could provide some administrative commands by which the scarcity of a resource and the importance of some debugging session could be indicated. This might be unwieldy to implement or use.

A simpler approach has the server extending a timeout on some resource allocation until a client, not under control of the same debugger, requests the resource. At that point the resource is reclaimed and reallocated.

Converting date/time data.

A client that is being debugged may notice inconsistent timing if it receives explicit date/time values from a server, for instance as the date of last modification of a file. A server can convert this time data using the convert_debuggee_time procedure. However this may not be enough to preserve complete time consistency without also reproducing the contents of the file at an appropriate time in the past. It is important to recognize limits to how much network servers can do to maintain time consistency, since their main purpose is still to provide reliable and efficient service to all of their clients.

There are many aspects of time consistency which servers cannot achieve with the support described. The time of future events cannot be converted correctly and clients cannot time real world events over a breakpoint. However time consistency is maintained about events within the program being debugged even when time data about such events is communicated through other parties such as servers

The two debugging support procedures constitute a security problem which has not been addressed in the design of Pilgrim. A malicious program could provide implementations of these procedures allowing it more favoured treatment from servers. Thus some kind of authentication is needed so servers can detect misuse.

6.3 A strategy for debugging clients of shared services.

The facilities described in the previous section do not insulate the program being debugged from all outside influence and do not prevent that program from potentially affecting other users of a distributed system. A strategy for debugging clients of shared services is required which considers the needs of both the programmer using the debugger and other users in the system.

In the early stages of debugging a client, the servers it uses can usually be shared with other users but the data objects operated on should be private copies (e.g. private files rather than public shared ones). It is wise to do this, not only to avoid delaying other clients while debugging, but also because the possibly faulty client may carry out erroneous operations on the object or even destroy it.

In the final stages of debugging and tuning and after the program goes into service, the client will be operating on "live data". It will still be possible to examine the program under the debugger, but primary consideration should be given to other users of the distributed system. Use of the debugger in this situation should probably be restricted to examining the program state. Extensive debugging would require a return to one of the private or semi-private environments used earlier.

7. Conclusion.

This work has emphasized target environment, source language debugging. Debugging in the target environment is important because the most troublesome bugs often appear after the program has been transferred to an environment in which there is no support for debugging. Debugging at the source language level is important so that the concepts and constructs which are helpful through the design and development of a program need not be abandoned for the debugging stage.

Although Pilgrim was designed for Concurrent CLU, the techniques described here are relevant to debuggers for other similar languages such as Ada or Modula-2+. In addition the distributed breakpointing mechanism and the techniques for debugging clients of shared servers are applicable to languages which use message passing rather than RPC for remote communication.

Acknowledgements.

I wish to thank Roger Needham and Jean Bacon for their comments and suggestions on this paper.

References.

A. D. Birrell and B. J. Nelson "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, **2** (1), February 1984, p. 39.

T. A. Cargill "Implementation of the BLIT Debugger." *Software – Practice and Experience*, **15** (2), February 1985, p. 153.

S. Y. Chiu *Debugging Distributed Computations in a Nested Atomic Transaction System*. PhD. Dissertation, Technical Report MIT/LCS/TR-327, Laboratory for Computer Science, MIT, December 1985.

R. C. B. Cooper and K. G. Hamilton "Preserving Abstraction in Concurrent Programming." *IEEE Transactions on Software Engineering (to appear).* (Also available as Technical Report 76, Computer Laboratory, University of Cambridge, August 1985).

A. Di Maio, S. Ceri, S. Crespi Reghizzi "Execution Monitoring and Debugging Tool for Ada Using Relational Algebra." *Ada in Use, Proc. Ada International Conference*, Paris, (*Ada Letters*, **5** (2)), September 1985, p. 109.

K. G. Hamilton *A Remote Procedure Call System*. PhD. Dissertation, Technical Report 70, Computer Laboratory, University of Cambridge, December 1984.

D. R. Jefferson "Virtual Time." *ACM Transactions on Programming Languages and Systems*, 7 (3), July 1985, p. 404.

L. Lamport "Time, Clocks and the Ordering of Events in a Distributed System." *Communications of the ACM*, 21 (7), July 1978, p. 548.

B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder *CLU Reference Manual*, Vol. 114, *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

P. V. Mockapetris "Analysis of Reliable Multicast Algorithms for Local Networks." *Proc. Eighth Data Communications Symposium*, ACM, October 1983, p. 150.

R. M. Needham and A. J. Herbert *The Cambridge Distributed Computing System*. Addison-Wesley, London, 1982.