BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M2

The CARE Package for CApture and REplay of Parallel Code Sequences

by
Erik J. Morse

The CARE Package for CApture and REplay of Parallel Code Sequences

Erik J. Morse

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

May 1991

[illegible]

Professor Steven P. Reiss
Advisor

The CARE Package for CApture and REplay of Parallel Code Sequences

Erik J. Morse
Box 1910, Computer Science Department
Brown University
Providence, RI 02912

January 22, 1991

Abstract

Debugging and testing of parallel programs is difficult due to the nondeterminism introduced by parallel constructs. In this paper we discuss previously proposed solutions, including: static source code analysis, problem based systems that describe and recognize problematic event occurences, reverse execution of programs, and capture/replay tools that observe and reproduce executions of parallel programs. The latter approach provides the parallel program developer with traditional cyclic debugging capability. We describe a general capture and replay approach that we have implemented. Our system provides the user with routines for interprocess communication (ipc) via message passing and safe access to shared memory. The capture and replay mechanisms are built into these routines. The system provides a large amount of flexibility not found in other reproducible execution solutions.

Contents

Chapter 1

Introduction

Cyclic debugging is the most common method for software programmers to find errors in their code. Cyclic debugging is the process of running a program, locating an error, correcting the error and then repeating the procedure until no errors appear to exist. In traditional sequential programs the process of the debugging cycle is fairly well understood. Each re-execution of the same program with the same inputs results in the same code sequence being executed in the program under scrutiny. Cyclic debugging depends on this reproducibility to isolate the source of the error. Unfortunately, the method encounters problems when applied to parallel programs. We use the term parallel program to describe programs that support concurrent execution of processes in a distributed and/or multiprocessor environment, where a process is an independent thread of control that is sequential in nature.

1.1 The Debugging Problem

Debugging and testing of parallel programs is difficult due to the fact that the programs are most often nondeterministic in nature and many bugs are timing related. Because of this it is often hard to reproduce errors on subsequent executions of the program, as subsequent executions may exercise different code sequences. A programmer writing a parallel program generally has no means of isolating the events that led to an error and no way to reproduce the events that caused the error. Often the introduction of a debugger into the program will change the timing of the events in the system and the error will disappear or mutate only to reappear once the debugger is removed.

1.2 The Testing Problem

When writing a test for a program a user generally will invoke the program with some defined set of inputs and then evaluate the program results to make sure they match the expected results. Alternatively, a user may execute the program under test manually and inspect the results. In both cases a program is determined to be correct if the expected program results occur. Unfortunately, when a program exhibits nondeterministic behavior, there is no guarantee that if it performed correctly on a set of inputs during one invocation that it will perform correctly on the same set of inputs on subsequent invocations. If different invocations may execute different code sequences then the tester must verify that each code sequence is valid and a test designed to exercise a specific code sequence must execute that code.

1.3 A Solution

The two problems described above are similar in nature. In order to solve both of the problems the user must be able to execute a nondeterministic program in a deterministic fashion. To do this it is necessary to determine a set of synchronization events within a program that fully describe a unique execution of the program. Then it is necessary to capture the events as they occur during an initial run of the program and then replay them in subsequent runs of the program. In addition to this capture and replay type of mechanism a variety of other approaches have attempted to deal with the problem of testing and debugging parallel programs. Static evaluation of parallel programs have attempted to isolate potential problems by locating concurrent accesses to shared memory and locations of possible nondeterminism by observing the programs source code. Problem based approaches have attempted to describe possible improper event sequences and actions to be taken if such a sequence is recognized. Additionally, derivatives of the capture and replay method known as reverse execution have been implemented. Reverse execution allows the user to back up to a known good state in the program and continue execution from there.

1.4 Types of Parallelism

A variety of different mechanisms for introducing parallelism into programs exist. Some languages such as Ada [1] provide explicit parallel constructs. Other systems offer programmers mechanisms to introduce parallelism into their programs (for example a remote procedure call system). There are two common communication mechanisms in both language based and general parallel systems. They are message passing and shared memory.

1.4.1 Message Passing Systems

In a distributed environment where no two processes share the same memory space a message passing mechanism can be used to allow processes to communicate. A message passing systems consists of a set of primitives such as sends, calls (sends that expect replies), blocking receives that wait for a message to arrive, and non-blocking receives that return if no message is currently pending. Processes communicate by passing messages between them. Synchronization events within these systems depend on the order that messages are received at each process.

1.4.2 Shared Memory

Systems that allow multiple processes to access shared data provide another type of parallelism. In these systems synchronization events are defined by reads and writes of the shared memory. In order for data integrity to be maintained systems must provide some type of restricted access to the shared data. These access mechanisms may allow for exclusive reads and writes, concurrent reads and exclusive writes or any other access patterns that preserve data integrity.

1.5 Paper Overview

The goal of this paper is to examine some of the approaches that have been used to aid in the debugging and testing of concurrent programs. We will concentrate primarily on the debugging issue, but the majority of the discussion can be applied to the testing problem as well. We examine each method's positive points and inadequacies. Then we describe a system that we have implemented to aid in this process. Our system is based on the capture and replay approach to debugging.

Chapter 2

Previous Systems and Approaches

2.1 Static Analysis

Static analysis of a program is an attempt to understand the potential execution of a program by examining the program source code, not the actual program execution. Static analysis of parallel programs is primarily concerned with potential memory access conflicts and with control flow operations, not the value of predicates that may determine the flow.

2.1.1 START (STatic Anomaly Reporting Tool)

Appelbe and McDowell in [2] define a system with the goal of preventing program error resulting from improper concurrent access to shared data. To accomplish this the system converts a parallel program into a compressed dataflow graph that contains only information about synchronization events. Then corresponding program statements are merged back into their respective places in the graph and potential data access conflicts (e.g. deadlock) are checked. The system is limited in focus (it is designed for the scientific community and handles only augmented Fortran77) and exponential graph blowup is a possibility though aids to prevent this occurrence have been added.

2.1.2 Faust

Emrath and Padua have designed a system [10] as part of the Faust programming environment at the University of Illinois. This system attempts to locate possible points of nondeterminism within a parallel program. This extends the START approach in that it looks at control flow as well as data access. The system is designed for use in multiprocessor non-distributed environment. The hope is that the system will help the programmer recognize points of nondeterminism that were unintended, such as a missed synchronization statement. This is accomplished by viewing the program as a graph consisting of data dependency edges and control flow synchronization edges. If the former exists without a matching edge of the latter type then nondeterminism exists.

2.1.3 Static Analysis Summary

Static analysis may be expensive to implement and time consuming to run. The examples we have seen have been limited in their focus. It often considers control flow direction that may be impossible (due to lack of predicate evaluation) and thus may detect errors that can not occur [18]. If the user can easily recognize these false alarms static analysis may be helpful, but a great deal of work still needs to be done in this area to provide the user with a truly useful tool.

2.2 Problem Based Solutions

A problem based approach to debugging involves creating a method for the programmer to specify a set of primitive events that should not occur in a given program. These events may be as simple as the zeroing of a variable or a complicated sequence of event interactions. An underlying system then monitors all events that the program produces and checks for the occurrence of a problematic sequence. If a specified sequence is detected the system acts in some way to notify the user.

2.2.1 EBBA (Event Based Behavioral Abstraction) and Data Path Expressions

Both systems [3] [12] employ an event definition language to allow the user to specify events and event interactions, including parallelism. The systems maintain a set of primitive events and whenever a primitive event occurs the system sends it through a filtering process and then on to a recognizer. The recognizer is a finite state automata like machine created from the the user specified events sequences and interactions. When the system recognizes an event sequence appropriate actions are performed (e.g. debugger breaks).

2.2.2 Problem Based Summary

The main problem with these systems is that all events must be fed through a central recognizer. The event volume is generally enormous and consequently the systems are fairly slow. Additionally, compiler support is needed to obtain and send all occurrences of the primitive events to the recognizer. One goal for these systems is that they eventually be used in conjunction with a capture/replay method (see below) to recognize erroneous event behavior and then replay it for the user to observe.

2.3 Capture and Replay

The capture and replay method of parallel debugging involves capturing essential synchronization points during a programs execution and then using this information to allow for reproducible deterministic execution of the program. The synchronization points may include communication via message exchanges or accesses of shared memory locations. The programmer may use the capture log to try to identify a bad sequence of events (if the log contains readable information) or may replay the execution, perhaps under debugger control.

2.3.1 BugNet

BugNet [21] is the system that most closely resembles our work in the area of message passing. The goal of BugNet is to aid in the debugging of distributed C programs within a local area Unix[1] network. BugNet captures interprocess communication (ipc) information as well as input/output activity and execution traces. BugNet does not allow for shared memory communication. BugNet does however, make an attempt at capturing system time stamps and replaying programs in approximately the same time frame as the initial invocation. BugNet operates by spawning tracer and trap processes (for ipc capture and i/o capture respectively) on each node that the program may run on. A program must initially register itself in order to connect to BugNet. While running, BugNet periodically stops all processes in order to checkpoint their state. This allows the user to continue execution of a process from a recently checkpointed state.

BugNet completely captures all information passed via the ipc mechanism. This is necessary because BugNet replays messages from the capture file data instead of waiting for the actual message to arrive. This approach has both positive and negative points. The user has the ability to replay a single process without worrying about the re-execution of the rest of the system. However, the user can not replay the entire system and observe various processes at different points during its execution. Another problem will arise if if there are a large number of messages being passed in a system or if the messages are very large in size. Since BugNet captures complete message information this could result in a fast growth in the capture file size. BugNet uses unreliable datagrams as a base for communications and as a consequence the system may abort when a large number of messages are send because messages get lost. Due to the BugNet interface and separate trace and trap processes a program may have trouble connecting when the user uses a debugger. BugNet's logging slows ipc communication by a factor of three. BugNet does not provide a mechanism for nonblocking receives and this limits its use as a general purpose ipc system.

[1]Unix is a trademark of AT&T Bell Laboratories.

2.3.2 Deterministic Execution Debugging

Carver and Tai [5] present a system for capture and replay within the confines of the concurrent Ada language. The approach is called language based due to the fact that the synchronization events are defined in terms of the language constructs. The definition of these events in concurrent Ada is more difficult than in simple message passing or shared memory systems. The reason for this is the availability of the rendezvous construct that allows processes (called tasks in Ada) to interact in manners different than blocking and nonblocking sends and receives. It is possible for a calling task to rendezvous with a called task at one of a variety of different locations in the called task.

The system works by translating the source code of a given Ada program into a similar program that has the ability to capture the defined synchronization events, but in no other way affects the results of the program. The new program may then be run to capture the synchronization events. Then a second translation is performed on the original program. This translation creates a program that, when given the captured data, will replay the program and guarantee the results will be the same as the program when the capture version was run.

The system has some serious limitations. Access to shared variables is not allowed, nor are statements that reply on the system clock. Furthermore, use of several standard Ada constructs is not allowed.

2.3.3 Instant Replay

LeBlanc's Instant Replay [16] models all communication as operations on shared data objects. Messages are modeled as operations on a shared port or mailbox. Version numbers for each shared memory object are maintained. Each write creates a new version of the object and each read reads a particular version of the object. The system provides routines that allow for concurrent reads and exclusive writes. Each access must be explicitly specified by the program as ReaderEnter, ReaderExit, WriterEnter or WriterExit. If accesses are made without using these routines Instant Replay can not guarantee accurate capture and replay.

The authors claim that simply forcing each process to replay correctly will guarantee that the results of the entire program will be the same. This is not necessarily the case if different processes output to the same location (file, terminal etc...).

During replay the order of output events may change. Monitoring needs to be in place on output as well as input. Instant Replay depends on the virtual machine environments to be the same during each execution. This is a similar problem to the one described in the previous section where accesses to machine dependent features such as the real time clock can not be replayed.

2.3.4 Capture and Replay Summary

As we can see by the systems presented here each has points that make it a good choice for some programs and a bad choice for others. Capturing complete data allows the user to replay a single process as desired, but may cause a blowup in capture data size. Capturing minimal data keeps the capture log small, but forces replay of the entire system even when it is not needed. Capturing shared memory accesses aids in debugging local parallel systems, but not in debugging distributed systems. Capturing only message data has the opposite result. Some of the approaches above altered the user's original program. This forces the user to have an understanding of the underlying capture and replay mechanisms in order to debug the altered program.

2.4 Reverse Execution

Reverse execution attempts to address one of the major drawbacks we saw in the capture/replay method, namely the potential preponderance of captured data. One type of reverse execution system may periodically checkpoint the entire program state and then only capture data until the next checkpoint. When a checkpoint is reached all previous captured data is discarded. Upon realization of an error the system may be rolled back to the last checkpoint and then proceed until the error location.

2.4.1 IGOR

IGOR [4] is a prototype system that does not attempt to capture synchronization event behavior, instead it periodically saves all dirty memory pages. When the program halts IGOR can recreate a past state of the system for the user to examine. Then the program can move forward from that state via an interpreter.

IGOR currently only handles single processes, but it is not difficult to see how this concept could be extended for use with parallel programs. Because IGOR does not capture synchronization behavior, backing up to a previous known good state and then moving forward would not guarantee that in a parallel environment the same code sequence would occur. Furthermore, since IGOR's checkpointing scheme is potentially problematic since it revolves around dirty pages. If a program's working page set constantly changes checkpointing has to occur often. This would dramatically slow down, and most likely change the execution of the program versus a run without checkpointing. IGOR required changes to the underlying operating system in order to checkpoint with this dirty page method.

2.4.2 Recap

Recap [17] is another system that attempts to solve the parallel debugging problem via checkpointing and reverse execution. Recap captures all system calls and shared memory accesses. It checkpoints by saving the address space and processor state for a process or group of processes. This is accomplished by forking a child process and suspending it. The parent process continues in a capture mode while the child is set to a replay mode. Replay is accomplished by continuing execution of the child process from the checkpoint.

Recap depends on compiler modifications to capture shared memory accesses and special run time library calls to capture system calls. Though this may not be pleasant to implement, it does guarantee capture of all shared memory accesses, not just those that are contained within synchronization primitives. Recap only supports replay of events from the capture log. As we saw in the other capture/replay systems this means that during replay the programmer can only look at one process at a time as the other processes are not running.

2.4.3 Reverse Execution Summary

Reverse execution solves many problems. By checkpointing periodically useless capture data can be discarded thereby keeping the capture log small. An additional advantage that checkpointing provides is the ability to replay a small amount of the program to get to the point of error. If a program has been running for an extended period of time (minutes, hours, etc..) the programmer may not want to wait until the complete program replays. Optimally, we would like to see a true reverse execution tool. One that allows the user to run a program backwards step by step in order to determine the course of events that led to program error.

Chapter 3

The CARE Package

3.1 Our Goals

We were concerned with providing a mechanism to aid in the testing and debugging of parallel programs. We opted to do this within the framework of the tools available to us here at Brown University, namely a local area network of workstations running Unix and serviced by NFS. All our work is in the C programming language [14] and we make use of the Brown Threads [9] package to provide us with local parallelism. During the course of our study in this area the tools we as developers found to be most useful were those that performed some kind of capture and replay of the parallel code sequences. We wanted to design a tool that provided this functionality. At the same time we wanted to allow the user of our tool to be free of any specific knowledge of the capture and replay details of the underlying system. As was mentioned before, other systems require the user to be aware of changes to their code in order to properly use the capture and replay mechanism. At the same time we wanted to provide the user with a tool that impeded their programs performance as little as possible during capture and not at all when the capture and replay option was turned off. Our final goal was to make our system flexible, allowing the user to choose a mode of operation that best suited the program under development. With the above in mind we decided to implement our own interprocess communication (ipc) system and our own shared memory access system with all capture and replay functionality resident in the underlying system.

3.2 The Design

Our system consists of two pieces, one to handle concurrent shared memory access and one to handle distributed communication. We adapted LeBlanc's Instant Replay model [16] to allow capture and replay of shared memory accesses. This system allows for concurrent reads and exclusive writes of shared memory objects.

We designed our message passing system based partially on [13] and [7]. The latter system provided nonblocking sends, blocking receives and calls (a blocking send that expects a reply). We augmented this model to include nonblocking receives and message capture and replay capabilities.

We wanted our system to be as flexible as possible. During the capture stage we wanted to capture enough information to allow deterministic re-execution of the program. The user should have the option of replaying just a single process or the entire program. In order to allow this we designed our system with a variety of different modes for capture and replay.

3.3 Definition of Synchronization Events

Each process, given a specific set of inputs to that process, executes in a deterministic fashion since the only things that can affect the execution of the process are the inputs. The possible inputs in a multi-threaded and/or distributed C program therefore define the synchronization events. The possible inputs that may change from one execution to the next in our environment are: message receipt, access of a shared variable, file or user input and access to dynamic system variables such as the clock, directory listings, etc... For simplicity we will always use the system clock as the example for this class of inputs.

There is one other synchronization event that is important to note. That is file output. If it is important for the overall system to generate the same output on each run (if for instance two processes write to the same file or terminal window and the order of writes is important) then this output also defines a synchronization event.

3.4 Limitations

Accesses to the system clock that affect the execution of a process are not allowed as the current system does not replay system clock times. Note that this problem is not unique to parallel programs. Sequential programs that access the system clock will also have indeterminate results during a normal debugging or testing situation.

We currently limit input to user input (which we trust the user to replay) and non-shared file input (no two processes accessing the same file for input in a non-deterministic manner). We do not have a mechanism in place for capturing and replaying shared output.

Our shared memory accesses scheme requires the use of our locking primitives. No support is provided for capturing and replaying shared memory access that does not make use of our locking primitives.

3.5 Guaranteeing Deterministic Replay

As a consequence of our imposed limitations we need only concentrate on guaranteeing that each process receives its messages in the correct order and that each access to a shared variable yields the same result it did during the initial run of the program.

Our system may be run in six different modes. These modes are

- NORMAL - No capture or replay is performed.
- CAPTURE - Capture minimal information. Used in conjunction with the REPLAY mode to replay an entire program and all its interactions.
- FULL_CAPTURE - Capture complete information. Used in conjunction with REPLAY_ONE to replay a single process.
- REPLAY - Replay an entire program.
- REPLAY_ONE - Replay a single process.
- CHECKPOINT - See additional features section below.

3.5.1 Guaranteeing Message Receipt Replay

Each message is stamped with a unique identifier from the system that sends it (currently a simple incremental number is used). When the system is run in a capture mode messages are caught by the receive thread and placed in a first in first out (fifo) queue. Each receive request that the user issues takes the first message out of the queue and writes the message sender and message identifier to the capture file. If the mode is FULL_CAPTURE the complete message contents are also logged. If the receive request is a timed receive and no message is in the queue when the time out expires, a blank message is written to the file indicating that a nonblocking receive timed out.

When the user program issues a receive request during REPLAY_ONE (the capture mode must have been FULL_CAPTURE) , the system simply reads the first message out of the capture file and returns it to the user. If the user receive call timed out during capture the replay system notices this by reading the blank message and returns an empty message.

If the user is replaying the entire program (mode is REPLAY and capture mode was CAPTURE), the receive thread places messages as they arrive into a queue. Note however that this may not be the same order in which they arrived during capture. When the user system issues a receive request the ipc system reads the next message identifier and sender name from the capture file. If the message in the capture file is blank, indicating a nonblocking receive timed out during the capture phase, the receive routine simply returns a empty message. If the message in the capture file is not blank the system searches the queue for a message that matches the sender name and message identifier retrieved from the capture file. If the message is found it is pulled from the queue and returned to the user. If the message is not currently in the queue the ipc system waits until the message arrives and then returns it to the user. This guarantees the user system sees the messages in the same order during replay as it did during capture, even if they arrive to the process in a different order. The user is blind to this capture and replay activity.

3.5.2 Guaranteeing Replay of Shared Memory Accesses

As was mentioned earlier we have adapted the Instant Replay system [16] for capture and replay of shared memory accesses. The system provides four routines, ReadEntry, ReadExit, WriteEntry and WriteExit. Each time a shared memory object is accessed for reading the user must call ReadEntry prior to access and ReadExit after the access. The user may make multiple accesses to an object between a ReadEntry and ReadExit, but parallelism may be decreased if a large number of other instructions are included in between the calls. During capture the ReadEntry routine waits for any pending write on the object to complete and then increments the number of active readers of the object. Then, if the mode is FULL_CAPTURE, it logs the name and contents of the object to the capture file. If the mode is simply CAPTURE it logs the name and version of the object to the capture file. The ReadExit routine simply increments the total number of readers for the version of the object and decrements the active number of readers.

The WriteEntry routine waits for any active readers to finish. Then, if the mode is CAPTURE, it logs the object name, version and total readers for the version of the object to the capture file. No additional action is taken if the mode is FULL_CAPTURE. In both capture modes the WriteExit routine simply increments the version of the object and zeros the total readers for the new version of the object.

During replay if the mode is REPLAY_ONE the contents of the object are simply retrieved from the capture file and restored into the object during a ReadEntry call. If the mode is REPLAY then the ReadEntry routine reads the object version from the capture file and waits to return until the correct version of the object is available in memory.

The WriteEntry and WriteExit routines are no-ops for REPLAY_ONE, but in REPLAY the WriteEntry routine reads from the capture file the version of the object that it is supposed to write and the total number of readers of the previous version of the object. It then waits until all readers of previous versions of the object have completed their reads before writing the current version of the object into memory. In REPLAY mode the WriteExit routine behaves as it does in the capture case simply zeroing the objects total readers field and incrementing the objects version number.

This method allows for concurrent readers and exclusive writers and guarantees that during replay each object read receives the same value it did during the initial capture stage.

3.6 Overcoming Limitations

Our goal in writing this system was to force others to change their code as minimally as possible. We do require the user to use our ReadEntry, ReadExit, WriteEntry and WriteExit routines in order to guarantee that shared variable access can be captured and replayed. We don't consider this a major imposition since some form of restricted access routines would be required if the user wishes to safely access shared variables. A problem with this approach is that is does not catch cases where the user mistakenly forgets to use the access routines. This is often a point of error in the development of parallel programs. A possible solution to this problem is to have either the compiler or some type of cross-referencer check the code for unguarded uses of shared variables and generate the capture and replay code for the variables.

We do not provide a mechanism for handling system calls that return dynamic results. We could provide our own versions of these system routines. Our routines would log return values during capture and simply return these values during replay.

Another limitation that we did not address is shared file access. This is a problem when separate processes read from or write to a shared file, terminal window, etc... We could deal with this problem with a mechanism similar to our memory access solution. Each i/o destination could have a version associated with it. During capture a process would log the version it used and increment the global version number. During replay a process could not perform the i/o operation until the correct version number was reached.

3.7 Additional Features

One of the major problems in any trace system is that the capture log may grow to be very large. This may happen if the volume of synchronization events is very high or if the program runs for an extended period of time before an error occurs. In order to solve this problem we have adopted a checkpointing scheme similar to that found in the Recap system [17]. Our method of checkpointing is for each process to capture a user specified number of events per process before the process checkpoints. We only allow checkpointing of individual processes and consequently the mode of capture must be complete information. This means that during the capture part of checkpointing the mode is identical to FULL_CAPTURE. During the replay of a checkpointed process the mode is identical to REPLAY_ONE. When the process reaches the checkpoint the capture log is emptied and the process forks itself. This effectively creates an exact duplicate of the checkpointing process with state maintained. The child process sets its mode to REPLAY_ONE and immediately suspends itself. The parent process continues in the FULL_CAPTURE checkpoint mode. At future checkpoints the existing child is killed before the fork and the creation of the new child. When the parent process dies due to error the user can wake up the child process and replay the last sequence of events that led to error in the parent. When a child is wakened it immediately forks itself so that the user can repeatedly replay this part of the program if desired simply by awakening the new child.

Chapter 4

CARE User Manual

4.1 General User Routines

We have developed a set of routines that provide the user with a general purpose interprocess communication mechanism consisting of blocking and nonblocking sends and blocking and nonblocking receives. We also provide a set of locking primitives that allow for concurrent reads and exclusive writes of shared memory data. All arguments are input parameters unless designated as being of type output.

4.1.1 Init

```
Pinfo *Init(name)
char *name;
```

The Init routine initializes a process so that it may use the ipc and shared memory access mechanisms. The name argument is the name to be used to identify this process to other processes that wish to communicate with it (see FindPartner). Name must be less than MAX_NAME_SIZE characters or it will be truncated to MAX_NAME_SIZE characters. Names must be unique within a program. The Init routine creates a receive thread to receive communications and registers name as being ready to receive communications. Init returns a process information structure that is used as an argument to all other shared memory access and ipc routines.

4.1.2 Finish

```
void Finish(pinfo)
Pinfo *pinfo;
```

The Finish routine shuts down the ipc system. It terminates the receive capabilities of the system and removes the calling process' name from the list of registered systems. The pinfo argument is a pointer to the Pinfo structure returned by Init.

4.1.3 FindPartner

```
int FindPartner(pinfo, name)
Pinfo *pinfo;
char *name;
```

The FindPartner routine returns the internal identifier for the named partner. The identifier is used to name the destination of ipc communications. If a communications link to the named partner has not been established when FindPartner is called, one is made. The FindPartner routine returns a nonnegative identifier on success, -1 on failure. Failure will result if the partner is not currently registered for communications (see Init). The pinfo argument is a pointer to the Pinfo structure returned by Init.

4.2 Message Routines

Message routines may communicate with other processes on the local host or on a remote host. No distinction is made by the routines. Messages are sent to a named recipient and processed in the order in which they arrive. If the system is run in a capture mode (see Modes of Operation below), the internal ipc system traces the message information as it arrives and stores it in a file. No modifications to the user program are needed to capture trace information. Later the system may be run in a replay mode. The ipc system will then replay the user program guaranteeing that the messages the user program receives occur in the exact same order as when the program was run with the capture option. Again no changes to the user program are needed for replay.

4.2.1 MsgReceive

```
unsigned char *MsgReceive(pinfo, host, name, message_id,
                          identifier, type, time_out)
Pinfo *pinfo;
char *host;              /* output */
char *name;              /* output */
int *message_id;         /* output */
int *identifier;         /* output */
int *type;               /* output */
int time_out;
```

The MsgReceive routine returns the first available message for processing. If no message is currently available MsgReceive blocks until a message arrives or until the time_out period expires. The time_out argument specifies the time (in 100thes of a second) to wait for a message to arrive. A negative time_out indicates wait forever, a zero value tells the routine not to block. If the time_out period expires a NULL message is returned and the other output variables are unaffected. Otherwise, if host and/or name are non-NULL pointers (they should be character arrays of HOST_SIZE and MAX_NAME_SIZE bytes) MsgReceive writes into these arguments the host and the name of the sender of the message. If message_id and/or identifier and/or type are nonzero, MsgReceive writes into these arguments the message_id, the internal identifier and the type (NORMAL for non-blocking sends, EXPECT_REP for messages that expect a reply) of the received message. The internal identifier may be used to direct communication to the sender of the message and the message_id must be used in a reply to identify which message is being replied to. The pinfo argument is a pointer to the Pinfo structure returned by Init.

4.2.2 MsgCall

```
unsigned char *MsgCall(pinfo, identifier, message, message_length, time_out)
Pinfo *pinfo;
int identifier;
unsigned char *message;
int message_length;
int time_out;
```

4.2.3 MsgSend

```
void MsgSend(pinfo, identifier, message, message_length)
Pinfo *pinfo;
int identifier;
unsigned char *message;
int message_length;
```

4.2.4 MsgReply

```
void MsgReply(pinfo, identifier, message, message_length, message_id)
Pinfo *pinfo;
int identifier;
unsigned char *message;
int message_length;
int message_id;
```

The MsgCall, MsgSend and MsgReply routines all send a message of length message_length to the process with the given internal identifier. The MsgSend routine is a non-blocking send. The MsgReply routine is also a non-blocking send, but it indicates that the message is a reply to the message named by the message_id. The MsgCall routine expects a reply. MsgCall blocks until a reply arrives or until the time_out period expires. The time_out argument specifies the time (in 100thes of a second) to wait for a reply to arrive. A negative time_out indicates wait forever, a zero value tells the routine not to block. If the time_out period expires a NULL reply is returned. The pinfo argument is a pointer to the Pinfo structure returned by Init.

4.3 Shared Memory Access

All accesses to shared memory should be made within the scope of the routines listed below. The first step to safely allowing a variable to be shared among processes is to initialize the variable using ObjectInit. This allows the system to recognize the variable as being shared and gives the user an object structure containing the variable to pass to the access routines. If the user program is run in a capture mode, information concerning the values of objects during reads and writes is stored in a trace file. If the user later runs the program in a replay mode the system will guarantee that each shared memory read will obtain the same value as it did during the capture phase (see Modes of Operation below). For further information on constructing a multi-threaded program see [9].

4.3.1 ObjectInit

```
Object *ObjectInit(data, name)
char *data;
char *name;
```

The ObjectInit routine initializes an object for shared memory access. It must be called before any read or write accesses to the object are made. It need only be called once before any thread accesses the object. The data argument is a pointer to the data that will be stored in the shared memory object. The name argument is the name of the object and must be unique within the user program. The Object returned is the object that will be passed to the ReadEntry, ReadExit, WriteEntry and WriteExit routines described below. Use of these routines will allow the user to safely perform concurrent reads and exclusive writes to shared memory data. All references to the data in the object should be made via the Object->data field. NOTE: The Init and the ObjectInit routines are the only ones that do not take a pinfo argument.

4.3.2 ReadEntry

```
void ReadEntry(pinfo, object)
Pinfo *pinfo;
Object *object;
```

The ReadEntry routine must be called to initiate read access to shared memory. Upon return this routine allows concurrent reads to the shared memory object. The pinfo argument is a pointer to the process information structure returned by the Init routine. The object argument is the pointer to the shared memory object returned by the ObjectInit routine.

4.3.3 ReadExit

```
void ReadExit(pinfo, object)
Pinfo *pinfo;
Object *object;
```

The ReadExit routine must be called to conclude read access to shared memory. This routine indicates that the process is no longer reading the shared memory object. The pinfo argument is a pointer to the process information structure returned by the Init routine. The object argument is the pointer to the shared memory object returned by the ObjectInit routine.

4.3.4 WriteEntry

```
void WriteEntry(pinfo, object)
Pinfo *pinfo;
Object *object;
```

The WriteEntry routine must be called to initiate write access to shared memory. When it returns the user has exclusive access to the shared memory object until the WriteExit routine is called. The pinfo argument is a pointer to the process information structure returned by the Init routine. The object argument is the pointer to the shared memory object returned by the ObjectInit routine.

4.3.5 WriteExit

```
void WriteExit(pinfo, object)
Pinfo *pinfo;
Object *object;
```

The WriteExit routine must be called to conclude write access to shared memory. The call to WriteExit releases exclusive access to the shared memory object named by object (the pointer returned by the ObjectInit routine). The pinfo argument is a pointer to the process information structure returned by the Init routine.

4.4 Modes of Operation

The user can force the underlying ipc and memory access system to capture trace data relating to synchronization points within the user program. This may aid the user in debugging of the program. The user may observe the trace data to find the source of a bug or may force the program to replay, executing exactly as it did during the capture run. No timing related changes will result.

The user accomplishes capture and replay by setting the global MODE variable. This may be set dynamically at the start of the program or may be set prior to compilation by editing the configuration file (CAREconfig.h). MODE may be set one of the following six values:

- NORMAL - No capture or replay is performed.
- CAPTURE - Capture minimal information. Used in conjunction with the REPLAY mode to replay an entire program and all its interactions.
- FULL_CAPTURE - Capture complete information. Used in conjunction with REPLAY_ONE to replay a single process. In both CAPTURE and FULL_CAPTURE modes a file is created for each process. This file contains the captured data and is named process_name_rec, where process_name is the name supplied to the Init routine for the process.
- REPLAY - Replay an entire program. In this mode messages are sent and received just as they were in the capture phase. Suspending a process (e.g.

with a breakpoint) will suspend all other processes that share memory or receive messages from the suspended process. This method is used when the user needs to study multiple interacting processes during replay.

- REPLAY_ONE - Replay a single process. Each process is replayed from the capture file. No messages are sent or received. MsgReceive calls retrieve message information from the capture file and ReadEntry calls update the object with the proper values from the capture file. This method is useful if the user only needs to observe a single process at a time.

- CHECKPOINT - Perform FULL_CAPTURE and then REPLAY_ONE from the last checkpoint forward. During programs that execute for extended periods of time the user may wish to use the CHECKPOINT mode. This will force each process to save its state after tracing CP_COUNT synchronization items. CP_COUNT may be set dynamically at the start of a program or may be set prior to compilation in the CAREconfig.h file. The user may replay any process only from its latest checkpoint. This is accomplished by looking in the capture file for that process. The first line indicates a process identifier for a suspended process that, when awakened, will replay (in REPLAY_ONE fashion) the tail end of the process. A process is awakened by sending the shell the following command:

  ```
  kill -30 pid
  ```

  where 30 is the signal for SIGUSR1 and pid is the process identifier retrieved from the capture file. After re-execution of the process the user must type the following command to terminate the process:

  ```
  kill -9 pid
  ```

  If a process is terminated before it reaches an initial checkpoint, it may replay by setting the MODE to REPLAY_ONE and then executing.

Appendix A provides a simple client/server example. Appendix B illustrates how to use the different modes of operation. Appendix C shows sample capture logs for the different modes.

Chapter 5

Performance Characteristics

5.1 Nonblocking send/receive (64 byte messages)

Type	NORMAL	CAPTURE	FULL_CAPTURE
remote	1330/sec	1250/sec	530/sec
local	740/sec	690/sec	370/sec

The columns of the table indicate the mode of operation. The rows indicate if the tests were run on a single host (local) or from one host to another remote host. For comparison sake, straight TCP allows approximately 1330 64 byte messages per second on remote transfer. This implies that the computations of our system in NORMAL mode do not affect the speed of message passing. The CAPTURE mode does slow the system slightly and FULL_CAPTURE more significantly. This is due to the i/o that takes place during these modes. An improved approach to the logging output would aid in the performance of the capture modes. Local transfer was slower than remote due to the fact that the send and receive processes were sharing a single cpu.

5.2 Blocking calls (64 byte messages)

Type	NORMAL	CAPTURE	FULL_CAPTURE
remote	43/sec	43/sec	31/sec
local	43/sec	36/sec	31/sec

Blocking calls (sends that wait for replies) are much slower than their nonblocking counterparts. This is expected. The numbers represent the number of round trip calls. Each call consists of a send, a receive, followed by a reply send and receipt of the reply. Only then does the next call get executed. Users should avoid making a large number of calls if speed is important. We see here, as in the nonblocking table, that CAPTURE only slightly affects speed and that FULL_CAPTURE has a slightly greater impact.

5.3 Shared Memory Accesses

Type	NORMAL	CAPTURE	FULL_CAPTURE
Reads	9090/sec	2780/sec	1820/sec
Writes	26320/sec	1560/sec	N.A.

The columns of the table represent the mode of operation and the rows indicate if the test was for reads or writes. The speed of the system in NORMAL mode represents memory accesses surrounded by ReadEntry/ReadExit or WriteEntry/WriteExit pairs. The capture modes have a much greater relative impact on performance for shared memory accesses than for message passing due to the relative speed of the two operations. During the FULL_CAPTURE read tests, the object consisted of a single integer. FULL_CAPTURE mode is not applicable to writes because writes are no-ops in that mode.

CHECKPOINT modes were not tested because CHECKPOINT is a derivative of the FULL_CAPTURE mode.

Chapter 6

Conclusions

Debugging distributed programs is a complicated issue and tools are needed to aid in this task. A variety of systems have been proposed to help solve this problem. Each system we have examined has its own strengths and weaknesses. The effectiveness of the different systems varies depending on the problem presented.

We have observed that capture and replay looks like a promising mechanism to aid in distributed debugging as well as regression testing of parallel programs. We believe that placing the capture and replay abilities into the mechanisms that provide parallelism is a good approach to solving this problem. This method requires less user knowledge of the internal workings of the system and simplifies changes to the system. The implementation was not overly difficult and requires no external support from compilers, the operating system, etc...

In addition to our goal of creating a usable system we wanted to provide flexibility. All of the other capture and replay systems that we examined were limited in this aspect. We wanted to include all the positive features that the other systems provided without damaging our performance or comprehensibility. Thus we added the various modes of operation: capture and replay of interacting processes, capture and replay of individual processes, and checkpointing of processes that execute for extended periods of time. All of these built into a system that provides synchronous and asynchronous message passing and safe access to shared memory.

Our system does have limitations. We have described these limitations and possible means of over coming them. Perhaps the biggest problem with all the systems encountered, including our own, is one we have not addressed. That is intrusive logging. All forms of logging in some way affect the execution of the program under test by changing the timing of event interactions. This intrusion may mask some bugs that later appear when logging is turned off. This renders the debugging system useless in determining the cause of those bugs. In loosely coupled systems communication generally occurs infrequently due to the cost of sending a reliable message. The process of logging events adds a relatively small amount of overhead in this environment. In tightly couple systems the problem of logging intrusion is a more important issue as synchronization events are likely to occur often. Our system could be improved by adding buffered output capabilities. We could maintain a log buffer in memory and periodically write that buffer to disk. Upon unexpected program termination signal and interrupt handlers would have to be modified to dump the remaining buffered log information to the file. Another possible solution is to add a separate process to handle logging. On certain systems this might aid performance substantially.

We are pleased with what we have accomplished in this small arena, but there is a lot of work to be done. Hopefully, the future will bring better tools for static analysis and problem based error detection as well as better graphical aids for viewing synchronization event occurrences. This technology could be integrated with capture and replay type tools to create a truly useful environment for parallel debugging and testing.

Appendix A

Sample Programs

A.1 A Sample Client

client1.c

```
#include "CAREdefs.h"

/***************************
 * Sample Program - client *
 ***************************/
client1()
{
    int server_id, i;
    char buffer[100];
    Pinfo *pinfo;
    char *reply;

    /* Initialize */
    pinfo = Init("client1");

    /* Find the server */
    if ((server_id = FindPartner(pinfo, "server")) == -1)
    {
        printf("Error, server routine not accessible\n");
        exit(1);
    }

    /*
     * Send messages, alternate between nonblocking
     * and blocking sends
     */
    for (i = 1; i < 11; i++)
    {
        sprintf(buffer, "This is client1 message %d", i);
        if (i % 2)
            MsgSend(pinfo, server_id, buffer, strlen(buffer) + 1);
        else
        {
            reply = (char *)MsgCall(pinfo, server_id, buffer,
                                    strlen(buffer) + 1, -1);
            printf("Reply: %s\n", reply);
        }
    }
    /* Clean up */
    Finish(pinfo);
}

/********
 * MAIN *
 ********/
main()
{
    THREADgo(0, 0, client1, 0, 0, 16384, 8);
}
```

A.2 A Sample Client with Shared Memory

client2.c

```
#include "CAREdefs.h"

Object *obj;
/*************
 * Client 2a *
 *************/
client2a()
{
    Pinfo *pinfo;
    static struct timeval tv = {1, 0};
    int client2b_id, server_id, i;
    char buffer[100], *reply;

    /* Initialize ourselves */
    pinfo = Init("client2a");

    /* Wait a second for client2b to register */
    select(0, NULL, NULL, NULL, &tv);

    /* Find the client2b */
    if ((client2b_id = FindPartner(pinfo, "client2b")) == -1)
    {
        printf("Error, client2b routine not accessible\n");
        exit(1);
    }

    /* Find the server */
    if ((server_id = FindPartner(pinfo, "server")) == -1)
    {
        printf("Error, server routine not accessible\n");
        exit(1);
    }
    /* send messages to server and other client */
    for (i = 1; i < 11; i++)
    {
        sprintf(buffer, "This is client2a message %d", i);

        /* Nonblocking send to server */
        if (i % 2)
            MsgSend(pinfo, server_id, buffer, strlen(buffer) + 1);
        else
        {
            /* write into shared object */
            WriteEntry(pinfo, obj);
            *(int *)obj->data = i;
            WriteExit(pinfo, obj);

            /* Read shared object */
            ReadEntry(pinfo, obj);
            printf("2a Number pre-call: %d\n", *(int *)obj->data);
            ReadExit(pinfo, obj);

            /* Call client 2b */
            reply = (char *)MsgCall(pinfo, client2b_id, buffer,
                                    strlen(buffer)+1, -1);
            printf("2a received reply: %s\n", reply);

            /* Object value changes after call as set by client2b */
            ReadEntry(pinfo, obj);
            printf("2a Number post-call: %d\n", *(int *)obj->data);
            ReadExit(pinfo, obj);
        }
    }
    /* Clean up */
    Finish(pinfo);
}
```

```
/*************
 * Client 2b *
 *************/
client2b()
{
    Pinfo *pinfo;
    char name[MAX_NAME_SIZE], host[HOST_SIZE];
    int m_id, id, type, i;
    unsigned char *message, reply_buffer[100];

    pinfo = Init("client2b");

    /* Process messages from client2a and change object value */
    for (i = 50; i < 55; i++)
    {
        message = MsgReceive(pinfo,host,name,&m_id,&id,&type,-1);
        printf("client2b: from %s, id = %d: %s\n",
               name, m_id, message);
        if (type == EXPECT_REP)
        {
            sprintf(reply_buffer,
                    "c2b reply to message %d from %s on host %s",
                    m_id, name, host);

            /* Change object value */
            WriteEntry(pinfo, obj);
            *(int *)obj->data = i;
            WriteExit(pinfo, obj);

            /* Send a reply */
            MsgReply(pinfo, id, reply_buffer,
                     strlen(reply_buffer) + 1, m_id);
        }
        free(message);
    }
    /* This MsgReceive will time out after about 10 seconds */
    message = MsgReceive(pinfo,host,name,&m_id,&id,&type,1000);
    Finish(pinfo);
}

/************
 * Client 2 *
 ************/
client2()
{
    static int number = 9;
    static char *name = "Number";

    /* Initialize object once for shared access */
    obj = ObjectInit((char *)&number, name);

    /* Create two new threads */
    THREADcreate(client2a, 0, 0, 1, 16384, 8);
    THREADcreate(client2b, 0, 0, 1, 16384, 8);
}

/********
 * MAIN *
 ********/
main()
{
    THREADgo(0, 0, client2, 0, 0, 16384, 8);
}
```

A.3 A Sample Server

server.c

```
#include "CAREdefs.h"

/***************************
 * Sample Program - server *
 ***************************/

void server()
{
    unsigned char *message;
    char host[HOST_SIZE], name[MAX_NAME_SIZE];
    int mid, type, id;
    char reply_buffer[100];
    Pinfo *pinfo;

    /* Initialize */
    pinfo = Init("server");

    /*
     * Process messages as they arrive
     * If the message expects a reply send it one
     */
    while (1)
    {
        message = MsgReceive(pinfo,host,name,&mid,&id,&type,-1);
        printf("server: from %s, id = %d: %s\n",
               name, mid, message);
        if (type == EXPECT_REP)
        {
            sprintf(reply_buffer,
                    "reply to message %d from %s on host %s",
                    mid, name, host);
            MsgReply(pinfo, id, reply_buffer,
                    strlen(reply_buffer) + 1, mid);
        }
        free(message);
    }
}

/********
 * MAIN *
 ********/
main()
{
    THREADgo(0, 0, server, 0, 0, 16384, 8);
}
```

Appendix B

Steps for Using Capture and Replay

B.1 Normal Capture and Replay

- Edit the CAREconfig.h file and change the MODE line to

  int MODE = CAPTURE;

- Rebuild the client and server executables.
- Run the executables.
- Observe the capture files.
- Edit the CAREconfig.h file and change the MODE line to

  int MODE = REPLAY;

- Rebuild the client and server executables.
- Run the executables (repeatedly if desired)

B.2 Full Information Capture and Replay

- Edit the CAREconfig.h file and change the MODE line to

  ```
  int MODE = FULL_CAPTURE;
  ```

- Rebuild the client and server executables.
- Run the executables.
- Observe the capture files.
- Edit the CAREconfig.h file and change the MODE line to

  ```
  int MODE = REPLAY_ONE;
  ```

- Rebuild the client and server executables.
- Run any executable (repeatedly if desired)

B.3 Checkpointing

- Edit the CAREconfig.h file and change the MODE line to

  ```
  int MODE = CHECKPOINT;
  ```

- Rebuild the client and server executables.
- Run the executables.
- Obtain a process identifier from the first line of a capture file.
- Give the shell the command: kill -30 pid
  This will execute the process from the last checkpoint forward.

- Give the shell the command: kill -9 pid
  This will kill the checkpointed process.
- Repeat the last three steps as desired.
- Kill all outstanding processes with the command: kill -9 pid

Appendix C

Sample Capture Logs

These logs were created by running the sample server program and the two sample client programs simultaneously. The server and client2 were run on the same host (dooley), client1 was run on a seperate host (harpo).

We do not include CHECKPOINT logs in the following sections because the contents are the same as the tail end of a FULL_CAPTURE log.

C.1 Client1 Log for CAPTURE

client1_rec

```
reply from server on host dooley to message 2
reply from server on host dooley to message 4
reply from server on host dooley to message 6
reply from server on host dooley to message 8
reply from server on host dooley to message 10
```

C.2 Client1 Log for FULL_CAPTURE

client1_rec

```
reply from server on host dooley to message 2 len 46 -
    reply to message 2 from client1 on host harpo
reply from server on host dooley to message 4 len 46 -
    reply to message 4 from client1 on host harpo
reply from server on host dooley to message 6 len 46 -
    reply to message 6 from client1 on host harpo
reply from server on host dooley to message 8 len 46 -
    reply to message 8 from client1 on host harpo
reply from server on host dooley to message 10 len 47 -
    reply to message 10 from client1 on host harpo
```

C.3 Client2 Logs for CAPTURE

client2a_rec

```
wrote object Number version 1 total readers 0
read object Number version 2
reply from client2b on host dooley to message 2
read object Number version 3
wrote object Number version 3 total readers 1
read object Number version 4
reply from client2b on host dooley to message 4
read object Number version 5
wrote object Number version 5 total readers 1
read object Number version 6
reply from client2b on host dooley to message 6
read object Number version 7
wrote object Number version 7 total readers 1
read object Number version 8
reply from client2b on host dooley to message 8
read object Number version 9
wrote object Number version 9 total readers 1
read object Number version 10
reply from client2b on host dooley to message 10
read object Number version 11
```

client2b_rec

```
from client2a on host dooley id 2
wrote object Number version 2 total readers 1
from client2a on host dooley id 4
wrote object Number version 4 total readers 1
from client2a on host dooley id 6
wrote object Number version 6 total readers 1
from client2a on host dooley id 8
wrote object Number version 8 total readers 1
from client2a on host dooley id 10
wrote object Number version 10 total readers 1
from - on host - id 0
```

C.4 Client2 Log for FULL_CAPTURE

client2a_rec

```
read object Number version 2 contents - 2
reply from client2b on host dooley to message 2 len 52 -
    c2b reply to message 2 from client2a on host dooley
read object Number version 3 contents - 50
read object Number version 4 contents - 4
reply from client2b on host dooley to message 4 len 52 -
    c2b reply to message 4 from client2a on host dooley
read object Number version 5 contents - 51
read object Number version 6 contents - 6
reply from client2b on host dooley to message 6 len 52 -
    c2b reply to message 6 from client2a on host dooley
read object Number version 7 contents - 52
read object Number version 8 contents - 8
reply from client2b on host dooley to message 8 len 52 -
    c2b reply to message 8 from client2a on host dooley
read object Number version 9 contents - 53
read object Number version 10 contents - 10
reply from client2b on host dooley to message 10 len 53 -
    c2b reply to message 10 from client2a on host dooley
read object Number version 11 contents - 54
```

client2b_rec

```
from client2a on host dooley id 2 index 0 type 2 len 27 -
    This is client2a message 2
from client2a on host dooley id 4 index 0 type 2 len 27 -
    This is client2a message 4
from client2a on host dooley id 6 index 0 type 2 len 27 -
    This is client2a message 6
from client2a on host dooley id 8 index 0 type 2 len 27 -
    This is client2a message 8
from client2a on host dooley id 10 index 0 type 2 len 28 -
    This is client2a message 10
from - on host - id 0 index 0 type 0 len 0 -
```

C.5 Server Log for CAPTURE

server_rec

```
from client1 on host harpo id 1
from client1 on host harpo id 2
from client1 on host harpo id 3
from client2a on host dooley id 1
from client1 on host harpo id 4
from client1 on host harpo id 5
from client2a on host dooley id 3
from client1 on host harpo id 6
from client1 on host harpo id 7
from client2a on host dooley id 5
from client2a on host dooley id 7
from client2a on host dooley id 9
from client1 on host harpo id 8
from client1 on host harpo id 9
from client1 on host harpo id 10
```

C.6 Server Log for FULL_CAPTURE

server_rec

```
from client2a on host dooley id 1 index 0 type 0 len 27 -
    This is client2a message 1
from client2a on host dooley id 3 index 0 type 0 len 27 -
    This is client2a message 3
from client2a on host dooley id 5 index 0 type 0 len 27 -
    This is client2a message 5
from client2a on host dooley id 7 index 0 type 0 len 27 -
    This is client2a message 7
from client1 on host harpo id 1 index 1 type 0 len 26 -
    This is client1 message 1
from client2a on host dooley id 9 index 0 type 0 len 27 -
    This is client2a message 9
from client1 on host harpo id 2 index 1 type 2 len 26 -
    This is client1 message 2
from client1 on host harpo id 3 index 1 type 0 len 26 -
    This is client1 message 3
from client1 on host harpo id 4 index 1 type 2 len 26 -
    This is client1 message 4
from client1 on host harpo id 5 index 1 type 0 len 26 -
    This is client1 message 5
from client1 on host harpo id 6 index 1 type 2 len 26 -
    This is client1 message 6
from client1 on host harpo id 7 index 1 type 0 len 26 -
    This is client1 message 7
from client1 on host harpo id 8 index 1 type 2 len 26 -
    This is client1 message 8
from client1 on host harpo id 9 index 1 type 0 len 26 -
    This is client1 message 9
from client1 on host harpo id 10 index 1 type 2 len 27 -
    This is client1 message 10
```

Bibliography

[1] *The Ada Programming Language Reference Manual,* US Department of Defense, ANSI/MILSTD 1815A Document, US Government Printing Office (1983)

[2] William F. Appelbe and Charles E. McDowell, "Integrating Tools for Debugging and Developing Multitasking Programs," *SIGPLAN Notices Vol. 24, No. 1,* pp. 78-88 (January 1989)

[3] Peter Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *SIGPLAN Notices Vol. 24, No. 1,* pp. 11-22 (January 1989)

[4] Channing B. Brown and Stuart I. Feldman, "IGOR: A System for Program Debugging via Reversible Execution," *SIGPLAN Notices Vol. 24, No. 1,* pp. 112-123 (January 1989)

[5] Richard H. Carver, Evelyn E. Obaid, K. C. Tai, "Deterministic Execution Debugging of Concurrent Ada Programs," *IEEE 0730-3157/89/0000/0102,* pp. 102-109 (1989)

[6] Thomas L. Casavant, James E. Lumpp, Jr., Dan C. Marinescu, Howard Jay Siegel, "A Model for Monitoring and Debugging Parallel and Distributed Software," *IEEE 0730-3157/89/0000/0081,* pp. 81-88 (1989)

[7] D. Cheriton, Michael A. Malcolm, Lawrence S. Melen and Gary R. Sager, "Thoth, a Portable Real-time Operating System," *Communications of the ACM, Vol. 22, No. 2,* pp. 105-115 (February 1979)

[8] Lori A. Clarke and Douglas L. Long, "Task Interaction Graphs for Concurrency Analysis," *Proceedings, Second Workshop On Software Testing, Verification, and Analysis,* pp. 134-135 (July 1988)

[9] Thomas W. Doeppner Jr., "A Threads Tutorial," Technical Report (CS-87-06), Department of Computer Science, Brown University (September 1988)

[10] Perry A. Emrath and David A. Padua, "Automatic Detection of Nondeterminacy in Parallel Programs," *SIGPLAN Notices Vol. 24, No. 1,* pp. 89-99 (January 1989)

[11] David Helmbold and David Luckham, "Debugging Ada Tasking Programs," *IEEE Software, Vol. 2, No. 2,* pp. 47-57 (March 1985)

[12] Wenway Hseush and Gail E. Kaiser, "Modeling Concurrency in Parallel Debugging," *ACM 089791-350-7/90/0003/0011,* pp. 11-20 (1990)

[13] Steven P. Reiss, "Integration Mechanisms in the FIELD Environment," Technical Report (CS-88-18), Department of Computer Science, Brown University (October 1988)

[14] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1978)

[15] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM, Vol 21, No. 7* pp. 558-565 (July 1978)

[16] Thomas J. LeBlanc and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers, Vol. C-36, No. 4* pp. 471-481 (April 1987)

[17] Mark A. Linton and Douglas Z. Pan, "Supporting Reverse Execution of Parallel Programs," *SIGPLAN Notices Vol. 24, No. 1,* pp. 124-129 (January 1989)

[18] Sol M. Shatz, "Analysis of Concurrent Software," *IEEE 0730-3157/89/0000/0060,* pp. 60-61 (1989)

[19] K. C. Tai, "Testing of Concurrent Software," *IEEE 0730-3157/89/0000/0062,* pp. 62-64 (1989)

[20] Richard N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM, Vol. 26, No. 5,* pp. 362-376 (May 1983)

[21] Larry D. Wittie, "Debugging Distributed C Programs by Real Time Replay," *SIGPLAN Notices Vol. 24, No. 1,* pp. 57-67 (January 1989)

[22] Stephen R. Schach and Nancy J. Wahl, "A Methodology and Distributed Tool for Debugging Dataflow Programs," *Proceedings, Second Workshop On Software Testing, Verification, and Analysis,* pp. 98-105 (July 1988)

[23] Stewart N. Weiss, "A Formal Framework for the Study of Concurrent Program Testing," *Proceedings, Second Workshop On Software Testing, Verification, and Analysis,* pp. 106-113 (July 1988)

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <thread.h>
#include <math.h>
#include <signal.h>

#ifndef def
#define def

/*
 * Header information
 */
#define REPLY_BYTE      0     /* Byte that indicates if message is a reply */
#define NORMAL          0     /* Message is a normal send                  */
#define REPLY           1     /* Message is a reply                        */
#define EXPECT_REP      2     /* Message is a call                         */
#define TIME_BYTE       1     /* Location of message id                    */
#define TIME_STAMP_SIZE 4     /* Size of message id                        */
#define HOST_BYTE       5     /* Location of sending host name             */
#define HOST_SIZE       11    /* Max length for host name                  */
#define NAME_BYTE       16    /* Location of senders name                  */
#define MAX_NAME_SIZE   16    /* Max size for process name                 */
#define HEADER_SIZE     32    /* Size of entire message header             */

/*
 * Modes of operation
 */
#define REPLAY          1
#define CAPTURE         2
#define REPLAY_ONE      3
#define FULL_CAPTURE    4
#define CHECKPOINT      5

/*
 * Other defs
 */
#define TRUE            1     /* True and false defs                       */
#define FALSE           0
#define NUM_CON         16    /* Initial number of connections allowed     */
#define READ_SIZE       1024  /* Size for attempted reads                  */
#define EOM_CHAR        '\3'  /* Special end of message character          */
#define OBJ_NAME_SIZE   16    /* Max size for an object name               */

/*
 * Macros
 */
#define MsgSend(P,I,M,L)     (void)send_message((P),(I),(M),(L),NORMAL,-1)
#define MsgReply(P,I,M,L,T)  (void)send_message((P),(I),(M),(L),REPLY,(T))
#define get_rep(M) M[0] == '\0' ? NORMAL : (M[0] == 'R' ? REPLY : EXPECT_REP)

#endif

/*
 * Include structure definitions
 */
#include "CAREstructs.h"
```

```
/*
 * Forward declarations
 */
Pinfo *Init();
unsigned char *MsgReceive();
unsigned char *receive_rep();
unsigned char *MsgCall();
char *get_host();
Object *ObjectInit();
```

```
/* Message buffer */
typedef struct _mbuf {
    struct _mbuf *next;
    struct _mbuf *prev;
    unsigned char *buffer;   /* message contents */
    int buf_size;            /* message size     */
    int index;               /* connection id    */
    int time_stamp;          /* message id       */
    int type;                /* message type     */
} MBUF;

/* Message queue */
typedef struct _queue {
    MBUF *head;
    MBUF *tail;
} QUEUE;

/* Shared memory object */
typedef struct _object {
    THREAD_MONITOR mon, mon_write;      /* CREW access monitors              */
    int active_read, total_read;        /* Active readers and total readers */
    int version;                        /* Object version                    */
    char *data;                         /* Pointer to object data            */
    char name[OBJ_NAME_SIZE];           /* Object name                       */
} Object;

/* Connection information */
typedef struct _ci {
    int socket;                  /* Socket to send to */
    char host[HOST_SIZE];        /* Remote host       */
    char name[MAX_NAME_SIZE];    /* Connector's name  */
    int filled;                  /* True if this structure contains data */
    char p_message[READ_SIZE];   /* Partially read message on this con.  */
    int  p_read;                 /* Size of partially read message       */
} CI;

/* Process information */
typedef struct _pinfo {
    QUEUE My_Queue, My_Rep_Q;      /* Received message and reply queues */
    char  My_Name[MAX_NAME_SIZE];  /* Process name                      */
    char  My_Host_Name[HOST_SIZE]; /* Host name                         */
    CI   Connection[NUM_CON];      /* Our connections                   */
    THREAD_MONITOR q_mon;          /* Queue monitor                     */
    THREAD_MONITOR rq_mon;         /* Reply buffer monitor              */
    THREAD My_Rec_Thread;          /* Receive thread                    */
    fd_set RFDS;                   /* Read file descriptors             */
    int Num_Cons;                  /* Current number of connections     */
    FILE *fp;                      /* File descriptor for logging       */
    int cp_count;                  /* Number of checkpoint events       */
    int mode;                      /* Mode of operation                 */
    int pid;                       /* pid of checkpointed process       */
    int halt;                      /* THREADmurder broken hack          */
} Pinfo;

/* Arguments passed to receive thread */
typedef struct _wfa {
    int sock;                    /* Our own socket information */
    struct sockaddr_in *name;
    Pinfo *pinfo;                /* Our process information    */
} WFA;

/* Globals */
THREAD_MONITOR ghn;              /* For gethostname safety                   */
```

```c
#include "CAREdefs.h"
#include "CAREconfig.h"
int HAND = TRUE;
/****************************************************************************
 * SENDERS  SIDE                                                            *
 ****************************************************************************/
/********************
 * Connect Routine *
 ********************/
int con(host, port)
char *host;                 /* host to send to */
short port;                 /* port to send to */
{
    int sock;
    struct sockaddr_in name;
    struct hostent *h;
    THREAD_MONITOR_BLOCK tmbn;

/*
 * Get a socket to send on
 */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }
/*
 * Set up sockaddr structure for connect and then connect
 */
    THREADmonitorentry(ghn, &tmbn);
    h = gethostbyname(host);
    bcopy(h->h_addr, &(name.sin_addr.s_addr), h->h_length);
    THREADmonitorexit(ghn);
    name.sin_family = htons(AF_INET);
    name.sin_port = htons(port);
    if (connect(sock, &name, sizeof(name)) == -1)
    {
        perror("connect");
        exit(1);
    }
    return(sock);
}
```

```c
/******************
 * Locate Partner *
 ******************/
int FindPartner(pinfo, routine)
Pinfo *pinfo;
char *routine;
{
    static int ST_Index = -1;
    FILE *fopen(), *fp;
    char host[HOST_SIZE];
    short port;
    int sock, i;
    char file_name[MAX_NAME_SIZE + 7];
    struct timeval tv;

    tv.tv_sec = 0;
    tv.tv_usec = 100000;

/*
 * First check if we are already connected
 */
    for (i = 0; i <= ST_Index; i++)
    {
        if (strcmp(routine, pinfo->Connection[i].name) == 0)
            return(i);
    }
/*
 * Not connected so try to find partner
 */
    sprintf(file_name, ".%s", routine);
    if ((fp = fopen((char *)file_name, "r")) == NULL)
        return (-1);   /* Not opened return error */
    else
    {
        if ((fscanf(fp, "%s %hd", host, &port)) != 2)
        {
            fclose(fp);
            return (-1);     /* Poor format return error */
        }
        else
        {
            fclose(fp);
            sock = con(host, port); /* connect to partner */
            pinfo->Num_Cons++;
            FD_SET(sock, &pinfo->RFDS);    /* show our interest in reading */
            select(0, NULL, NULL, NULL, &tv);  /* make sure it shows up */
/*
 * Set up array with socket, host and name information
 */
            ST_Index++;
            pinfo->Connection[ST_Index].socket = sock;
            strcpy(pinfo->Connection[ST_Index].host, host);
            strcpy(pinfo->Connection[ST_Index].name, routine);
/**
 ** This is where we should send initial header information
 ** and check size and realloc if connection array is full
 **/
            return(ST_Index);
        }
    }
}
```

```
/******************
 * Send a message *
 ******************/
int send_message(pinfo, index, message, mlen, type, time_to)
Pinfo *pinfo;
int index;
char *message;
int mlen;
int type;       /* NORMAL, REPLY, EXPECT_REP */
int time_to;    /* Time stamp of message we are replying to - replies only */
{
    int num_sent;
    int total_sent;
    fd_set wfds;
    int sock;
    char *host;
    char *name;
    static int time_stamp = 0;
    int ntime;
    unsigned char *tmp_message;
    short port;

/*
 * This is a no-op for replay one
 */
    if (MODE == REPLAY_ONE)
        return(0);

/*
 * TRANSFORM THE MESSAGE, include My_Name, My_Host_Name, Time
 * Try to do something different than the 'R' in the first character
 */
    if ((tmp_message = (unsigned char *)malloc(HEADER_SIZE + mlen)) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    if (type == NORMAL)
        tmp_message[REPLY_BYTE] = '\0';
    else if (type == REPLY)
        tmp_message[REPLY_BYTE] = 'R';
    else
        tmp_message[REPLY_BYTE] = 'E';
    if (type == REPLY)
        ntime = htonl(time_to);
    else
        ntime = htonl(++time_stamp);
    bcopy(&ntime, &tmp_message[TIME_BYTE], 4);
/**
 ** The host and name can be sent once at connection time
 ** This would reduce the size of the header
 **/
    strcpy(&tmp_message[HOST_BYTE], pinfo->My_Host_Name);
    strcpy(&tmp_message[NAME_BYTE], pinfo->My_Name);
    bcopy(message, &tmp_message[HEADER_SIZE], mlen);
    tmp_message[HEADER_SIZE + mlen] = EOM_CHAR;
/*
 * Send the message
 * Make sure whole message gets sent
 */
    total_sent = 0;
    while (total_sent < mlen + HEADER_SIZE + 1)
    {
        FD_ZERO(&wfds);
        FD_SET(pinfo->Connection[index].socket, &wfds);
        select(ulimit(4, 0), NULL, &wfds, NULL, NULL);
        num_sent = send(pinfo->Connection[index].socket,
                        &tmp_message[total_sent],
                        mlen + HEADER_SIZE + 1 - total_sent, 0);
        total_sent += num_sent;
/**
 ** This code can be used to check for broken messages
        if (num_sent != mlen + HEADER_SIZE + 1)
            printf("num_sent - %d, total - %d, mlen- %d\n", num_sent,
                   total_sent,
                   mlen + HEADER_SIZE + 1);
**/
    }
    return(time_stamp);
}
```

```
/***************************************************************************
 * RECEIVERS  SIDE                                                         *
 ***************************************************************************/
/*******************************
 * Accept a connection routine *
 *******************************/
void accept_connect(pinfo, l_sock)
Pinfo *pinfo;
int l_sock;    /* listening socket */
{
    int new_sock, i;
    struct sockaddr addr;
    int addrlen = sizeof(addr);
/*
 * Accept a new connection
 */
    if ((new_sock = accept(l_sock, &addr, &addrlen)) < 0)
    {
        perror("accept");
        exit(1);
    }
    pinfo->Connection[pinfo->Num_Cons].socket = new_sock;
/**
 ** We don't have all the information on this connection yet
 ** If we sent an initial message at connect time we could avoid this
 **/
    pinfo->Connection[pinfo->Num_Cons].filled = FALSE;
    FD_SET(new_sock, &pinfo->RFDS);
}
```

```
/***************************
 * Process message routine *
 ***************************/
void queue_message(pinfo, buffer, buf_size, index, time_stamp, type)
Pinfo *pinfo;
char *buffer;
int buf_size;
int index;
int time_stamp;
int type;
{
    MBUF *mbuf;
    THREAD_MONITOR monitor;
    THREAD_MONITOR_BLOCK mon_block;
    QUEUE *queue;
    int i;
/*
 * Build a dynamic linked list of messages
 * Reply type messages go in seperate queue
 */
    if ((mbuf = (MBUF *)malloc(sizeof(MBUF))) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    if ((mbuf->buffer = (unsigned char *)malloc(buf_size)) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    (void)bcopy(buffer, mbuf->buffer, buf_size);
    mbuf->buf_size = buf_size;
    mbuf->index = index;
    mbuf->time_stamp = time_stamp;
    mbuf->type = type;

    if (type == REPLY)  /* reply, use reply queue */
    {
        monitor = pinfo->rq_mon;
        queue = &pinfo->My_Rep_Q;
    }
    else /* normal message, use regular queue */
    {
        monitor = pinfo->q_mon;
        queue = &pinfo->My_Queue;
    }
    THREADmonitorentry(monitor, &mon_block);  /* exclusive access to queue */
    if (queue->head == NULL)  /* queue is empty */
    {
        mbuf->prev = NULL;
        queue->head = mbuf;
    }
    else  /* queue not empty put at queue tail */
    {
        mbuf->prev = queue->tail;
        queue->tail->next = mbuf;
    }
    mbuf->next = NULL;
    queue->tail = mbuf;
    THREADmonitorexit(monitor);
}
```

```
/*************************
 * Get a message routine *
 *************************/
void get_message(pinfo, con_idx)
Pinfo *pinfo;
int con_idx;
{
    CI *ci;    /* connection array information */
    int num_read, ptr = 0, tb_ptr, time_stamp;
    char buffer[READ_SIZE], tmp_buff[READ_SIZE];

    ci = &(pinfo->Connection[con_idx]);
/*
 * Redo partially read header or message if any
 */
    tb_ptr = ci->p_read;
    if (tb_ptr != 0) /* partial read before */
    {
        bcopy(ci->p_message, tmp_buff, tb_ptr);
        ci->p_read = 0;
    }

/*
 * We do a limited read here so that we are fair, don't want one connection
 * to be the only one talking
 */

    num_read = read(ci->socket, buffer, READ_SIZE);
    if (num_read == 0)
    {
        printf("Connection dropped\n");
        /** FIX UP THE ARRAY TOO ???? **/
        FD_CLR(ci->socket, &pinfo->RFDS);
        return;
    }

/**
 ** This would be unnecessary if we send initial info once at connect time
 **/
    if (ci->filled == FALSE)
    {
        strcpy(ci->host, &buffer[HOST_BYTE]);
        strcpy(ci->name, &buffer[NAME_BYTE]);
        ci->filled = TRUE;
    }

    while (1)
    {
/*
 * Read a header
 */
        if (tb_ptr < HEADER_SIZE)
        {
            while ((ptr < num_read) && (tb_ptr < HEADER_SIZE))
            {
                tmp_buff[tb_ptr] = buffer[ptr];
                ptr++; tb_ptr++;
            }

            if ((tb_ptr != HEADER_SIZE) || (ptr == num_read - 1))
            {
                ci->p_read = tb_ptr;
                bcopy(tmp_buff, ci->p_message, tb_ptr);
                return;
            }
        }


/*
 * Read a message
 */
        while (ptr < num_read)
        {
            if (buffer[ptr] != EOM_CHAR)
            {
                tmp_buff[tb_ptr] = buffer[ptr];
                ptr++; tb_ptr++;
            }
            else
            {
                bcopy(&tmp_buff[TIME_BYTE], &time_stamp, TIME_STAMP_SIZE);
                time_stamp = ntohl(time_stamp);
/*
 * If we get a whole message, put it in the queue
 */
                queue_message(pinfo,
                              &tmp_buff[HEADER_SIZE], tb_ptr - HEADER_SIZE,
                              con_idx, time_stamp, get_rep(tmp_buff));
                tb_ptr = 0;
                ptr++;
                break; /* from inside while */
            }
        }
        if (tb_ptr != 0) /* save partial message */
        {
            ci->p_read = tb_ptr;
            bcopy(tmp_buff, ci->p_message, tb_ptr);
            return;
        }

        if (ptr == num_read) /* done with this read */
            return;
    }
}
```

```
/******************
 * Receive thread *
 ******************/
void *receive_thread(wfa)
WFA *wfa;
{
    int sock;                  /* socket to wait on          */
    struct sockaddr_in *name;  /* name info for receive      */
    int size = sizeof(*name);
    fd_set tmp_rfds;
    int num_select, max_fds, idx;
    struct timeval tv;
    Pinfo *pinfo;

    sock = wfa->sock;
    name = wfa->name;
    pinfo = wfa->pinfo;
    tv.tv_sec = 0;
    tv.tv_usec = 100000;    /* .1 seconds */
    max_fds = ulimit(4, 0);
    while (1)
    {
/*
 * Check for new connections initiated by our side every .1 seconds
 */
        tmp_rfds = pinfo->RFDS;
        num_select = select(max_fds, &tmp_rfds, NULL, NULL, &tv);

        /** Hack since THREADmurder doesn't seem to work **/
        if ((pinfo->halt == TRUE) || (MODE == REPLAY_ONE))
            return;
/**
 ** If check pointing in main thread pause here until done
 ** Basically it would be a good idea to stop i/o here when checkpointing
 **/
        if (num_select < 0)
        {
            perror("select");
            exit(1);
        }
/*
 * If there is a new connection pending accept it
 * If there is a message available for reading on any connection, read it
 */
        if (num_select > 0)
        {
            if (FD_ISSET(sock, &tmp_rfds))
            {
                accept_connect(pinfo, sock);
                pinfo->Num_Cons++;
                num_select--;
            }
            for (idx = 0; (idx < pinfo->Num_Cons) && (num_select > 0); idx++)
            {
                if (FD_ISSET(pinfo->Connection[idx].socket, &tmp_rfds))
                {
                    get_message(pinfo, idx);
                    num_select--;
                }
            }
        }
    }
}
```

```c
/*******************
 * Receive routine *
 *******************/
unsigned char *MsgReceive(pinfo, host, name, time_stamp, index, type, time_out)
Pinfo *pinfo;
char *host;
char *name;
int *time_stamp;
int *index;
int *type;
int time_out;    /* 100thes of seconds */
{
    unsigned char *buffer;
    struct timeval tv;
    static struct timeval wf = {32767, 0};
    MBUF *temp;
    THREAD_MONITOR_BLOCK qmb;
    char file_name[MAX_NAME_SIZE + 5];
    MBUF *curr;
    char exp_name[MAX_NAME_SIZE];
    char exp_host[HOST_SIZE];
    int exp_time, exp_len, i, exp_index, exp_type;
    static int is_open = 0;
    int count = 0;
    QUEUE *My_Queue;

/*
 * If this is a replay one just get the message from the file
 */
    if (MODE == REPLAY_ONE)
    {
        if (fscanf(pinfo->fp,
                   "from %s on host %s id %d index %d type %d len %d - ",
                   exp_name, exp_host, &exp_time, &exp_index, &exp_type,
                   &exp_len) == -1) /* receive blocks forever */
            while(1)
                select(0, NULL, NULL, NULL, &wf); /* never return */
/*
 * If a blank message was written return NULL
 */
        if (exp_name[0] == '-')
            return(NULL);

/*
 * We have a real message, read it in, then return
 */
        if ((buffer = (unsigned char *)malloc(exp_len)) == NULL)
        {
            printf("Memory allocation error\n");
            exit(1);
        }
        for (i = 0; i < exp_len; i++)
            fscanf(pinfo->fp, "%c", &buffer[i]);
        fscanf(pinfo->fp, "\n");
        if (host != NULL) /* return host info */
        {
            if ((host = (char *)malloc(HOST_SIZE)) == NULL)
            {
                printf("Memory allocation error\n");
                exit(1);
            }
            strcpy(host, exp_host);
        }
        if (name != NULL) /* return name info */
        {
            if ((name = (char *)malloc(MAX_NAME_SIZE)) == NULL)
            {
                printf("Memory allocation error\n");
                exit(1);
            }
            strcpy(name, exp_name);
        }
        if (time_stamp != 0) /* return time info */
            *time_stamp = exp_time;
        if (index != 0) /* return index */
            *index = exp_index;
        if (type != 0) /* return type */
            *type = exp_type;
        return(buffer);
    }

/*
 * NOT REPLAY_ONE
 * Set process information into local variables
 */
    My_Queue = &pinfo->My_Queue;

    tv.tv_sec = 0;
    tv.tv_usec = 100;

/*
 * Normal replay
 */
    if (MODE == REPLAY)
    {
/*
 * Read message from file
 */
        fscanf(pinfo->fp, "from %s on host %s id %d\n",
               exp_name, exp_host, &exp_time);

/*
 * If a blank message was written return NULL
 */
        if (exp_name[0] == '-')
            return(NULL);

/*
 * Check if message is in queue
 */
        while (1) /* keep checking until we find it */
        {
            THREADmonitorentry(pinfo->q_mon, &qmb);
            if (My_Queue->head == NULL)
            {
                THREADmonitorexit(pinfo->q_mon);
                select(0, NULL, NULL, NULL, &tv);
            }
            else
            {
/*
 * While not at the end of the queue and not message found keep looking
 */
                curr = My_Queue->head;
                while (curr != NULL)
                {
                    if ((strcmp(exp_name, pinfo->Connection[curr->index].name) == 0) &&
                (exp_time == curr->time_stamp) &&
                (strcmp(exp_host, pinfo->Connection[curr->index].host) == 0))
            {
                if ((buffer = (unsigned char *)malloc(curr->buf_size)) == NULL)
                {
                    printf("Memory allocation error\n");
                    exit(1);
                }
                if (host != NULL) /* return host info */
                {
                    if ((host = (char *)malloc(HOST_SIZE)) == NULL)
                    {
                        printf("Memory allocation error\n");
                        exit(1);
                    }
                    strcpy(host, pinfo->Connection[curr->index].host);
                }
                if (name != NULL) /* return name info */
                {
                    if ((name = (char *)malloc(MAX_NAME_SIZE)) == NULL)
                    {
                        printf("Memory allocation error\n");
                        exit(1);
                    }
                    strcpy(name, pinfo->Connection[curr->index].name);
                }
                if (time_stamp != 0) /* return time info */
                    *time_stamp = curr->time_stamp;
                if (index != 0) /* return index */
                    *index = curr->index;
                if (type != 0) /* return type */
                    *type = curr->type;
                (void)bcopy(curr->buffer, buffer, curr->buf_size);
                free(curr->buffer);
/*
 * fix up the pointers in the queue
 */
                if (curr->prev == NULL) /* if it is the head */
                {
                    if (curr->next == NULL) /* and the tail too */
                    {
                        My_Queue->head = NULL;
                        My_Queue->tail = NULL;
                    }
                    else /* it's just the head */
                    {
                        My_Queue->head = curr->next;
                        My_Queue->head->prev = NULL;
                    }
                }
                else if (curr->next == NULL) /* it's just the tail */
                {
                    My_Queue->tail = curr->prev;
                    My_Queue->tail->next = NULL;
                }
                else /* it's in the middle */
                {
                    curr->next->prev = curr->prev;
                    curr->prev->next = curr->next;
                }
                free(curr);
                THREADmonitorexit(pinfo->q_mon);
                return(buffer);
            }
                    else
                        curr = curr->next;
                }
                THREADmonitorexit(pinfo->q_mon);
                select(0, NULL, NULL, NULL, &tv);
            }
        }
    }
/*
 * Not a replay mode at all
 */
    else
    {
        if (time_out > 0)
            time_out *= TimeAdjust; /* adjust the time out to the system */
        while (1)
        {
            THREADmonitorentry(pinfo->q_mon, &qmb);
            if (My_Queue->head == NULL) /* queue is empty */
            {
                THREADmonitorexit(pinfo->q_mon);
                select(0, NULL, NULL, NULL, &tv);
/*
 * if time_out expires return, if it's capture write a blank message to the
 * file before returning
 */
                if (count == time_out)
                {
                    if (MODE == CAPTURE)
                        fprintf(pinfo->fp,"from - on host - id 0\n");
                    else if ((MODE == FULL_CAPTURE) || (MODE == CHECKPOINT))
                        fprintf(pinfo->fp,
                                "from - on host - id 0 index 0 type 0 len 0 - ");
                    return(NULL);
                }
                count++;
            }
            else /* There is a message in the queue */
            {
                if ((buffer =
                     (unsigned char *)malloc(My_Queue->head->buf_size)) == NULL)
                {
                    printf("Memory allocation error\n");
                    exit(1);
                }
                if (host != NULL) /* return host info */
                    strcpy(host, pinfo->Connection[My_Queue->head->index].host);
                if (name != NULL) /* return name info */
                    strcpy(name, pinfo->Connection[My_Queue->head->index].name);
                if (time_stamp != 0) /* return time info */
                    *time_stamp = My_Queue->head->time_stamp;
                if (index != 0) /* return index */
                    *index = My_Queue->head->index;
                if (type != 0) /* return type */
                    *type = My_Queue->head->type;
                (void)bcopy(My_Queue->head->buffer, buffer,
                            My_Queue->head->buf_size);
/*
 * If we are in a cpature mode log the message info
 */
                if (MODE == CAPTURE)
                {
                    fprintf(pinfo->fp,"from %s on host %s id %d\n",
                            pinfo->Connection[My_Queue->head->index].name,
                    pinfo->Connection[My_Queue->head->index].host,
                    My_Queue->head->time_stamp);
            }
            else if ((MODE == FULL_CAPTURE) ||
                     (MODE == CHECKPOINT))
            {
                fprintf(pinfo->fp,
                    "from %s on host %s id %d index %d type %d len %d - ",
                    pinfo->Connection[My_Queue->head->index].name,
                    pinfo->Connection[My_Queue->head->index].host,
                    My_Queue->head->time_stamp, My_Queue->head->index,
                    My_Queue->head->type, My_Queue->head->buf_size);

                /* print the message now */
                for (i = 0; i < My_Queue->head->buf_size; i++)
                    fprintf(pinfo->fp, "%c", My_Queue->head->buffer[i]);
                fprintf(pinfo->fp, "\n");

                if (MODE == CHECKPOINT)
                {
                    if (pinfo->cp_count == CP_COUNT)
                        check_point(pinfo);
                    else
                        pinfo->cp_count++;
                }
            }
            free(My_Queue->head->buffer);
            temp = My_Queue->head->next;
            free(My_Queue->head);
            My_Queue->head = temp;
            if (My_Queue->head != NULL)
                My_Queue->head->prev = NULL;
            THREADmonitorexit(pinfo->q_mon);
            return(buffer);
        }
    }
  }
}
```

```
/***************************
 * Receive a reply routine *
 ***************************/
unsigned char *receive_rep(pinfo, message_id, time_out)
Pinfo *pinfo;
int *message_id;
int time_out;
{
    unsigned char *buffer;
    struct timeval tv;
    static struct timeval wf = {32767, 0};
    MBUF *temp;
    THREAD_MONITOR_BLOCK rqmb;
    char file_name[MAX_NAME_SIZE + 9];
    MBUF *curr;
    char exp_name[MAX_NAME_SIZE];
    char exp_host[HOST_SIZE];
    int exp_mid, exp_len, i;
    static int is_open = 0;
    static int count = 0;
    QUEUE *My_Rep_Q;

/*
 * If this is a replay one just get the message from the file
 */
    if (MODE == REPLAY_ONE)
    {
        if (fscanf(pinfo->fp,
                "reply from %s on host %s to message %d len %d - ",
                exp_name, exp_host, &exp_mid, &exp_len) == -1)
            while(1) /* blocking receive never returned */
                select(0, NULL, NULL, NULL, &wf);

/*
 * If a blank message was written return NULL
 * else return the message
 */
        if (exp_name[0] == '-')
            return(NULL);

        if ((buffer = (unsigned char *)malloc(exp_len)) == NULL)
        {
            printf("Memory allocation error\n");
            exit(1);
        }
        for (i = 0; i < exp_len; i++)
            fscanf(pinfo->fp, "%c", &buffer[i]);
        fscanf(pinfo->fp, "\n");
        *message_id = 0;
        return(buffer);
    }
/*
 * NOT REPLAY_ONE
 * Set process information into local variables
 */
    My_Rep_Q = &pinfo->My_Rep_Q;
    tv.tv_sec = 0;
    tv.tv_usec = 100;

/*
 * Normal replay
 */
    if (MODE == REPLAY)
    {
/*
 * Read reply message from file
 */
        fscanf(pinfo->fp, "reply from %s on host %s to message %d\n",
               exp_name, exp_host, &exp_mid);

/*
 * If a blank message was written return NULL
 */
        if (exp_name[0] == '-')
            return(NULL);

/*
 * Check if message is in queue
 */
        while (1)
        {
            THREADmonitorentry(pinfo->rq_mon, &rqmb);
            if (My_Rep_Q->head == NULL)
            {
                THREADmonitorexit(pinfo->rq_mon);
                select(0, NULL, NULL, NULL, &tv);
            }
            else
            {
/*
 * While not at the end of the queue and not message found keep looking
 */
                curr = My_Rep_Q->head;
                while (curr != NULL)
                {
                    if ((strcmp(exp_name, pinfo->Connection[curr->index].name) == 0) &&
                        (exp_mid == curr->time_stamp) &&
                        (strcmp(exp_host, pinfo->Connection[curr->index].host) == 0))
                    {
                        if ((buffer = (unsigned char *)malloc(curr->buf_size)) == NULL)
                        {
                            printf("Memory allocation error\n");
                            exit(1);
                        }
                        *message_id = curr->time_stamp;
                        (void)bcopy(curr->buffer, buffer, curr->buf_size);
                        free(curr->buffer);
/*
 * Fix up the pointers in the reply queue
 */
                        if (curr->prev == NULL) /* message is at head */
                        {
                            if (curr->next == NULL) /* and tail too */
                            {
                                My_Rep_Q->head = NULL;
                                My_Rep_Q->tail = NULL;
                            }
                            else /* it's just the head */
                            {
                                My_Rep_Q->head = curr->next;
                                My_Rep_Q->head->prev = NULL;
                            }
                        }
                        else if (curr->next == NULL) /* it's just the tail */
                        {
                            My_Rep_Q->tail = curr->prev;
                            My_Rep_Q->tail->next = NULL;
```

```
                        }
                        else /* it's in the middle */
                        {
                            curr->next->prev = curr->prev;
                            curr->prev->next = curr->next;
                        }
                        free(curr);
                        THREADmonitorexit(pinfo->rq_mon);
                        return(buffer);
                    }
                    else
                      curr = curr->next;
                }
                THREADmonitorexit(pinfo->rq_mon);
                select(0, NULL, NULL, NULL, &tv);
            }
        }
    }
/*
 * Not a replay mode
 */
    else
    {
        while (1)
        {
            THREADmonitorentry(pinfo->rq_mon, &rqmb);
            if (My_Rep_Q->head == NULL) /* queue is empty */
            {
                THREADmonitorexit(pinfo->rq_mon);
                select(0, NULL, NULL, NULL, &tv);
/*
 * if time_out expires return, if it's capture write a blank message to the
 * file before returning
 */
                if (count == time_out)
                {
                    if (MODE == CAPTURE)
                        fprintf(pinfo->fp,"reply from - on host - to message 0\n");
                    else if ((MODE == FULL_CAPTURE) || (MODE == CHECKPOINT))
                        fprintf(pinfo->fp,
                                "reply from - on host - to message 0 len 0 - ");
                    return(NULL);
                }
                count++;
            }
            else /* got a message */
            {
                if ((buffer =
                     (unsigned char *)malloc(My_Rep_Q->head->buf_size)) == NULL)
                {
                    printf("Memory allocation error\n");
                    exit(1);
                }
                *message_id = My_Rep_Q->head->time_stamp;
                (void)bcopy(My_Rep_Q->head->buffer, buffer,My_Rep_Q->head->buf_size);
/*
 * If we are in a cpature state log the message info
 */
                if (MODE == CAPTURE)
                {
                    fprintf(pinfo->fp,"reply from %s on host %s to message %d\n",
                            pinfo->Connection[My_Rep_Q->head->index].name,
                            pinfo->Connection[My_Rep_Q->head->index].host,
                            My_Rep_Q->head->time_stamp);
                }
                else if ((MODE == FULL_CAPTURE) ||
                         (MODE == CHECKPOINT))
                {
                    fprintf(pinfo->fp,
                            "reply from %s on host %s to message %d len %d - ",
                            pinfo->Connection[My_Rep_Q->head->index].name,
                            pinfo->Connection[My_Rep_Q->head->index].host,
                            My_Rep_Q->head->time_stamp,
                            My_Rep_Q->head->buf_size);

                    /* print the message now */
                    for (i = 0; i < My_Rep_Q->head->buf_size; i++)
                        fprintf(pinfo->fp, "%c", My_Rep_Q->head->buffer[i]);
                    fprintf(pinfo->fp, "\n");

                    if (MODE == CHECKPOINT)
                    {
                        if (pinfo->cp_count == CP_COUNT)
                            check_point(pinfo);
                        else
                            pinfo->cp_count++;
                    }
                }
                free(My_Rep_Q->head->buffer);
                temp = My_Rep_Q->head->next;
                free(My_Rep_Q->head);
                My_Rep_Q->head = temp;
                if (My_Rep_Q->head != NULL)
                    My_Rep_Q->head->prev = NULL;
                THREADmonitorexit(pinfo->rq_mon);
                return(buffer);
            }
        }
    }
}
```

```
/******************************************************************************
 * BOTH  SIDES                                                                *
 ******************************************************************************/
/*************************
 * Initialization Routine *
 *************************/

Pinfo *Init(whoami)
char *whoami;
{
    int sock;
    struct sockaddr_in name;
    WFA *wfa;
    char file_name[MAX_NAME_SIZE + 7];
    char cfile_name[MAX_NAME_SIZE + 9];
    FILE *fopen(), *fp;
    int size = sizeof(name);
    int len;
    char host_name[HOST_SIZE];
    Pinfo *pinfo;
/*
 * Make sure whoami is less than MAX_NAME_SIZE characters
 */
    if (strlen(whoami) > (MAX_NAME_SIZE - 1))
    {
        printf("** WARNING name %s truncated to %.15s **\n", whoami, whoami);
        bcopy(whoami, whoami, MAX_NAME_SIZE);
        whoami[MAX_NAME_SIZE - 1] = '\0';
    }

/*
 * Get our host name
 */
    if (gethostname(host_name, HOST_SIZE) == -1)
    {
        perror("gethostname");
        exit(1);
    }

/*
 * Allocate a structure to hold local process info
 */
    if ((pinfo = (Pinfo *)malloc(sizeof(Pinfo))) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }

/*
 * Initialize queues
 */
   pinfo->My_Queue.head = NULL;
   pinfo->My_Queue.tail = NULL;
   pinfo->My_Rep_Q.head = NULL;
   pinfo->My_Rep_Q.tail = NULL;

/*
 * Set the global for our name and host name
 */
   strcpy(pinfo->My_Name, whoami);
   strcpy(pinfo->My_Host_Name, host_name);

   /** Hack since THREADmurder doesn't seem to work **/
```

```
    pinfo->halt = FALSE;
    /** End hack **/

/*
 * Set the mode of operation
 */
    pinfo->mode = MODE;

/*
 * If capture is turned on open the capture file for writing
 */
    if ((MODE == CAPTURE) || (MODE == FULL_CAPTURE) || (MODE == CHECKPOINT))
    {
        sprintf(cfile_name, "%s_rec", whoami);
        if ((pinfo->fp = fopen(cfile_name, "w")) == NULL)
        {
            printf("Error opening file %s\n", cfile_name);
            exit(1);
        }
    }

/*
 * If replay is turned on open the capture file for reading
 */
    else if ((MODE == REPLAY) || (MODE == REPLAY_ONE))
    {
        sprintf(cfile_name, "%s_rec", whoami);
        if ((pinfo->fp = fopen(cfile_name, "r")) == NULL)
        {
            printf("Error opening file %s\n", cfile_name);
            exit(1);
        }
    }

/*
 * Get a socket and a name
 */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("sock");
        exit(1);
    }
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = htonl(INADDR_ANY);
    name.sin_port = htons(0);
/*
 * Bind the socket
 */
    if (bind(sock, &name, sizeof(name)) == -1)
    {
        perror("bind");
        exit(1);
    }
/*
 * Get port chosen
 */
    len = sizeof(name);
    if (getsockname(sock, &name, &len) == -1)
    {
        perror("getsockname");
        exit(1);
    }
/*
 * Initialize the monitor that send will use and the queue monitor
 */
    ghn = THREADmonitorinit(0, NULL);
    pinfo->q_mon = THREADmonitorinit(0, NULL);
    pinfo->rq_mon = THREADmonitorinit(0, NULL);

/*
 * Zero the interesting read file descriptor set and mark our interest
 */
    FD_ZERO(&pinfo->RFDS);
    FD_SET(sock, &pinfo->RFDS);

/*
 * The initial number of connections is zero
 */
    pinfo->Num_Cons = 0;

/*
 * Start listening for connection attempts
 */
    if (listen(sock, 5) < 0)
    {
        perror("listen");
        exit(1);
    }

/*
 * Create a receive thread
 * Allocate a structure to hold arguments for the thread
 */
    if ((wfa = (WFA *)malloc(sizeof(WFA))) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    wfa->sock = sock;
    wfa->name = &name;
    wfa->pinfo = pinfo;
    pinfo->My_Rec_Thread = THREADcreate(receive_thread, wfa, 0, 1, 16384, 8);
/*
 * Register yourself to be talked to
 */
    sprintf(file_name, ".%s", whoami);
    if ((fp = fopen((char *)file_name, "w")) == NULL)
    {
        printf("Error writing routine %s into database.\n", whoami);
        return((Pinfo *)-1);
    }
    else
    {
        fprintf(fp, "%s\n%d\n", host_name, ntohs(name.sin_port));
        fflush(fp);
        fclose(fp);
    }
    return(pinfo);
}
```

```
/*********************
 * Shut down Routine *
 *********************/
void Finish(pinfo)
Pinfo *pinfo;
{
    char tmp[10 + MAX_NAME_SIZE];

    if (pinfo->halt == FALSE)
    {
        THREADmurder(pinfo->My_Rec_Thread);
    /** Hack since THREADmurder doesn't seem to work **/
        pinfo->halt = TRUE;
    /** End hack **/
        sprintf(tmp, "rm .%s", pinfo->My_Name);
        system(tmp);
    }
    if (MODE != NORMAL)
        fclose(pinfo->fp);
}

/****************
 * Call Routine *
 ****************/
unsigned char *MsgCall(pinfo, id, message, mlen, time_out)
Pinfo *pinfo;
int id;
unsigned char *message;  /* message to send */
int mlen;                /* message length  */
int time_out;
{
    unsigned char *temp;
    int message_id, ret_mid;

    message_id = send_message(pinfo, id, message, mlen, EXPECT_REP, -1);
    if (message_id == -1)  /* error */
        return((unsigned char *)-1);
/*
 * Block waiting for reply to this message
 */
    temp = receive_rep(pinfo, &ret_mid, time_out);
    while ((ret_mid != message_id) && (temp != NULL))
    {
        printf("** WARNING - received reply with wrong message id, waiting... **\n");
        free(temp);
        temp = receive_rep(pinfo, &ret_mid, time_out);
    }
    return(temp);
}
```

```
/****************************************************************************
 * Shared memory routines                                                   *
 ****************************************************************************/
/**********************************************
 * Initialize object for shared memory access *
 **********************************************/
/**
 ** May want to add a size here so that we know how much data to
 ** write when an object is saved in FULL_CAPTURE mode
 **/
Object *ObjectInit(data, name)
char *data, *name;
{
    Object *obj;

    if ((obj = (Object *)malloc(sizeof(Object))) == NULL)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    obj->data = data;
    obj->mon_write = THREADmonitorinit(0, NULL);
    obj->mon = THREADmonitorinit(0, NULL);
    obj->active_read = 0;
    obj->total_read = 0;
    obj->version = 1;
    (void)strcpy(obj->name, name);
    return(obj);
}
```

```
/***********************
 * Get read permission *
 ***********************/
ReadEntry(pinfo, object)
Pinfo *pinfo;
Object *object;
{
    THREAD_MONITOR_BLOCK mon_blk, wmon_blk;
    char name[OBJ_NAME_SIZE];
    int exp_version;
    struct timeval tv;

/**
 ** If mode is replay one we need to fill in the object from the file
 ** This requires some knowledge of the object size and structure
 ** Currently we only handle integers
 **/
    if (MODE == REPLAY_ONE)
    {
        fscanf(pinfo->fp, "read object %s version %d contents - ", name, &exp_version);
        fscanf(pinfo->fp, "%d\n", object->data);
        return;
    }
    else if (MODE == REPLAY)
    {
/*
 * Read the version from the tape
 * Wait until that version is active
 */
        tv.tv_sec = 0;
        tv.tv_usec = 1000;
        fscanf(pinfo->fp, "read object %s version %d\n", name, &exp_version);
        while (object->version != exp_version)
            select(0, NULL, NULL, NULL, &tv);
    }
    else
    {
        /* make sure there is no write going on */
        THREADmonitorentry(object->mon_write, &wmon_blk);
        /* make sure we are the only ones incrementing */
        THREADmonitorentry(object->mon, &mon_blk);
        object->active_read++;
        /** if (object->active_read > 1) printf("**concurrent read**\n"); **/
        THREADmonitorexit(object->mon);
        THREADmonitorexit(object->mon_write);
    }
/*
 * If the mode is capture log the read contents to the file
 */
    if (MODE == CAPTURE)
        fprintf(pinfo->fp, "read object %s version %d\n", object->name,
                object->version);
    else if ((MODE == FULL_CAPTURE) || (MODE == CHECKPOINT))
    {
        fprintf(pinfo->fp, "read object %s version %d contents - ", object->name,
                object->version);
/**
 ** Log the contents - again this requires object size information
 ** Currently we only integers
 **/
        fprintf(pinfo->fp, "%d\n", *(int *)object->data);

        if (MODE == CHECKPOINT)
            if (pinfo->cp_count == CP_COUNT) /* checkpoint if it's time */
                check_point(pinfo);
            else
                pinfo->cp_count++;
    }
}

/***********************
 * Finish read control *
 ***********************/
ReadExit(pinfo, object)
Pinfo *pinfo;
Object *object;
{
    THREAD_MONITOR_BLOCK mon_blk;

    if (MODE == REPLAY_ONE) /* no-op here */
        return;

    /* make sure we are the only ones incrementing and decrementing */
    THREADmonitorentry(object->mon, &mon_blk);
    object->total_read++;
    object->active_read--;
    THREADmonitorexit(object->mon);
}
```

```
/************************
 * Get write permission *
 ************************/
WriteEntry(pinfo, object)
Pinfo *pinfo;
Object *object;
{
    THREAD_MONITOR_BLOCK mon_blk, wmon_blk;
    static struct timeval tv = {0, 1000};
    char *exp_name[OBJ_NAME_SIZE];
    int exp_version, exp_tr;


    if (MODE == REPLAY_ONE) /* no-op */
        return;
    else if (MODE == REPLAY)
    {
        fscanf(pinfo->fp, "wrote object %s version %d total readers %d\n",
               exp_name, &exp_version, &exp_tr);

        /* wait for expected version to become available */
        while (object->version != exp_version)
            select(0, NULL, NULL, NULL, &tv);

        while (object->total_read < exp_tr)
            select(0, NULL, NULL, NULL, &tv);
    }
    else
    {
        /* gain exclusive access */
        THREADmonitorentry(object->mon_write, &wmon_blk);

        /* let active readers finish */
        while (object->active_read)
            select(0, NULL, NULL, NULL, &tv);
    }
    /* write object name, object version to file */
    if (MODE == CAPTURE)
        fprintf(pinfo->fp, "wrote object %s version %d total readers %d\n",
                object->name, object->version, object->total_read);
}
```

```
/************************
 * Finish write control *
 ************************/
WriteExit(pinfo, object)
Pinfo *pinfo;
Object *object;
{
    THREAD_MONITOR_BLOCK mon_blk;

    if (MODE == REPLAY_ONE)
        return;
    else if (MODE == REPLAY)
    {
        object->total_read = 0;
        THREADmonitorentry(object->mon, &mon_blk);
        object->version++;
        THREADmonitorexit(object->mon);
    }
    else
    {
        object->total_read = 0;
        object->version++;
        THREADmonitorexit(object->mon_write);
    }
}
```

```
/******************************
 * Check point signal handler *
 ******************************/
int handler()
{
  HAND = FALSE; /* indicate that we were called */
  return(0);
}

/***********************
 * Check point routine *
 ***********************/
check_point(pinfo)
Pinfo *pinfo;
{
    int pid;
    char tmp_file[MAX_NAME_SIZE + 5];

/*
 * Get rid of last checkpoint
 */
    if (pinfo->pid != 0)
        kill(pinfo->pid, 9);
    fclose(pinfo->fp);
/**
 ** Probably best to prevent receive thread from doing
 ** any i/o during fork
 **/

/*
 * Make a new checkpoint
 */
    pinfo->pid = fork();
/*
 * If we are now the child...
 */
    if (pinfo->pid == 0)
    {
/*
 * Prepare for replay
 */
/**
 ** This is to keep forking the child so that we can
 ** keep replaying
        while (pinfo->pid == 0)
        {
**/
/*
 * Get rid of the childs receive thread
 */
            /** THREADmurder hack **/
            pinfo->halt = TRUE;
            MODE = REPLAY_ONE; /* set child to replay mode */
            signal(SIGUSR1, handler); /* wake up call */

/*
 * Wait until someone tells us to replay
 */
            while (HAND)
                pause();   /* continue when we get the user signal */
/**
 ** To allow multiple reexecutions keep forking a child
 ** Child goes into wait parent does replay
 **/
/**
            pinfo->pid = fork();
        }
**/
/*
 * Make sure file pointer is in correct place and read out pid
 */
        sprintf(tmp_file, "%s_rec", pinfo->My_Name);
        pinfo->fp = fopen(tmp_file, "r");
        fscanf(pinfo->fp, "pid = %d\n", &pid);
        printf("replay starts now for pid %d - halt %d\n", pid, pinfo->halt);
    }
    else
    {
/*
 * Else we are the parent.  Prepare to start capture again
 */
        sprintf(tmp_file, "%s_rec", pinfo->My_Name);
        pinfo->fp = fopen(tmp_file, "w");
        fprintf(pinfo->fp, "pid = %d\n", pinfo->pid);
        MODE = CHECKPOINT;
        printf("capture starts now\n");
    }
    pinfo->cp_count = 0;
}
```