

Performing Replay in an OSF DCE Environment

Yuh Ming Yong
David J. Taylor

Abstract

Debugging a distributed application is inherently difficult because such factors as network delay and varying system loads may cause the behaviour of the application to change from one execution to another. Using a standard debugger with such an application is also likely to perturb its behaviour sufficiently that some bugs will not be manifested when the debugger is used. Fortunately, a replay mechanism can be used to circumvent these problems. Replaying a program involves a monitoring phase, during which the behaviour of the program is logged, and a replay phase, during which the behaviour of the program is coerced to follow the partial order logged during the monitoring phase.

In this paper,¹ we describe an implementation of the replay technique for use with the Open Software Foundation's Distributed Computing Environment (OSF DCE). OSF DCE presents a special problem in that servers are multithreaded and incoming RPCs are assigned dynamically to threads. To preserve the event partial order, it is necessary to ensure that an RPC uses the same thread on replay. This difficulty is dealt with by ensuring that the DCE thread service sees the same resource state during replay and hence makes the same thread-allocation decision.

¹ The IBM contact for this paper is Patrick Finnigan, Client Server Enabling Tools, Application Development Technology Centre, IBM Canada Ltd., Mail Stop 3P, 1150 Eglinton Avenue East, North York, Ontario M3C 1W3.

1 Introduction

There is a fundamental difference between debugging sequential and distributed programs. A sequential program, with a single thread of control, usually has well-defined behaviour, depending only on its inputs. There is also a well-defined notion of physical time for each instance of program execution. Repeated execution of such a program will, in most cases, produce identical behaviour. On the contrary, repeated executions of a distributed application may easily produce different behaviour. Such differences in behaviour have several possible causes. A distributed program may have multiple processes or threads running on different (or possibly the same) machines. Because loads on machines vary over time, the relative speed of execution for processes or threads in the distributed application will also vary. Communication among the distributed processes will encounter varying network delays, which may also change the behaviour of the distributed program. As well, the application may be treated differently by the operating-system scheduler on different executions, again causing differences in interaction between processes or threads in the application.

Our work in debugging distributed applications is based primarily on the concept of a partial-order relation between the events that occur. That is, event A may precede event B, event B may precede event A, or events A and B may be concurrent. Events that are concurrent do not necessarily take place at the same instant of real time, but one event cannot affect the other. Timing perturbations that simply

shift the relative real times of concurrent events thus cannot change the behaviour of the application. Hence, the preceding problems may be summarised as “various factors may change the partial order of events when a distributed application is executed repeatedly.”

To obtain consistent behaviour on repeated executions, a replay mechanism must be used. In general, replay consists of two stages. During the monitoring phase, a replay log, which records activities of the distributed program, is created. Then during the replay phase, the replay log is used to coerce event occurrences to follow the same partial order as the initial execution of the distributed program.

Replay has been described previously by other authors for other environments [2, 3]. This paper describes an implementation of replay for use with the Open Software Foundation’s Distributed Computing Environment (OSF DCE) [4], which presents a special problem because servers are multithreaded and incoming RPCs are assigned dynamically to threads. To preserve the event partial order, it is necessary to ensure that an RPC uses the same thread on replay. This difficulty, which is not normally handled by a replay mechanism, is dealt with by ensuring that the DCE thread service sees the same resource state during replay and hence makes the same thread-allocation decision. The replay mechanism has been integrated with a tool, POET, for visualising execution histories of distributed programs and with a sequential debugger to create an effective overall debugging environment.

The remainder of the paper is organised as follows. Section 2 provides an overview of the replay facilitator, the process responsible for controlling replay. Section 3 describes the replay-interval structure, which is used to manage replay information. Section 4 presents the technique used to control thread use during replay. Section 5 describes the various components of the replay facilitator and the method used for performing replay in the OSF DCE environment. Section 6 presents a better algorithm for constructing the replay-interval structure, which improves replay efficiency and requires less space. Finally, Section 7 summarises the paper and suggests possibilities for further research.

2 Overview of the Replay Facilitator

This paper is based on work done to add debugging functions to POET. POET can best be described as a tool for instrumenting a distributed application to collect and display event traces showing the execution of its component processes or threads [5, 6, 7]. Because POET can be used with a variety of application environments, the monitored entities with sequential behaviour are referred to as *traces* and may represent processes, threads, monitors, semaphores, etc., depending on the application environment. In OSF DCE, each trace represents a thread in a client or server process. Thus, in this paper, the terms “thread” and “trace” are nearly interchangeable, but the former will generally be used when referring to the DCE application and the latter when referring to the event data managed by POET.

In DCE the events of interest are primarily RPCs. These events are captured by code in the client and server stubs. We have endeavoured to capture events with as little overhead as possible, in order to minimise the probe effect on the application. It is, of course, impossible to eliminate the probe effect entirely when using software-based instrumentation. We originally modified the stubs by post-processing the stub code produced by the IDL compiler. More recently, since the source for the IDL compiler was placed in the public domain, we have modified the IDL compiler itself to add instrumentation to the stub code. The control required for replay is also obtained by modification of the stub code.

The extended architecture of POET for replay is shown in Figure 1, in which boxes represent processes such as the disk server, replay facilitator, checkpoint process, and debug-session process, directed lines represent the flow of data or information such as event data fetched from the disk server and replay-control information sent to the application, and data stores represent storage facilities for the disk server and checkpoint processes. The disk-server data stores contain the program-execution history. The primary disk server maintains the execution history for the initial execution and the secondary disk server maintains the exe-

cution history observed during replay. To initiate replay, the replay facilitator requests event data from the primary disk server and uses that data to create the replay-control structure, which the replay facilitator uses to manage replay. Based on the recorded partial order of events, the replay facilitator then controls the replay sequence by sending control information to the replay-agent embedded in the application, which acts as a “liaison” between the replay facilitator and the application. The replay-agent is responsible for delaying execution of events until appropriate authorisation is received, managing thread use and release by the application, and handling the communication between the replay facilitator and the application. The control information sent by the replay facilitator informs the application when it can proceed with the execution of a specific event. As the application is replayed, its run-time events are sent to the secondary disk server, which makes the run-time event data available to the replay facilitator. The replay facilitator uses the run-time event information to monitor the progress of the replay execution and to determine the next replay interval. The replay-facilitator, secondary disk server, checkpoint, and debug-session processes collect and manage run-time events as an application is replayed, whereas the primary group of processes provides replay information, which was previously captured during the monitoring phase, to the replay facilitator.

At present, all of the processes making up POET itself must run on a single computer system, but the target (application) processes may run on multiple systems, distinct from each other and from the system running POET. The systems involved also need not all be of the same architecture. DCE, of course, provides data-representation conversion between the component processes of the application. POET also provides data-representation conversion between the disk server and the target processes.

Another view of the replay facilitator is shown in Figure 2, which provides a macroscopic view of the various components that make up the replay facilitator.

3 Replay-Interval Control

The present implementation of the replay facilitator controls execution by using a set of “replay intervals” representing blocks of events that may all be executed once an initial condition has been met. Each replay interval consists of a sequence of RPC events involving one pair of traces together with any intervening unary events. The replay-interval structure maintains the replay points for each trace. Each entry in the replay-interval structure stores the number of the event ending an interval. The replay facilitator constructs the replay-interval entries for all traces recorded in the event-history log, with each replay interval corresponding to a pair of traces. During replay, the replay facilitator checks whether both the current and partner traces are ready to replay the same interval. If so, both traces are executed up to their corresponding events in their replay intervals. Figure 3 shows an example of a replay-interval structure, corresponding to the program-execution history shown in Figure 4

In Figure 3, a replay interval is indicated by a pair of boxes joined by a dotted line and the number in each box indicates the event number at the end of the replay interval for that trace. Thus, the boxes labelled “2” on the first two lists indicate that there is a replay interval extending to event 2 in each of these traces. Since it is the first entry for each trace, the interval contains events 0 through 2 in each trace. Thus, this interval can be replayed immediately. The first box on the third list, also labelled “2” represents events 0 through 2 in its trace, but replay for that trace cannot begin immediately. The problem is that events 3 and 4 of the first trace are also in the replay interval and event 3 is not initially ready to be executed. However, once the first replay interval is completed, the first trace becomes ready to execute event 3, and then this interval can be replayed.

4 Thread Management During Replay

Since DCE threads are automatically allocated by the server process, there is no guarantee that

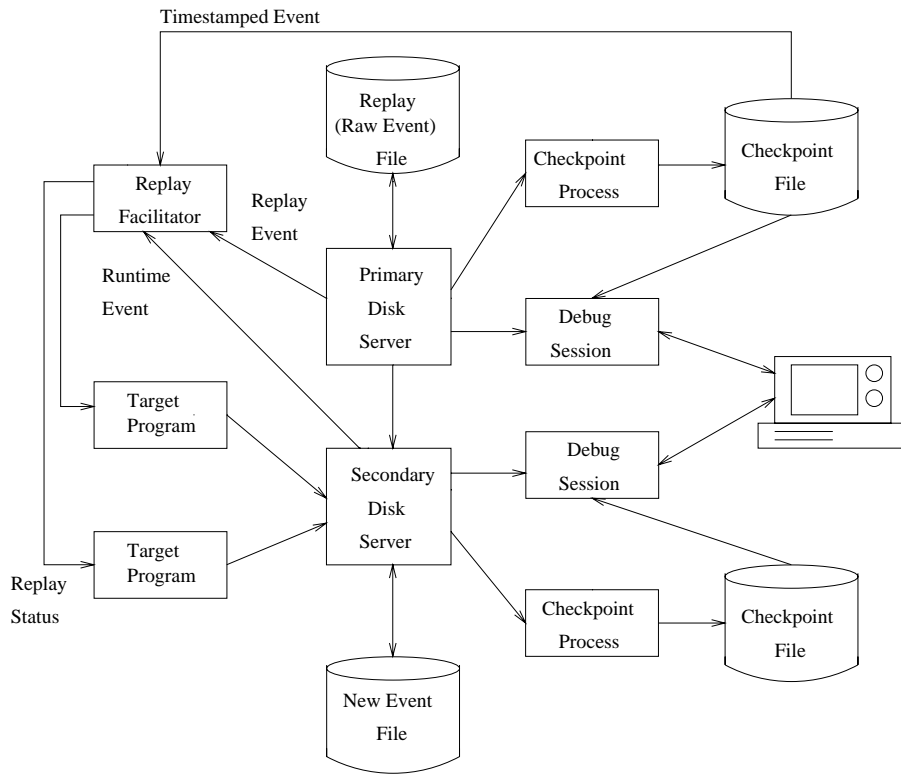


Figure 1: POET architecture extended for replay.

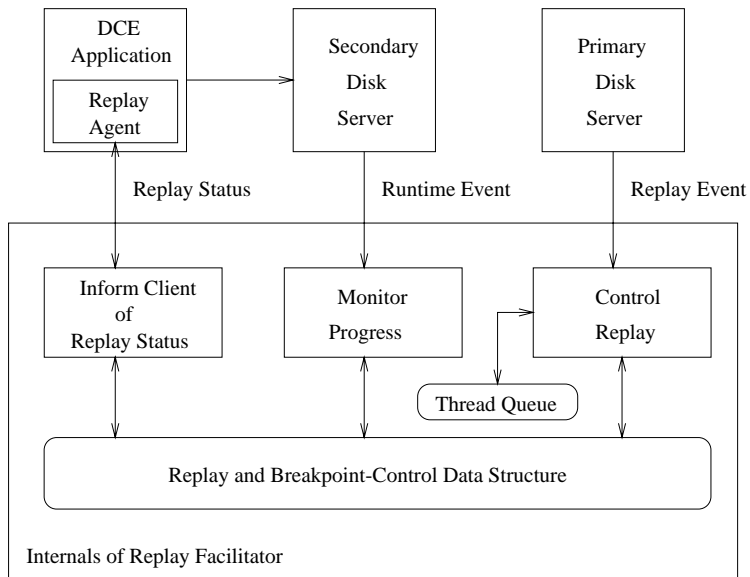


Figure 2: Components of the replay facilitator.

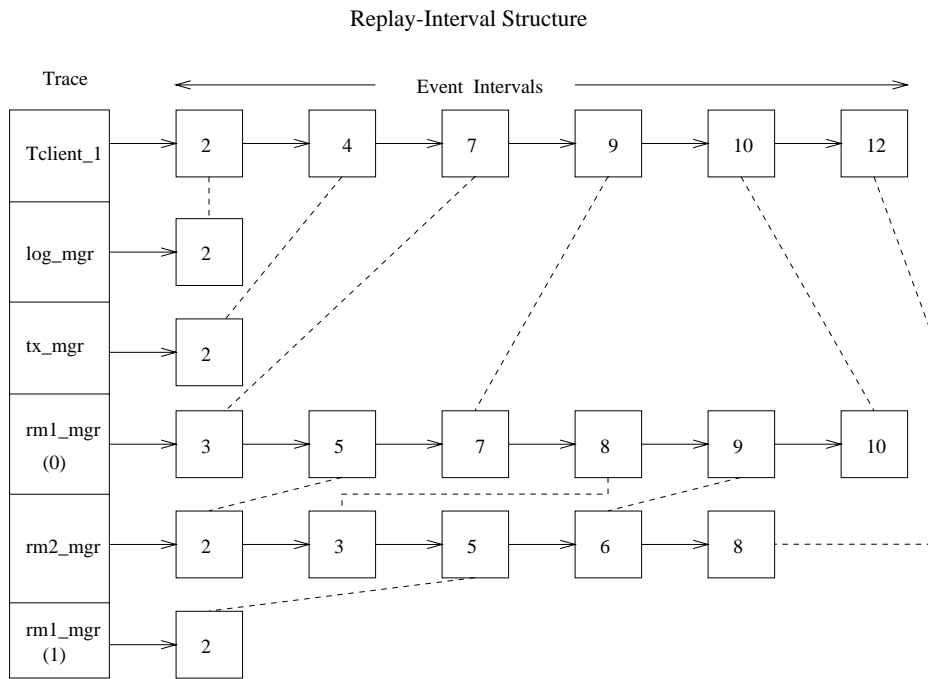


Figure 3: Replay-interval control structure.

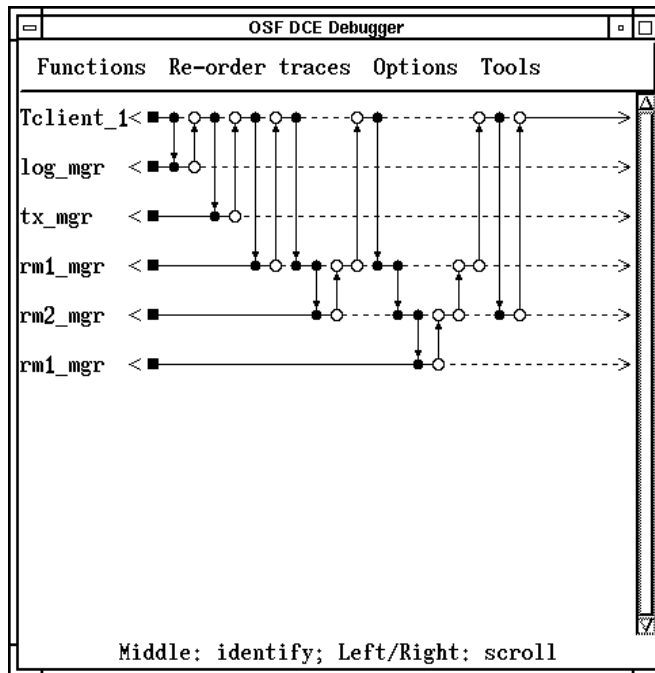


Figure 4: DCE events captured by POET during the monitoring phase.

the same thread will be assigned to an RPC during replay. Hence, it is essential that the replay facilitator control not only the partial-order relation of the events but also the assignment of threads to RPCs. The replay facilitator must ensure that the events are replayed in a controlled manner so that the DCE call-thread service assigns the same threads to the events. Client events do not pose any problem for thread allocation because their threads are user-generated and the replay facilitator has control over the events. To handle server-thread assignment the replay facilitator maintains status information for each thread. At any point in time, there can exist at most two incomplete RPCs on a server thread; an incoming RPC from a client to that server thread and an outgoing RPC from that thread to another server. This property allows the replay facilitator to simply store the identity of two partner traces for each thread: the call-by and call-to partner-trace identities. The call-by partner trace indicates which client trace has made a request to this server thread. If the server thread has made a request-for-service to another server process (to complete the pending client request), then the identity of the other server is recorded as the call-to partner trace.

At the start of a replay interval, the replay facilitator must set the call-to and call-by trace identities appropriately for the threads involved and then reset them appropriately at the end of the interval. The start of a replay interval need not represent the beginning of an RPC and the end of a replay interval need not represent the end of an RPC. (Consider, for example, the replay interval involving the ninth and tenth events of Tclient_1 in Figure 4. That interval represents the last event pair of one RPC followed by the first event pair of another RPC between the same two threads.) Thus, it is important to determine whether a new RPC is being initiated or an old one is being completed, and so on, to determine when the trace identities attached to a thread need to be set or reset.

In addition to verifying that both threads are ready to enter a given replay interval, the replay facilitator must verify that a thread to be used is not currently assigned to another active replay interval. Since the

DCE thread-assignment algorithm assigns the lowest-numbered available thread, it is also necessary to verify that all lower-numbered threads are currently assigned. Thus, the replay facilitator must also control events that will release an assigned thread, so that other events originally executed on higher-numbered threads can be replayed properly.

5 Control of Replay

There are three essential routines which work collectively to provide the replay functionality, each performing one specific task. For purpose of explanation, these three routines will be called send-status, monitor-progress, and update-control. The update-control routine is responsible for controlling replay and managing the replay-control structure. The send-status routine sends replay-control messages to the application. The monitor-progress routine monitors the progress of the replay. The three routines are called from the main event loop of the replay facilitator and are thus periodically invoked to perform their functions. At each invocation, the routines check if there are pending tasks to perform.

The replay facilitator maintains a separate linked list, called proceed-replay, of threads that need to be informed of their new replay status. The proceed-replay list is created to alleviate the effort of verifying the replay status of a thread. Consequently, the send-status routine need only use the proceed-replay list to accomplish its tasks (rather than using the replay-control structure to verify the replay status of every trace at each invocation). Each entry in the proceed-replay list contains two pointer fields that reference traces in the replay-control structure. After both the trace and its partner trace have been successfully informed of their new replay status, they are removed from the proceed-replay list. The send-status routine returns to the event loop after all items in the proceed-replay list have been considered.

As events are replayed, they are collected by the secondary disk server. The monitor-progress routine receives run-time event information from the secondary disk server to mon-

itor the progress of the replay. Table 1 lists the fields in each event-data packet received from the secondary disk server. The *ctrl_type* field indicates the purpose of the information contained in the packet. Valid control types are (1) run-time event data which is used for monitoring the replay, (2) request to adjust replay movie speed, (3) request to disable the replay facilitator to run the application in a non-replay environment, and (4) request to clean up and shut down the replay facilitator. All control types except the first require no update to the replay-control structure. The *pid* and *trace_id* fields are, respectively, the process and thread numbers for this trace. These two fields are used for mapping the run-time event information to the replay-control structure. Before the replay facilitator can handle the run-time event information, it must know how to map the run-time event information to the corresponding parts of the replay-control structure. This is because the identification information obtained during the monitoring phase and during replay may be different. When an application process connects to the replay facilitator, the process uses a unique trace number (as recorded in the replay log by POET) for identification. This unique identifier is placed in an environment variable when the replay facilitator starts the process. The replay facilitator uses this identifier to identify the process and thus to locate the appropriate replay-control structure. It can then store the replay-time process, thread, and (network) stream identifiers for the process in the appropriate locations. Once the replay-time identifiers have been obtained, the replay facilitator is able to map information in the replay log to the run-time events. It is possible for the replay facilitator to receive run-time event data (from the secondary disk server) before getting the “registration” information from the process. Hence, the replay facilitator maintains a buffer for handling such unusual situations. When a process connects to the replay facilitator, it checks for any unresolved run-time event information that needs to be mapped and placed in the replay-control structures for the threads of that process. Once the replay-control structures for that process have been updated with the identification informa-

tion, the replay facilitator is able to process the corresponding run-time event information received from the secondary disk server. The last field in Table 1 is the trace’s event count, *e_evcnt*, indicating that the event with that sequence number (in the trace) has occurred.

<i>Field</i>	Data Type	<i>Description</i>
ctrl_type	Integer	Type of control
trace_id	Integer	Thread number
pid	Integer	Process number
e_evcnt	Integer	Event count

Table 1: Run-time event information from the secondary disk server.

The update-control routine uses event information from the primary and secondary disk servers to control replay. The primary disk server provides event information that the replay facilitator uses to build the replay intervals. In the DCE environment, program behaviour is represented by (synchronous) RPCs. A normal RPC consists of two pairs of events: sending and replying. A call-with-no-reply RPC consists of only one pair of events. To replay an RPC, the replay facilitator must ensure that both the sender and receiver events are ready. When the replay facilitator receives event data from the secondary disk server, it determines if the partner event is also ready for replay. If it is, the replay facilitator determines the replay interval for that pair of events. Each trace maintains a (*can_proceed*) flag to indicate whether the trace is ready for replay. When a trace and its partner trace are both ready to replay, their flags are set to true, which signals the replay facilitator to fetch the corresponding replay interval. The flag is reset when the trace reaches the end of its replay interval. Because the readiness of a trace for replay depends on both that trace and a partner trace, the flag is used to provide a local indication of its replay status. The event data (from the primary disk server) also contains information about the trace’s thread that was previously allocated during the monitoring phase. The replay facilitator must ensure that the same thread is allocated to the trace during

replay by managing the thread pools. A family of traces (that is, traces with the same process number) shares one thread pool. If there is a request to use a thread and the thread slot in the corresponding thread pool is empty, the trace is assigned to the thread. If, however, the requested thread is already occupied, the trace must wait for its turn to be assigned to the thread. Since the DCE call-thread service allocates threads according to the availability of idle threads, the thread-dependency rule ensures that a thread is not assigned unless all other appropriate threads are busy, ensuring that the call-thread service will assign this thread. The send-status routine ensures that only traces with proper thread assignment and consistent with the thread-dependency rule are allowed to proceed replaying. Once a replay interval has been determined and the appropriate thread assignment has been successfully granted, the replay facilitator can then send the new replay status to the threads. Similarly, a thread cannot be released if there are other threads that depend on its being busy. As events are replayed, the run-time event information is relayed back to the replay facilitator. The replay facilitator uses this information to monitor the progress of the replay. When the event at the end of the replay interval is reached and there are more events to replay, the replay facilitator proceeds to another cycle of fetching the next replay interval, updating the replay-control structure, and managing thread assignment.

6 Better Algorithm for Replay-Interval Control

The algorithm described in Section 3 creates the replay-interval structure based on blocks of events involving only two processes. This section presents a better algorithm that allows the replay facilitator to detect events involved in races and then to control the replay based on those events. In this context, a *race* is a situation in which two calls are made to the same server and there is no synchronisation in the application to determine which will occur first. Thus, in some executions one call will be received first and in other executions, the

other call will be received first. The call event that happened first (as viewed by the server) is referred to as the *winner* of the race.

An event that was involved in a race and “lost” the race is delayed until the event that “won” the race has occurred. This replay method allows the replay facilitator to replay larger blocks of events and requires less space to maintain the replay-interval structure. An entry in the replay-interval structure represents a race condition in which two send-events to the same trace are causally independent. During replay, the replay facilitator delays the send-event that “lost” the race during initial execution until the other event is known to have occurred.

The algorithm requires, for each trace, a data structure which contains the timestamp of the last event processed, the timestamp of the last send-event received by the trace, and the list of race-events. After initialising the data structure for each trace, the algorithm invokes the timestamper to process the entire set of events captured during initial execution. As each event is timestamped, it is returned by the timestamper. The nature of the timestamp algorithm guarantees that the events are returned in a sequence consistent with the partial order. As events are received from the timestamper, the algorithm checks the event type and acts on the event accordingly. For send-events, the algorithm creates an entry in the race-list and constructs a timestamp that represents its precedence without the precedence effects resulting from pairing it with the corresponding receive event. For receive-events, the algorithm locates the corresponding send-event and checks for a race condition between this (new) send-event and the last send-event received by the trace. If such a condition exists, the (new) send-event is updated with the identity of the race-event which the send-event needs to wait for. If, however, the (new) send-event is a successor, the entry in its race-list is removed. During replay, the send-event is delayed until the race-event has occurred. The algorithm is shown in Figure 5.

The execution history shown in Figure 4 contains no races. Thus, this algorithm will determine that no events need to be delayed because of races. Using this algorithm, the execution

```

/* Data structure */
per trace:
    timestamp of last event
    timestamp of last send-event into trace
    list of (send) events to be delayed
        each element contains:
            event number
            timestamp
            trace and event number of send event which it needs to wait for
            link field

/* Initialise */
for all traces do
    zero timestamp of last event, timestamp of last send-event into trace
    create empty list
endfor

/* Construct replay interval */
while (receive an event from the timestamp routine) do

    if event type is send then
        add entry in list of the trace
        timestamp = saved timestamp + 1 in local position
    endif

    if event type is receive then
        locate corresponding ("new") send-event
        compare timestamp of new send against saved timestamp of send

        if new send-event is predecessor then
            disaster
        endif

        if new send-event is concurrent then
            put identity of saved send-event into list entry for new send
        endif

        if new send-event is successor then
            remove entry for it from list on other trace
        endif

        store timestamp of new send as timestamp of send into trace
    endif

    store timestamp of event just processed in structure of this trace
endwhile

```

Figure 5: Improved algorithm for generating replay intervals

would still not be replayed as one large block of events, because of the need to ensure proper thread assignment. In principle, a more sophisticated analysis could be performed to determine that delays to control thread assignment will also never be necessary for this execution history.

7 Conclusions and Further Work

This paper has presented a technique for replaying parallel programs based on the partial-order relation of events, coercing the execution to be consistent with the original execution. With the use of the replay facilitator, the non-determinism of the distributed-application behaviour is eliminated. (Of course, this is only true to the extent that RPC interactions are the source of non-determinism. If, for example, a program reads the real-time clock and then makes decisions based on its value, the mechanism described here will not be able to recreate the same behaviour on replay.) In addition, the control of event occurrences allows the user to work effectively despite the lack of a well-defined notion of global time for distributed processes. Two types of (DCE) events are observed and captured during the monitoring phase: synchronous-communication and non-communication events. Synchronous-communication events describe the RPC-type of communication between client and server processes; non-communication events describe other program behaviour such as thread initiation and termination. The replay technique controls the caller events, since in the client/server model of computation the caller events affect how the DCE resources are allocated at the server processes. Unfortunately, the replay facilitator does not have direct control over the server's resources. For example, thread allocation at the server process is automatically managed and controlled by the DCE call-thread service. The replay technique discussed in this paper ensures that the DCE thread service sees the same resource state during replay and hence makes the same thread-allocation decision. The discussion in this paper is necessarily brief; a fuller description can

be found elsewhere [8].

The replay facilitator controls the replay of events by using event information that is captured during the monitoring phase and event information that is collected during replay. To improve replay efficiency, the replay facilitator maintains and manages a replay-interval data structure, which indicates the last event in each replay interval for each trace. By replaying blocks of events, rather than individual events, we have reduced both the communication overhead between the replay facilitator and the application and the overhead in performing replay. We have also presented a better algorithm that allows the replay facilitator to detect race events and controls the replay based on these events. This replay method allows the replay facilitator to replay larger blocks of events that are consistent with the partial-order relation and requires less space to maintain the replay-interval structure.

The replay facilitator also allows a user to set breakpoints during replay, examine the values of variables, and so on. Such pauses during execution of a concurrent program would normally change its behaviour, but during replay, they simply cause all other threads to pause at the point where continuing further would be inconsistent with the recorded partial order. These facilities are made available by providing an interface to the debugger in the AIX Workbench [1].

The implementation of the replay facilitator can be used as a framework for performing research in other areas, such as incremental replay, performance analysis, and integration with a race-detection mechanism. The replay facilitator at present only allows replay of a program from the start to end. As well, the user must ensure that inputs from the external environment are identical during replay. One interesting extension to the replay facilitator would be to perform incremental replay, allowing the user to choose an arbitrary interval to begin replay. A practical extension would provide recording and replay of external inputs to the application. Another interesting extension would be to integrate tools for performing performance analysis on time-critical applications. Another extension would enable the user to examine the effect of race conditions during re-

play by coercing the race-events to occur in a different order, rather than in the same order. This could be particularly useful if a program contains a race condition that almost always has the same “winner”, so that it is very difficult during a normal execution to observe the program behaviour for the other race outcome.

Acknowledgments

The work described in this paper was supported by NSERC, under Research Grant OGP00003078, and by an IBM graduate-student fellowship.

About the authors

Yuh Ming Yong received his M.Math. degree in Computer Science from the University of Waterloo and is currently employed at the IBM Toronto Laboratory in the Networked Applications Development Centre. His research interests include distributed debugging and high-speed networks.

He can be reached at the address IBM Canada Ltd., 22/653, 844 Don Mills Road, North York, Ontario M3C 1V7. His e-mail address is ymyong@vnet.ibm.com.

David J. Taylor is an Associate Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1977. His research interests include distributed-systems software and software fault tolerance. During the 1991-1992 academic year, he spent a sabbatical at the Centre for Advanced Studies, IBM Toronto Laboratory.

He can be reached at the address Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. His e-mail address is dtaylor@uwaterloo.ca.

References

- [1] International Business Machines Corporation. *IBM AIX SDE Workbench/6000 User's Guide and Reference*. SC09-1453.
- [2] T. Leblanc and J. Mellor-Crummey. “Debugging parallel programs with instant replay”. *IEEE Transaction on Computers*, C-36(4):471–481, April 1987.
- [3] R. Netzer and J. Xu. “Adaptive message logging for incremental program replay”. *IEEE Parallel & Distributed Technology*, 1(4):32–40, November 1993.
- [4] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [5] D. Taylor. “A prototype debugger for Hermes”. *Proceedings of the 1992 CAS Conference*, I:29–42, November 1992.
- [6] D. Taylor. “The use of process clustering in distributed-system event displays”. *Proceedings of the 1993 CAS Conference*, pages 505–512, October 1993.
- [7] D. Taylor and M. Coffin. “Integrating real-time and partial-order information in event-data displays”. *Proceedings of the 1994 CAS Conference*, pages 157–165, November 1994.
- [8] Y. Yong. *Replay and Distributed Breakpoints in an OSF DCE Environment*. M.Math Thesis, University of Waterloo, Ontario, Canada, 1995.