Restoring Consistent Global States of Distributed Computations

Arthur P. Goldberg Ajei Gopal* Andy Lowry Rob Strom

Distributed Systems Software Technology Group
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights
New York 10598

Abstract

We present a mechanism for restoring any consistent global state of a distributed computation. This capability can form the basis of support for rollback and replay of computations, an activity we view as essential in a comprehensive environment for debugging distributed programs. Our mechanism records occasional state checkpoints and logs all messages communicated between processes.

Our mechanism offers flexibility in the following ways: *any* consistent global state of the computation can be restored; execution can be replayed either exactly as it occurred initially or with user-controlled variations; there is no need to know *a priori* what states might be of interest. In addition, if checkpoints and logs are written to stable storage, our mechanism can be used to restore states of computations that cause the system to crash.

1 Introduction

One reason that it is more difficult to debug distributed applications than single-cpu applications of comparable complexity is that global states of a distributed computation are more difficult to observe and manipulate than the purely local states of a single process. This paper presents the design of the Distributed State Restore (DSR) mechanism, which enables a user to manipulate global states in a natural manner for the purpose of debugging distributed programs.

We adopt a process-oriented computational model in which a distributed computation is carried out by a collection of *processes*, each with its own local state, and interacting only by means of message communication. With such a model it is common to define a *global state* of the computation as a vector of *local states*, one for each process participating in the computation.

In such a model there is only a partial time ordering among the events of a distributed computation [Lam78], and therefore among the local states of different processes. Events (or states) that are incomparable in this partial order are said to be *concurrent*. For example, Figure 1 depicts three processes and three messages communicated among them, and the partial order relating the individual send and receive events in time.[1] In this scenario, process P sends message $m1$ before it sends $m2$; we also know that message $m1$ is sent before message $m3$, because message $m1$ is received between the two send events. However we cannot claim that message $m2$ is sent before $m3$, nor is message $m1$ necessarily received before $m2$ is sent. Thus, for example, the receiving of $m1$ and the sending of $m2$ are concurrent events.

It is a simple matter to observe the sequence of local states visited by each process in a distributed computation, and all message communications that take place. However, it may be impossible to infer from this infor-

*Department of Computer Science, Cornell University. This research was begun when this author was visiting IBM. This author is partially supported by an IBM Graduate Fellowship.

[1] In this and other similar diagrams appearing in this paper, each process is represented by its own downward-pointing time line. Message communications are depicted by arrows from one process' time line to another's.

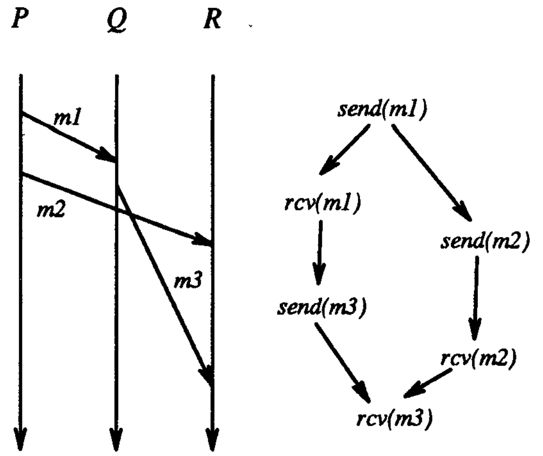

Figure 1: A partially ordered distributed execution.

mation the exact sequence of global states that a computation passed through during its execution; rather, the global states that it *might* have passed through are are those in which the component local states are mutually concurrent. We call such global states *consistent.*

The lack of complete ordering information and the ambiguity in the sequence of global states in a distributed computation make familiar debugging techniques such as single-stepping and setting of breakpoints less well defined than in the single-process case. While it is possible to define protocols that turn the partial order of distributed events into a consistent total order ([Lam78]), such an approach in a debugging environment would destroy the inherent concurrency, and may make certain synchronization-related bugs impossible to exercise during the debugging process. The DSR approach, rather than destroying or hiding the partial order of states, exploits it to provide added flexibility to the user.

DSR can operate in any of three modes:

- In *recording* mode, DSR records information gathered during the course of a distributed computation. This information can be used later to restore consistent global states of the computation and/or replay portions of the computation.
- In *analysis* mode, DSR restores computations to global states as requested by the user. Traditional single thread debugging technology can then support detailed examination of local process states.
- In *replay* mode, DSR uses information gathered in recording mode during a prior execution, to replay that execution from some intermediate global state.

Typically, users might debug a distributed application by running it in recording mode until the application fails or reaches some other interesting point. They will then enter analysis mode, where they can explore the computation in detail, restoring intermediate global states and examining the details of the resulting local states. Starting from any consistent global state, the user can enter replay mode to replay portions of the original execution. Alternatively, the user may request that the application simply be continued, perhaps under recording mode. In that case, DSR will not constrain the new execution, and the computation may or may not take a different course than it did originally, depending on the inherent nondeterminism of the program. Finally, replay mode and recording mode can be used in combination: the user requests replay, but with controlled deviations from the original execution.

It is important to note that DSR is capable of restoring a computation to *any* consistent global state once it has been executed in recording mode.

This paper does not address user interface issues or the precise manner in which the user might interact with DSR, although the above discussion illustrates some of the possible interactions.

The DSR recording, restoring, and replay components are discussed in Sections 2, 3, and 4. Section 5 presents some of the features that could be provided in a debugging environment based on DSR. Section 6 discusses some related work.

2 Recording Mode

When operating in recording mode, DSR performs three activities: logging, checkpointing, and causality tracking.

The logging activity proceeds independently for each process in the distributed computation. It records an ordered sequence of the *nondeterministic* local events that occur in the process, including message receipts (the receive order of messages arising from concurrent send events in different processes is unpredictable), expiration of timeouts, scheduling decisions in a multithreaded process, and others. The entire behavior of a process up to a given point is completely determined by its initial state and the sequence of nondeterministic local events occurring in the process to that point.

The checkpointing activity also proceeds independently in all processes. Each process occasionally checkpoints its local state, solely to expedite the restoration of that state and subsequent states. In general, a process restores a given local state by restoring the most recent prior checkpoint and then replaying its execution to the desired point with the aid of its nondeterministic local event log. The longer the interval between checkpoints, the longer it will take on average to restore an arbitrary local state.

Causality tracking permits DSR to identify consistent global states.

2.1 Events and States

Recall that in our computation model processes interact only via message communication. The partial ordering of events is thus completely determined by the total ordering of local events at each process, plus the constraint that the sending of a message always precedes its receipt. Precisely, let $e_{i,k}$ denote the k^{th} event occurring at process i. Let $e_s(m)$ and $e_r(m)$ denote, respectively, the events of sending and receiving a given message m. Then we have:

$$e_{i,k} \rightarrow e_{j,\ell} \text{ if } \begin{cases} i = j \text{ and } k < \ell; \text{ or} \\ e_{i,k} = e_s(m) \text{ and } e_{j,\ell} = e_r(m), \\ \quad \text{for some message } m. \end{cases}$$

In the first case, the ordering arises because the two events occur in the same process; in the second, the event order is determined by message communication. The relation "→", called the "happens-before" relation, is the smallest transitive relation satisfying the above conditions. As stated earlier, two events that are incomparable by this partial order are said to be concurrent.

If we define each event $e_{i,k}$ as giving rise to a new local state of process i, denoted $s_{i,k}$, then by extension we define a partial ordering on process states as follows:

$$s_{i,k} \rightarrow s_{j,\ell} \text{ iff } e_{i,k+1} \rightarrow e_{j,\ell} \text{ or } e_{i,k+1} = e_{j,\ell}.$$

That is, one process state precedes another if and only if the event terminating the former cannot follow the event giving rise to the latter. Process states that are incomparable under this relation are said to be concurrent. A consistent global state is one in which the component local states are mutually concurrent. To see why this definition of consistent global state makes sense, consider a global state in which two of the component process states are $s_{i,k}$ and $s_{j,\ell}$ with $s_{i,k} \rightarrow s_{j,\ell}$. Such a global state would simultaneously have process i in a state prior to the occurrence of event $e_{i,k+1}$ and process j in a state following the occurrence of event $e_{j,\ell}$. But since either $e_{i,k+1} \rightarrow e_{j,\ell}$ or $e_{i,k+1} = e_{j,\ell}$, it is impossible for $e_{j,\ell}$ to have occurred and not $e_{i,k+1}$. Thus such a global state cannot occur, and must be labeled inconsistent.

DSR maintains ordering information among the local states in a distributed computation by attaching *dependency* information to each message exchanged between processes. Specifically, if $s_{i,k} \rightarrow s_{j,\ell}$, we say that the latter state *depends on* the former. The states upon which a given process depends at any instant of time can be compactly encoded in a vector of length equal to the number of processes participating in the computation. Specifically, the vector lists the highest state number of each process on which the subject process' current state depends. Since dependence on a state $s_{i,k}$ implies dependence on all prior states of process i, the implied dependencies need not be explicitly represented.

It is a simple matter to show that a global state is consistent if and only if there are no component process states $s_{i,k}$ and $s_{j,\ell}$ such that $s_{j,\ell}$ depends on $s_{i,k}$.

2.2 State Intervals

In practice, to reduce the costs of dependency tracking, all the states appearing between two consecutive nondeterministic events can be coalesced into a single *state interval.*[2] We denote the k^{th} state interval in the execution of process i by $S_{i,k}$. Dependence between state intervals is easily defined: $S_{j,\ell}$ depends on $S_{i,k}$ if and only if one of the states included in $S_{j,\ell}$ depends on one of the states included in $S_{i,k}$. We also define the → relation on state intervals: $S_{i,k} \rightarrow S_{j,\ell}$ if and only if $S_{j,\ell}$ depends on $S_{i,k}$.

The above definitions are depicted in Figure 2. In that figure, we see many individual local events occurring in processes P and Q. Some events are deterministic, others are nondeterministic. In this example, all the

[2] Debugging support at a finer granularity than the state interval can be provided by traditional single-thread debugging techniques.

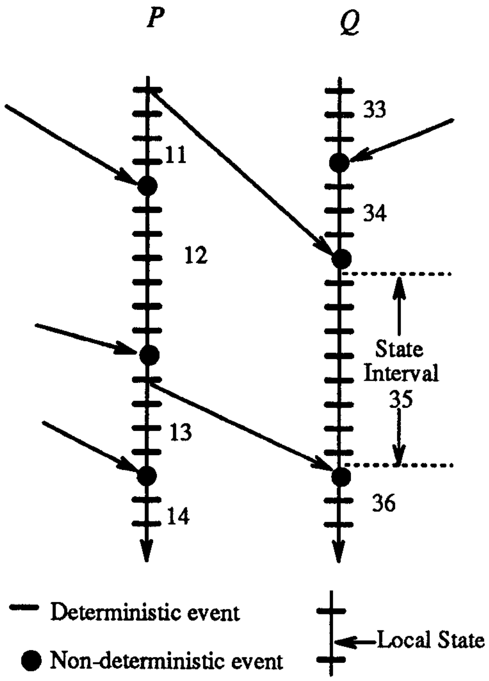

Figure 2: The relationships between events, states, and state intervals.

nondeterministic events are message receipts; note that message send events are deterministic.

A local process state fills the gap in the time-line between consecutive local events. The states bounded by two consecutive nondeterministic events form a state interval. In the diagram, we see P's state intervals 11 through 14, and Q's state intervals 33 through 36. The bounds of Q's state interval 35 are explicitly highlighted.

In the computation depicted in Figure 2, we see that Q's state interval 36 depends on P's state interval 13 (and all of P's earlier state intervals), because a message was sent by P while it was in state interval 13, and received by Q to start its state interval 36.

We can now define what it means for a global state *interval* vector to be consistent: it must not contain two state intervals $S_{i,k}$ and $S_{j,\ell}$ such that $S_{j,\ell}$ depends on $S_{i,k+1}$. We must allow $S_{j,\ell}$ to depend on $S_{i,k}$ because the sending of a message does not increment the sending process' state interval; that state interval may therefore coexist with the new state interval that begins in the receiving process when the message is received. In Figure 2, for example, even though Q's state interval 36 depends on P's state interval 13, a state interval vector containing both state intervals would be considered consistent (provided no other state intervals in the vector caused inconsistencies). However, no consistent state interval vector could contain both P's state interval 12 and and Q's state interval 36.

A consequence of this definition is that some inconsistent global state vectors correspond to consistent global state interval vectors. For example, in Figure 2, P's state interval 13 contains a state (its first state) that precedes the sending of the message that gives rise to Q's state interval 36. That state in P is thus inconsistent with any state in Q's state interval 36. Nevertheless, since there are also states in P's state interval 13 that follow the sending of this message, the two state intervals are considered consistent. We shall see below how the state restoring part of DSR ensures that restored global states are consistent even though dependency information is recorded at the state interval level.

2.3 Dependency Vectors

DSR maintains dependency information as follows:

- A current *dependency vector* is maintained on behalf of each process. A dependency vector is a vector of state intervals, one for each process participating in the computation. The current dependency vector for a given process contains the highest state interval of each other process on which the subject process depends. The entry for a process in its own dependency vector always contains its current state interval.
- When a message is sent, the sending process' current dependency vector is appended to the message.
- When a message is received by a process, the receiving process begins a new state interval and updates its own current dependency vector by taking the piecewise maximum (with respect to the → relation) between it and the vector attached to the incoming message. That is, suppose the current dependency vector entry for process i is $S_{i,k}$ and the incoming dependency vector entry for process i is $S_{i,k'}$. Then if $k' > k$, the current dependency vector is updated to include $S_{i,k'}$ for process i; otherwise, the entry for process i is not changed.

The current dependency vector of a process must be included in all of its checkpoints; likewise, dependency vectors attached to messages are included in the associated log entries.

At the start of the computation, process i is considered to be in state interval $S_{i,0}$, and all local dependency vectors are filled with these initial process state interval numbers.

3 Restoring Global States in Analysis Mode

Suppose that a user specifies a global state to restore by giving its global state interval vector. Then DSR restores to that global state by restoring each process independently to its associated state interval. This is accomplished as follows:

1. The process is restored to the state saved in its most recent checkpoint preceding the required state interval.
2. The process is allowed to run forward, replaying nondeterministic events from its event log, until the required state interval is reached.

After each process has independently restored to its required state interval, the processes still may not be in a consistent global state, as discussed at the end of Section 2. In addition, some messages may be "in-transit" in the sense that the sender of a message was restored to a state following the send event, but its recipient was restored to a state prior to the receive event. The restored processes must coordinate to address these two issues.

During replay, a process will send whatever messages were sent during the original execution. The processes receiving those messages will normally have obtained them from their own logs during replay, or will have started with a checkpointed state following their original receipt. Hence, a mechanism is required to allow processes to detect and ignore duplicate incoming messages. This can be achieved by providing message sequence numbers (unrelated to state interval numbers) for each pair of communicating processes. Each process maintains the number of messages it has received (sent) on each incoming (outgoing) message stream. When a message is sent, the sending process increments the associated send count and tags the message with the new value. The receiving process compares the tag value with its own associated local receive count. If the values match, the message is accepted and the local receive count is incremented; otherwise, the message is ignored.

3.1 Consistency Checks

A user may request the restoration of an inconsistent global state. DSR cannot normally know *a priori* whether a given state request specifies an inconsistent state. It must first restore the requested global state as described above, and then verify that the resulting global state is consistent. The verification is achieved by requiring that each restored process check that it does not depend (according to its restored dependency vector) on any state interval $S_{i,k+1}$ where $S_{i,k}$ is the state interval to which process i was restored. This check can be carried out in linear time by each process after the entire restored state interval vector is broadcast to all processes.

3.2 Underspecified Global States

In many cases, a user in a debugging session will want to focus attention on a subset of a program's processes. Therefore DSR allows the user to underspecify the global state to restore, by not specifying the state interval for some processes.

In response to an underspecified request DSR restores each process mentioned in the request as outlined above, and performs the local consistency check at each process. Each process must only check its restored dependency vector entries that correspond to processes appearing in the restoration request. If the consistency checks all succeed, DSR must then choose a state interval for each of the other processes such that when these state intervals are combined with the original request, the resulting state interval vector is consistent. This can always be done, because circular dependencies can never arise in the actual computation, as shown below.

Suppose mutually consistent state intervals have already been chosen for some, but not all of the processes. Let $\mathcal{X}$ be the set containing those state intervals, and suppose process j is not currently represented in $\mathcal{X}$. Now suppose $S_{j,\ell}$ is the highest state interval of process j that does not depend on any state interval $S_{i,k+1}$ with $S_{i,k} \in \mathcal{X}$. Then $S_{j,\ell}$ is mutually consistent with all the state intervals in $\mathcal{X}$. For suppose not, that is, suppose $S_{m,n} \in \mathcal{X}$ depends on $S_{j,\ell+1}$. By construction, either $S_{j,\ell+1}$ does not exist (and hence the above supposition is vacuous), or $S_{j,\ell+1}$ depends on some $S_{i,k+1}$ with $S_{i,k} \in \mathcal{X}$. Since dependencies are transitive, it must then be that $S_{m,n}$ depends on $S_{i,k+1}$, which contradicts the assumption that the state intervals in $\mathcal{X}$ are mutually consistent.

DSR adds missing processes to a state specification one-by-one, in each case choosing the highest state interval available for that process which is mutually consistent with all prior choices. In practice, other choices may be possible. If the user requires control over these decisions, a complete state interval vector may be built in an interactive fashion, process-by-process, as follows:

1. The user selects one of the processes not yet represented in the restoration request.
2. DSR presents to the user all the state intervals for the selected process that are consistent with the intervals already chosen for other processes.
3. The user selects a state interval from among the choices presented by DSR.
4. The above steps are repeated until all processes are covered in the restoration request.

At any point in this procedure, the user can ask DSR to fill in choices for the remaining unspecified processes. In general, the choices presented by DSR in step 2 will always be a range of consecutive state intervals for the process under consideration.

3.3 Consistent State Intervals With Inconsistent States

As indicated earlier, the coalescing of states into state intervals yields the potential for inconsistent state vectors mapping to consistent state interval vectors. Since our restoration procedure has been cast in terms of state intervals, there is a need to detect and remove any inconsistencies that remain after the requested state intervals have been restored. This can be done using the mechanisms outlined earlier for duplicate message detection.

Suppose an inconsistency arises: a state interval vector containing $S_{i,k}$ and $S_{j,\ell}$ is used for restoring a computation, but although the state interval vector is consistent, $S_{j,\ell}$ depends on $S_{i,k}$. That is, $S_{i,k} \rightarrow S_{j,\ell}$, but $S_{i,k+1} \not\rightarrow S_{j,\ell}$. This can only arise if some message m sent by process i in its state interval k is received by process j to begin its state interval ℓ. Now consider global states in which process i is in state interval k, and process j is in state interval ℓ. In all such global states, process j will have already received message m; but in some, m will not yet have been sent by process i.

The DSR mechanism can guarantee that any restored global state is free of these inconsistencies by checking the message sequence numbers maintained by all processes. Whenever one process appears to have received a message that the sending process did not yet send, the sending process is caused to (deterministically) run forward until the message in question has been sent. If execution of the receiving process is later resumed, the message will be detected as a duplicate and discarded.

Note that the following simple policy, though less flexible than the general scheme just described, is sufficient to avoid inconsistencies without requiring any negotiation among the restored processes: each process is run to the *end*, rather than the beginning, of the state interval to which it must be restored. This ensures that it has sent all messages might appear to have been received by any other process.

3.4 Recovering In-Transit Messages

Once a consistent global state has been achieved, there remains the possibility of in-transit messages. For example, suppose process P sends a message m to process Q. The user may request restoration to a global state in which P has sent m but Q has not yet received it. There is nothing technically wrong with this situation, since real messages actually experience non-zero in-transit times. However, we prefer to restore to a state in which communications have quiesced, meaning that all messages that have been sent have arrived at their destinations.[3]

In this section we assume that, in the process of halting the original computation in order to analyze it, all mes-

[3] Note that message arrival and message receipt are distinct occurrences. Message arrival would normally correspond to asynchronous operating system activity that captures an incoming message from the network and queues it for later retrieval by the intended recipient process. Message receipt would correspond to the actual dequeuing of the message by the receiving process.

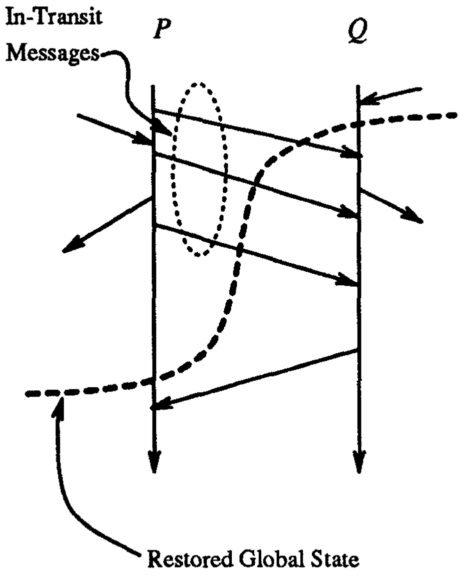

Figure 3: A restored state with in-transit messages.

sage communication is first allowed to quiesce; that is, if there are messages in-transit, they are delivered and recorded in the receiving process' logs before the user is allowed to enter analysis mode. In Section 5 we discuss ways to relax this assumption.

In general, all necessary refilling of message queues can be achieved by comparing send and receive counts among the various processes; messages that need to be delivered may be retrieved from the receiver's message log (looking beyond the messages that were used to fulfill the original restoration request) and placed on the receiving process' incoming message queue.

Suppose, for example, that after restoration to a consistent global state, process P's send count indicates that 132 messages have been sent from P to process Q, but Q's receive count for P has only reached 129. Figure 3 illustrates this scenario. Before continuing execution, Q must search forward in its message log for three messages from P, and insert them, in the order they are encountered, onto its incoming message queue.

4 Executing From a Restored State

As indicated earlier, once DSR has restored a consistent global state several options are available for continuing the execution:

1. DSR can replay the execution exactly as it occurred during the initial recording phase, up to the point where recording was terminated.
2. DSR can allow the computation to continue unconstrained. Depending on the outcome of subsequent non-deterministic events, the resulting execution may not be identical to the original.
3. The user can place constraints on how execution should proceed, so long as those constraints are not inconsistent with the "happens-before" relation inherent in the computation.

In options 1 and 3, the DSR replay mode mechanisms are required. For option 2, there is no need for replay mode. Recording mode support would be appropriate for both options 2 and 3, since in those cases the subsequent execution may be different from the original.

Replaying an execution requires that non-deterministic events take place in a predetermined fashion throughout the computation. In this section, we deal only with message receipt; Section 5 discusses how other kinds of non-deterministic events might be handled.

Message communication introduces non-determinism because the receive order of messages from multiple senders to a single receiver depends on unbounded communication delays. DSR's recording mode mechanisms must therefore record message receive order at each process. When identical replay is requested, DSR delivers messages to each process in the order that was recorded during the original execution.

The user can modify the execution by specifying a different receive order for messages that have arrived but have not yet been received. The only constraint is that multiple messages from a single sender to a single receiver must be received in the order sent.

The option to choose an ordering for future message receipts provides the user with a powerful "what-if" capability. Execution scenarios can be established and run with DSR recording mode enabled, and the resulting recordings can be attached in a tree-like fashion to the original "trunk," allowing the user to maintain and relate multiple variant executions in a single debugging session.

5 Extensions

In this section we explore some of the advanced features that could be provided in an implementation of DSR.

5.1 Trees of Execution Paths

Users may spend considerable time debugging a single distributed application. During such a debugging session they might halt, restore and re-execute the application many times. This activity can be represented by a tree, called a ***session tree***, in which each node represents a consistent global state and an edge represents an execution from that state to another. Each path from the root to a leaf in the session tree represents a single execution of the program. DSR can support the notion of session trees by structuring the checkpoint and log histories in the shape of the session tree, so that each checkpoint and log entry is stored only once.

The session tree naturally lends itself to graphical manipulation in a debugging environment. For example, clicking on a node in the tree might result in DSR restoring the computation to the corresponding consistent state. Clicking on an edge could result in a dialogue in which the user specifies a global state appearing along that edge; DSR would then restore that state. Whenever a new execution path is recorded (either free-running execution from a restored state or a controlled variation of some existing execution), a new edge would appear on the session tree. The user could discard unneeded nodes, edges, or entire subtrees by appropriate mouse actions on the session tree; DSR would discard the associated checkpoints and log entries.

5.2 Other Nondeterministic Events

In the foregoing sections, the only nondeterministic events we have considered in detail are message receipts. In practice, processes are subject to other sources of nondeterminism as well, including shared memory access. In this section we present some approaches to handling such events.

Two forms of access to physical shared memory are possible. If every access is carefully guarded by appropriate synchronizations (e.g. locks, monitors, etc.) with any other processes that might make conflicting concurrent access, we say the access is ***disciplined***; otherwise, the access is ***undisciplined.***

Disciplined access to shared memory provides an adequate hook for DSR: each synchronization must be recorded by DSR in sufficient detail to allow its precise replay from the log. For example, each time a lock is granted, DSR can record the process that acquired the lock and the range of memory addresses covered by the lock. During replay, a lock would be granted only if it were the next recorded lock; other lock requests would block even if the lock was free at the time of the request. Note that DSR cannot record this information independently for all the processes; a single locking activity log for each group of processes that share memory would be required.

If access to shared memory is undisciplined, it would appear that DSR must record all memory accesses in the order they are performed by the physical memory. Not only would the resulting logs be extremely large; the run-time cost of recording each memory access would almost certainly be unacceptable. Nevertheless, it may be possible to record the necessary information without imposing an undue logging burden. For example, Bacon and Goldstein propose special hardware to monitor and log cache transactions in a shared memory multiprocessor [BG91]. By exploiting the cache coherency mechanisms already implemented in such hardware, undisciplined shared memory accesses can be replayed using information recorded only during the relatively infrequent cache misses. Alternatively, undisciplined shared memory access might be considered a bug. Removing these ***access anomalies***, using techniques such as those described in [DS90] or [MC91], would allow DSR to record shared access patterns via synchronization hooks as described above.

5.3 Reducing Log and Checkpoint Volume

Several techniques can substantially reduce the volume of logged data, as discussed in [SBY88] and [Bac90].

First, we can avoid the cost of logging the *content* of a message if the sending process can be rolled back early enough to recreate the message's content. In general, relying on this ability can lead to the *domino effect* in which processes force each other to roll back to earlier and earlier states in order to regenerate needed messages

[Ran75]; in the extreme, the entire distributed computation may need to be reexecuted from its beginning, despite a multitude of intermediate checkpoints. The domino effect can be prevented by limiting reliance on message regeneration for replay, that is, by logging the contents of sufficiently many messages. Alternatively, globally coordinated checkpoints can create a barrier beyond which rollback will not be required [CL85].

As another optimization, various data compression techniques can be applied to the stream of logging information. For example, run-length encoding could be applied to lock activity logs. Bacon suggests logging based on *predictor functions* that effectively reduces the "happens-before" relationship by dropping the order of read-only message receipts from the log.

Incremental checkpoints can substantially reduce the storage requirements of DSR's checkpointing activities, by saving only that portion of a process' state that has changed since the prior checkpoint. Additionally, if the size of a process' state fluctuates greatly during the course of its execution (such as a process that repeatedly allocates and frees large arrays), DSR could attempt to take checkpoints during the low points of this fluctuation.

5.4 Halting Without Quiescence

In discussing the problem of recovering in-transit messages during state restoration, the assumption was made that when the process was originally halted prior to restoration, all communications were allowed to quiesce, thereby guaranteeing that in-transit messages would be available in the logs of the intended receivers. The quiescense requirement can be relaxed by making use of the message regeneration techniques suggested above in connection with the logging of message data. Specifically, when an in-transit message is detected, the sending process can be transparently restored to a state prior to the send event and then replayed in order to regenerate the required messages. All earlier discussion of the domino effect applies here.

5.5 Debugging Programs That Crash the System

Normally, checkpoints and logs can be saved in volatile memory. However, if DSR is careful to migrate checkpoints and logs to stable memory, programs can be debugged even though they crash the debugging environment. In practice, a programmer need only enable this feature as the application nears its crash point. The migration of logging information to stable storage can be done either synchronously with message receipt, or asynchronously. The former choice would result in algorithms resembling pessimistic log-based recovery schemes [Bar78, BBG$^+$89], while asynchronous logging would be suggestive of optimistic recovery algorithms [SY85, Joh89].

6 Related Work

Much recent work has studied the problem of debugging and state restoration in parallel and distributed systems. Research on debugging has presented algorithms for recording a concurrent computation during execution and then replaying it later for debugging [LMC87, LR85, GKY90, Smi84, GMGK84, JLSU85].

Leblanc [LMC87] models a parallel computation as a set of processes communicating only through *shared objects.* During the computation each write to a shared object results in the identity of the process performing the write and a count of the number of reads performed since the prior write. This information is sufficient to deterministically replay the execution from its beginning; presumably, coordinated checkpoints could be used for replay from an intermediate starting point. Prior to replay, the user can set breakpoints in processes. When replay hits the breakpoints, all the processes are guaranteed to eventually quiesce to a consistent global state. This approach has been used to debug programs on the BBN butterfly shared memory multiprocessor.

DSR contrasts with this approach in that uncoordinated checkpoints can be used to support replay from intermediate states. In addition, by allowing the user to manipulate global state vectors, we believe that DSR can provide new and natural modes interaction between the user and the debugging environment.

Some researchers have proposed the use of coordinated checkpoints to support restoration of intermediate global states in a distributed computation [CL85, MC88, KT87, CT90]. Recently, it has been pointed out the the message overhead required to coordinate checkpoints can be reduced by exploiting specific properties of interconnection topologies [LNP91].

Coordinated checkpoints by themselves support restoration only of the states at which checkpoints are taken. If done in conjunction with logging, they increase run-time overhead in exchange for quicker restoration, due to the need for run-time coordination and the elimination of such a need at restoration time.

If restoration is supported only to states saved in coordinated checkpoints, the debugging environment requires that the user be able to identify the interesting states of the computation in advance. Such knowledge can often be obtained only via extensive debugging sessions, resulting in a "divide-and-conquer" approach to zeroing in on interesting states. We believe that DSR's ability to restore any consistent global state of a recorded execution offers a great advantage over such an approach.

7 Conclusions

We have presented a mechanism called Distributed State Restorer (DSR) that allows users to debug their distributed applications by manipulating the global states of a computation in a natural and effective manner. We have identified three major modes in which the user operates with DSR: recording mode for collecting information about a particular execution of the application; analysis mode in which intermediate global states can be restored and examined; and replay mode which can replay a prior execution from a given state, either precisely or under user-controlled variations.

We believe that the mechanisms provided by DSR form an essential component of a comprehensive facility for debugging distributed applications. By combining DSR with appropriate graphical user interfaces and other debugging facilities (traditional debugging support for individual processes in isolation, anomaly detection and other execution analysis tools, etc.), we believe it is possible to provide a powerful debugging environment for distributed applications.

Though, DSR has not been implemented, many of the algorithms are similar with those of Optimistic Recovery [SY85], of which an experimental prototype implementation exists for a Mach platform [ABB$^+$86].

References

[ABB$^+$86] Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer Usenix Conference*, July 1986.

[Bac90] David F. Bacon. How to log all filesystem operations (while only writing a few to disk). Technical Report RC, IBM T.J. Watson Research Center, 1990.

[Bar78] J. F. Bartlett. A 'nonstop' operating system. In *11th Hawaii International Conference on System Sciences*, University of Hawaii, 1978.

[BBG$^+$89] Anita Borg, Wolgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[BG91] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CT90] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, 1990.

[DS90] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *2nd ACM Conference PPoPP*, pages 1–10, March 1990.

[GKY90] German S. Goldszmidt, Shmuel Katz, and Shaula Yemini. High level language debugging for concurrent programs. *Transactions on Computer Systems*, November 1990.

[GMGK84] Hector Garcia-Molina, F Germano, and W. Kohler. Debugging a distributed computer system. *IEEE Transactions on Software Engineering*, se-10(2):210–219, March 1984.

[JLSU85] J. Joyce, Greg Lomow, K. Slind, and Brian W. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1985.

[Joh89] David B. Johnson. *Distributed System Fault Tolerance Using Message Logging.* PhD thesis, Rice University, 1989.

[KT87] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, se-13(1), January 1987.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LMC87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, c-36(4), April 1987.

[LNP91] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing multicomputer applications. Technical Report CS-TR-315-91, Princeton University, Department of Computer Science, 1991. Submitted to the Symposium on Reliable Distributed Systems, Pisa Italy, Sep 1991.

[LR85] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *5th International Conference on Distributed Computer Systems*, pages 515–522, 1985.

[MC88] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *International Conference on Distributed Computer Systems*, 1988.

[MC91] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, Santa Clara, CA, April 1991.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[SBY88] Robert E. Strom, David F. Bacon, and Shaula Alexander Yemini. Volatile logging in n-fault-tolerant distributed systems. In *The Eighteenth Annual International Symposium on Fault-Tolerant Computing: Digest of Papers*, pages 44–49, June 1988.

[Smi84] Edward T. Smith. Debugging tools for message-based, communicating processes. In *4th International Conference on Distributed Computer Systems*, pages 303–310, 1984.

[SY85] Robert E. Strom and Shaula Alexander Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.