Optimistic Recovery in Multi-threaded Distributed Systems [*]

Om P. Damani, Ashis Tarafdar
Dept. of Comp. Sciences
Univ. of Texas at Austin
{damani,ashis}@cs.utexas.edu

Vijay K. Garg
Dept. of Electr. and Comp. Engg.
Univ. of Texas at Austin
garg@ece.utexas.edu

Abstract

The problem of recovering distributed systems from crash failures has been widely studied in the context of traditional non-threaded processes. However, extending those solutions to the multi-threaded scenario presents new problems. We identify and address these problems for optimistic logging protocols.

There are two natural extension to optimistic logging protocols in the multi-threaded scenario. The first extension is process-centric, *where the points of internal non-determinism caused by threads are logged. The second extension is* thread-centric, *where each thread is treated as a separate process. The process-centric approach suffers from false causality while the thread-centric approach suffers from high causality tracking overhead. By observing that the granularity of failures can be different from the granularity of rollbacks, we design a new* balanced *approach which incurs low causality tracking overhead and also eliminates false causality.*

1. Introduction

Multi-threading is becoming increasingly common in distributed systems owing to the need for light-weight concurrency. We address the problem of recovering multi-threaded distributed systems from process crash failures. Although recovery has been a widely studied problem in traditional non-threaded systems [5], extending these solutions to the multi-threaded scenario gives rise to new problems. We address those problems for the optimistic logging protocols.

The traditional distributed recovery problem deals with recovering a distributed system from process crash failures. One approach to solving the recovery problem is log-based rollback recovery, which combines checkpointing and message logging. When a failure occurs, the distributed system can make use of the checkpoints and message logs to restore itself to a consistent global state. We focus on optimistic logging, an important class of log-based rollback recovery.

Optimistic logging protocols log messages to stable storage asynchronously, thus incurring low failure-free overhead. On a failure, some unlogged messages may be lost, resulting in the loss of some states of the failed process. Furthermore, this results in the rollback of states on other non-failed processes that causally depend on the lost states. In order to determine which states need to be rolled back, the causal dependencies between states needs to be tracked. This can be implemented by having all messages piggyback a dependency vector of size $O(n)$, where n is the number of processes in the system [15].

While extending this solution to multi-threaded processes we have two natural choices: a *process-centric* approach and a *thread-centric* approach. In the process-centric approach, the internal non-deterministic events caused by threads are logged [7, 14]. With this provision, other researchers have used traditional optimistic protocols. This, however, gives rise to the problem of false causality between threads of a process. This problem has two serious repercussions. First, during failure-free mode, it causes the unnecessary blocking of outputs to the environment. Second, during recovery from a failure, it causes unnecessary rollbacks.

Attempting to eliminate false causality leads to the thread-centric approach. Here, each individual thread is treated as a process and a process crash is treated as multiple concurrent thread crashes. In this approach, during failure-free operation, causality is tracked at the level of threads. This makes causality tracking an expensive operation requiring a dependency vector of size $O(nm)$, where n is the number of processes and m is a bound on the number of threads per process. This increases the message size overhead, as well as space and time overhead.

Thus the process-centric and the thread-centric approaches present a trade-off between false causality and tracking overhead. We make the observation that processes fail independently and are thus *failure units* and that threads may be rolled back independently and are thus *rollback*

[*] supported in part by the NSF ECS-9414780, CCR-9520540, a General Motors Fellowship, Texas Education Board ARP-320 and an IBM grant

units. Therefore, it is sufficient to track the dependency of threads on processes. This *balanced* approach tracks causal dependencies using a dependency vector of size $O(n)$. At the same time it eliminates false causality, since threads are rolled back independently.

In Section 2, we present some background on optimistic recovery in traditional non-threaded environments. In Section 3, we describe the two natural extensions – process-centric logging and thread-centric logging – and the associated false causality versus tracking overhead trade-off. In Section 4, we describe our new *balanced protocol* for optimistic recovery in multi-threaded distributed systems. Section 5 is a note on generalizing the ideas of the paper.

2. Background: Optimistic Recovery

2.1. System Model and the Recovery Problem

We consider an application system consisting of n processes communicating only through messages. The communication system used is unreliable, in that it can lose, delay, or duplicate a message. The environment also uses messages to provide inputs to and receive outputs from the application system. Each process has its own volatile storage and also has access to stable storage. The data saved on volatile storage is lost in a process crash, while the data saved on stable storage remains unaffected by a process crash.

A process execution is a sequence of states. The states may be divided into *state intervals* consisting of the states between two consecutive message receipts by the application process. In single threaded systems, the execution within each interval is assumed to be completely deterministic, i.e., actions performed between two message receives are completely determined by the content of the first message received and the state of the process at the time of the first receive. As we will see, in multi-threaded systems, non-deterministic thread scheduling affects the state of a process. It is sufficient for our purposes to view process executions at the granularity of state intervals and not states. Therefore, for simplicity, we will sometimes use *states* to mean *state intervals*.

All n process executions together constitute a system execution. Two physical system executions are considered equivalent if their interaction with the environment is the same.

A process fails by simply crashing. In a crash failure, a process stops executing and loses the data in its volatile storage. The process does no other harm, such as sending incorrect messages. Pre-failure states of a process that cannot be recreated after a failure are called lost states.

The application system is controlled by an underlying recovery system. The type of control may be of various forms, such as saving a checkpoint of the application process, stopping an application process, adding control information to the state of an application process, adding control information to a message, rolling back the application to an earlier state, etc.

The *recovery problem* is to specify the behavior of a recovery system that controls the application system to ensure that despite crash failures, the system execution remains equivalent to a possible crash-free execution of the stand-alone application system.

2.2. Optimistic Logging

Log-based rollback recovery protocols [5] rely on checkpoints and message logs, using them during recovery to restore the whole system to a consistent global state (one in which every received message was sent). It is guaranteed that this restored state is one which could possibly have happened in a failure-free system execution and, therefore, this approach solves the recovery problem. Depending on when and where the received messages are logged, the log-based rollback recovery schemes can be divided into three categories: pessimistic, optimistic, and causal [5]. In this paper, our focus is on optimistic logging protocols.

We first present an example and then use it to specify the details of how a traditional optimistic logging protocol operates. The protocol we present is similar in spirit to the ones presented in [3, 15].

Example

An example of an optimistic recovery system is shown in Figure 1. Solid horizontal lines show the useful computation, and dashed horizontal lines show the computation that is either lost in a failure or rolled back by the recovery protocol. In the figure, $c1$ and $c2$, shown by squares, are checkpoints of processes $P1$ and $P2$ respectively. State intervals are numbered from $s0$ to $s7$ and they extend from one message receive to the next. The numbers shown in rectangular boxes will be explained later in this chapter.

In Figure 1(a), process $P1$ takes a checkpoint $c1$, acts on some messages (not shown in the figure) and starts the interval $s0$. $P1$ logs to stable storage all messages that have been received so far. It starts interval $s2$ by processinging the message $m0$. In interval $s2$, message $m2$ is sent to $P2$. $P1$ then fails without logging the message $m0$ to stable storage or receiving the message $m1$. It loses its volatile memory, which includes the knowledge about processing the message $m0$. During this time, $P2$ acts on the message $m2$.

Figure 1(b) shows the post-failure computation. On restarting after the failure, $P1$ restores its last checkpoint $c1$, replays all the logged messages and restores the interval $s1$. It then broadcasts a failure announcement (not shown in Figure 1). It continues its execution and starts interval $s6$ by processing $m1$. $P2$ receives the failure announcement in interval $s5$ and realizes that it is dependent on a lost state. It rolls back, restores its last checkpoint $c2$, and replays the

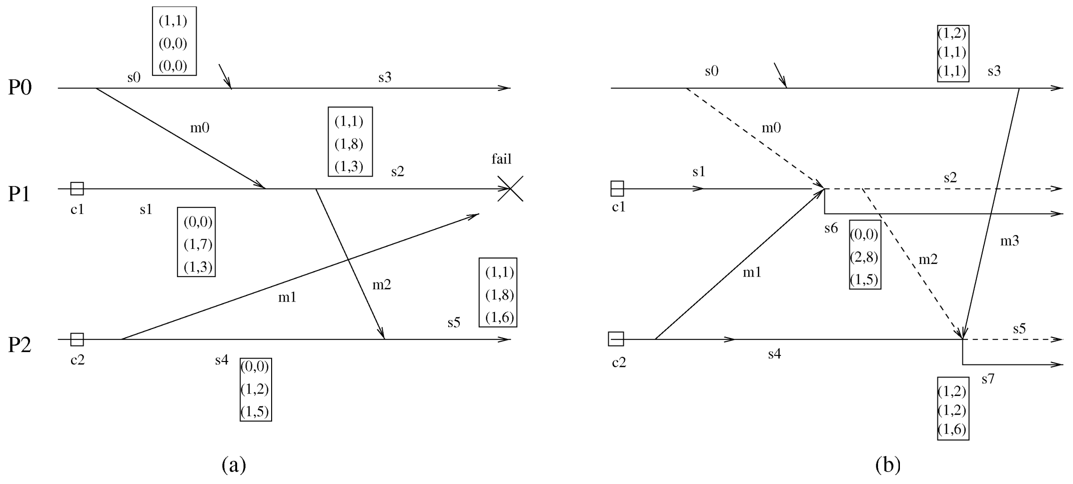


Figure 1. Example: Optimistic Recovery in Action

logged messages until it is about to process $m2$, the message that made it dependent on a lost state. It discards $m2$ and continues its execution by processing $m3$. The message $m2$ is not regenerated in post-failure computation. $P0$ remains unaffected by the failure of $P1$.

Detecting Orphans: Causally Precedes

As just seen, in optimistic logging, some messages may be lost in a failure. This may result in some lost states on the failed process. All states that "causally depend" on such lost states must also be detected and rolled back. Such states are known as *orphans*.

The intuitive notion of "causally depends" that we have used is formalized by the following relation. Let *causally precedes* (denoted by $\rightarrow$) be the smallest transitive binary relation on state intervals satisfying the following two conditions:

- $u \rightarrow v$ if the processing of an application message in state u results in state v, (for example, $s1 \rightarrow s6$ in Figure 1(b)),
- $u \rightarrow v$ if the processing of an application message sent from u starts v (for example, $s2 \rightarrow s5$ in Figure 1(a)).

By $s \underline{\rightarrow} u$, we mean $s \rightarrow u$ or $s = u$. Note that a failure or a rollback does not start a new interval. It simply restores an old interval. Now we can define an orphan state as:

$$orphan(s) \equiv \exists u : lost(u) \wedge u \underline{\rightarrow} s$$

Tracking Causal Dependencies: Dependency Vectors

The causally precedes relation needs to be tracked for orphan detection. For this purpose, a *dependency vector* (dv) is used. The dv of a process P_i has n *state interval indices*, where n is the number of processes in the system and a *state interval index* is a tuple containing an *incarnation number* and a *sequence number*.

Whenever a process fails and is restarted, it is said to be in a new *incarnation*. The incarnation number in the i'th entry of its dv is its own incarnation number. The incarnation number in the j'th entry is equal to the highest incarnation number of P_j which causally precedes P_i. Let state interval index e be $(t,\, i)$. Then, we define a total ordering, $<$, on state interval indices as $e_1 < e_2 \equiv (t_1 < t_2) \vee [(t_1 = t_2) \wedge (i_1 < i_2)]$.

Each process piggybacks its dv on every outgoing message. Before processing a message, a process updates its dv by taking a componentwise maximum of its dv with the dv of the incoming message and incrementing its own sequence number.

An example of dv is shown in Figure 1. The dv of each state is shown in a rectangular box near it. The k'th row of the rectangular box corresponds to $dv[k]$.

It has been previously demonstrated that dependency vectors track the causally precedes relation, and therefore, can be used to detect orphans [3, 15].

Recovering from a Crash

When a process P_j fails, it restores its most recent checkpoint and replays the logged messages that were processed after that checkpoint. Next, P_j broadcasts a failure announcement containing its state index, which is the ending index number of the failed incarnation. In Figure 1, failure announcement of $P2$ contains (1,7). It then waits for an acknowledgment from all processes.

Upon receiving a failure announcement, a process P_i compares its dv with that index. If the dv shows that P_i's state depends on a higher-index interval of the failed incarnation of P_j, P_i rolls back to undo the orphan states. Similarly, it discards the orphan messages from its log. It also saves the received failure announcement in an *incarnation end table* to discard any orphan message that may arrive in future. It then sends an acknowledgment to the sender of the failure announcement.

Handling Output Commits

Distributed applications often need to interact with the outside world. Examples include setting hardware switches, performing database updates, printing computation results, displaying execution progress, etc. Since the outside world in general does not have the capability of rolling back its state, the applications must guarantee that any output sent to the outside world will never need to be revoked. This is called the *output commit problem*.

In optimistic recovery, an output can be committed when the state intervals that the output depends on have all become *stable* [15]. (An interval is said to be *stable* if it can be recreated from the information saved on stable storage). To determine when an output can be committed, each process periodically broadcasts a logging progress notification to let other processes know which of its state intervals have become stable.

3. Optimistic Recovery with Multi-threaded Processes

3.1. System Model

Each process contains a set of threads and a set of shared objects. Threads of different processes communicate only through messages. Threads of the same process communicate through shared objects and messages. Any other form of communication is allowed between threads, as long as it can be modeled using shared objects or messages. For example, wait-notify synchronizations can be modeled using messages.

Threads of a process crash together. This happens not only in hardware crashes but also in most software crashes because threads share the same address space and are not protected from each other by the operating system.

The recovery system can restore the state of an individual thread or shared object to an old state without affecting the other threads or shared objects. This assumption will be discussed in Section 4.4.

3.2. Extending Optimistic Recovery

We now investigate how to extend the optimistic recovery protocol given in Section 2 to multi-threaded environments.

Strom and Yemini [15] presented the original optimistic protocol not in terms of processes, but in terms of *recovery units*. A recovery unit is a single unit of execution in optimistic recovery systems. Recovery units fail as a unit and roll back in response to another unit's failure.

In previous sections, we chose individual processes as recovery units. In a multi-threaded environment, there are two natural candidates for the recovery unit: a process or a thread.

3.2.1 Process-centric Logging

In treating a process as a recovery unit in a multi-threaded system, there is another source of non-determinism apart from the order of message receives. Depending on the scheduling, the threads may access shared objects in a different order. Therefore, after a failure, replaying the message log to a process is not sufficient to recreate the desired states.

To solve this problem, Goldberg et. al. [7] require that shared objects be accessed only in locked regions. The order in which threads acquire locks is logged. During a replay, the same locking order is enforced. This trace-and-replay technique has also been used in concurrent debuggers [10, 16].

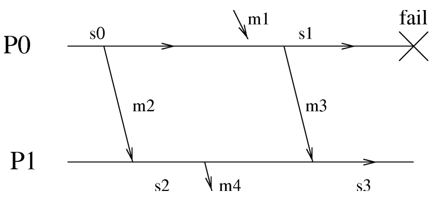
Another approach has been used by Elnozahy and Slye [14]. They focus on uniprocessor multi-threaded environments in which the points of non-determinism can be reduced to the thread switches. Therefore, they log the order of thread switches and ensure that thread switches occur in the same order during replay. Again, this approach has been used in concurrent debuggers [11, 13].

Given that the non-determinism due to thread scheduling can be tracked and replayed, the general optimistic recovery approach described before can be used with a process as a recovery unit.

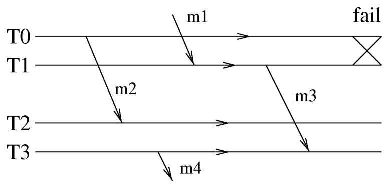
The False Causality Problem

An example of how the process-centric approach operates is shown in Figure 2(a). Receive of message $m1$, $m2$ and $m3$ starts the intervals $s1$, $s2$ and $s3$ respectively. When $P0$ fails and loses state interval $s1$, $P1$ has to roll back state interval $s3$. In the figure, the threads in each process are not shown, since processes are the recovery units.

Let us now take another look at this scenario at the level of threads instead of processes. Figure 2(b) shows the same scenario at the level of threads. The figure shows that process P1 consists of two threads T2 and T3. Therefore, $s3$ is an interleaving of the states of $T2$ and $T3$. Suppose that, after $m3$ was received, there was no shared object interactions between $T2$ and $T3$. So, only the states on $T3$ were really caused by $m3$ and needed to roll back. The states in $s3$ belonging to $T2$ were rolled back unnecessarily. This is



(a) Process-centric Logging



(b) Thread-centric Logging

Figure 2. Extending Optimistic Logging

said to be due to *false causality* induced between the states in the two threads.

Besides causing unnecessary rollbacks during recovery, false causality has an unwanted effect in failure-free mode as well. In Figure 2(a), the output message $m4$ from $s2$ cannot be committed until $s0$ has become stable. However, the thread view in Figure 2(b) shows us that this was, in fact, unnecessary. The waiting was a result of false causality induced between $m2$ and $m4$. Thus, false causality also increases the latency of output commits. As seen in the previous section, the latency of output commits is an important factor for message logging protocols.

How often does false causality arise? Lewis and Berg [9] have divided multi-threaded programs into two main categories: *inherently multi-threaded programs* and *not obviously multi-threaded programs*. Inherently multi-threaded programs require multi-threading for ease of programming and not for speedup. These programs have highly independent tasks that are naturally expressed as threads. Some examples of such programs are: servers which handle multiple requests simultaneously, debuggers which monitor a program while keeping multiple displays active at the same time, and simulators which simulate different entities that operate simultaneously. The other class of not obviously multi-threaded programs are those that require multi-threading for speedup on a multi-processor machine. Such programs have tightly-coupled threads that interact frequently through shared memory. Some examples are numerical programs and fine-tuning bottlenecks in existing code. Of these two categories, the inherently multi-threaded programs have highly independent threads which do not interact frequently, and therefore, would display false causality more often. Our focus will be on this important category of multi-threaded applications.

Given that the false causality problem is an important concern, how can it be addressed? The problem arises because threads, which are independent units, are grouped together as a single unit. To solve this problem, we now study an approach that models each thread as a recovery unit.

3.2.2 Thread-centric Logging

Threads can be modeled as recovery units since they can be rolled back independently. Failure of a process is modeled as concurrent multiple failure of all its threads.

There are a number of important issues that arise when treating threads as recovery units in the optimistic logging scheme. First, in addition to dependencies between threads due to messages, there are also dependencies caused by shared objects. These new dependencies must be tracked. Second, on a failure, just as a thread may have to roll back, a shared object may also have to roll back. Thus, orphan detection must be carried out for threads, as well as for shared objects. Third, both threads and shared objects must be restored to a checkpoint and replayed.

In order to address these issues, we now describe a way to model shared objects using messages and threads. A fictitious thread is associated with each shared object whose state is the same as that of the corresponding object. We model each method invocation on a shared object as a pair of messages between the invoking thread and the thread associated with the object. The first message is sent by the invoking thread and contains the method identifier and the method parameters. The second message is sent by the thread associated with the object and contains the return value of the method. This greatly simplifies presentation of the protocols because now messages are the only form of communication in the whole distributed system.

Further, the simplified model captures a way to deal with each of the three issues mentioned above. The new dependencies are tracked by treating the shared object accesses as messages and associating a vector with each shared object. Similarly, orphan detection and replay of shared objects is done just as in threads.

Since we have dealt with the new issues, all that remains is to apply the general optimistic logging scheme. So the computation in Figure 2(a) appears as that in Figure 2(b). When $T0$ and $T1$ fail, the thread-interval sending $m2$ is not lost in the failure. So only $T3$ and not $T2$ is rolled back. Also, message $m4$ is committed without waiting for the interval sending $m2$ to become stable.

Therefore, the thread-centric approach clearly reduces

type sii_type: (inc: int, seq: int); // type representing state interval index with incarnation // and sequence numbers

Thread T_j	
dv	array[n] of sii_type; // dependency vector
IET	array[n] of set of sii_type; // incarnation end table

Process P_i	
$\underline{sii}$	sii_type; // state interval index
$\underline{log}$	list of untyped objects; // log for messages and $\underline{sii}$ values
$\underline{LOG}$	list of untyped objects; // stable log for messages, // $\underline{sii}$ values, and checkpoints

Figure 3. Variables Used in the Protocol

false causality. The dependency tracking overhead, however, is greatly increased. A main factor in this overhead is that, instead of $O(n)$ entries, each dependency vector now has $O(mn)$ entries (where m is the maximum number of threads per process). A more detailed discussion on this overhead will be presented in Section 4.3.

An Inherent Trade-off?

The process-centric and thread-centric approaches offer a trade-off between dependency tracking overhead and extent of false causality. This trade-off seems to be an inherent one as it arises from the choice of granularity of the recovery unit. A larger recovery unit introduces more false causality and has lower tracking overhead than a smaller one. In database systems, an analogous trade-off exists between lock maintenance overhead and extent of false causality while choosing the lock granularity. Surprisingly, in multithreaded recovery, this trade-off can be avoided by a scheme that we now present.

4. The Balanced Protocol

We observe that a recovery unit plays two distinct roles in optimistic recovery. The first role is that of a *failure unit*. The defining characteristic of a failure unit is that it fails as a unit. The second role is that of a *rollback unit*. A rollback unit can be rolled back and restored to a previous state independently. For example, in the process-centric protocol, the process was both the failure unit and the rollback unit, whereas in the thread-centric protocol, the thread was the failure unit and rollback unit.

A general observation we can make about optimistic recovery is that: *to detect orphans, it is sufficient for a rollback unit to track its transitive dependency on a failure unit*. Then, the failure of a failure unit causes all orphaned rollback units to rollback, bringing the system back to a consistent state.

Thus, choosing a larger granularity failure unit reduces the dependency tracking overhead since there are fewer entities to track. Also, choosing a larger granularity rollback unit increases the extent of false causality since multiple entities are forced to roll back together. In the previous section, we saw that the trade-off between dependency tracking overhead and false causality depended on the granularity of the recovery unit. The separation of roles into failure units and rollback units allows the trade-off to be avoided, by choosing a larger granularity failure unit (process) and a smaller granularity rollback unit (thread). This is the central idea for our *balanced protocol*.

However, to achieve the separation of roles, we must simultaneously deal with both thread state intervals and process state intervals. This requires a redefinition of a process state interval as a "consistent" set of thread state intervals. We define this notion of consistency formally in [4].

4.1. The Balanced Protocol: Details

The protocol specifies the actions to be taken by the recovery system. The actions are divided into two categories, those for a process (failure unit) and those for a thread (rollback unit).

Figure 3 shows the variables used in the protocol. We use capital letters to indicate variables on stable storage (e.g. IET) and small letters to indicate variables on volatile storage (e.g. dv). Global variables common to all threads (process variables) are underlined (e.g. $\underline{sii}$). Calls to the run-time environment start with an underscore (e.g. $\_Send(data, dv)$ calls the environment, while $Send(data)$ calls the program's function).

Each thread maintains a dependency vector dv and an incarnation end table IET in order to detect orphans. The $\underline{sii}$ indicates the current process-wide state interval index. The $\underline{log}$ and $\underline{LOG}$ are the volatile and stable logs respectively. The volatile log is used as a buffer for the log before it is made stable. The stable log is used to recover from a process failure. The stable and volatile logs are also used to independently restore a thread to a previous state.

The actions of the protocol may be divided into two types: normal mode and recovery mode. Figure 4 lists the

```
Thread $T_j$

Initialize:
    $\forall k \neq j :\ dv[k]$ := (0, 0)
    $dv[j]$ := (1, 1)
    $\forall k :\ IET[k]$ := {}

Send($data$):
    _Send($data$, $dv$);

Receive:
    repeat
        $m$ := _Receive();
    until (¬ orphan($m.dv$, $IET$));
    $\underline{log}$ := $\underline{log}$ + ($m$, $j$);
    $dv$ := max($dv$, $m.dv$);
    $dv[j]$ := $\underline{sii}$;
```

```
Process $P_i$

Initialize:
    $\underline{sii}$ := (1,1)
    $\underline{log}$ := null
    $\underline{LOG}$ := null

Take Checkpoint:
    $\underline{log}$ := $\underline{log}$ + $\underline{sii}$;
    $\underline{LOG}$ := $\underline{LOG}$ + $\underline{log}$;
    $\underline{log}$ := null;
    $\underline{LOG}$ := $\underline{LOG}$ + _Checkpoint();
    $\underline{sii}.seq$ := $\underline{sii}.seq$ + 1;

Make Message Log:
    $\underline{LOG}$ := $\underline{LOG}$ + $\underline{log}$;
    $\underline{log}$ := null;

Start State Interval:
    $\underline{log}$ := $\underline{log}$ + $\underline{sii}$;
    $\underline{sii}.seq$ := $\underline{sii}.seq$ + 1;
```

Figure 4. Protocol for Normal-mode Operation

normal mode protocol. For simplicity, all actions are assumed to be atomic.

First, in order to ensure that process state intervals are consistent, the recovery system periodically starts a new state interval (**Start State Interval**) by incrementing the global $\underline{sii}$ value. The old $\underline{sii}$ is queued in the log marking the end of the previous receive set in the log. On the next receive event, each thread T_j assigns its local $dv[j]$ entry the value of this global $\underline{sii}$ (**Receive**), thus keeping track of the process state interval it belongs to.

Each thread keeps track of the highest process state interval that it is aware of using its local dependency vector dv. The dependency vector mechanism is the same as before. (**Send, Receive**).

On receiving a message, a thread must discard it, if it is an orphan message (**Receive**). It can detect this by looking at its incarnation end table IET which is updated appropriately in the recovery mode of the protocol. More precisely, we specify the predicate

$$orphan(dv, IET) = \exists j :\ \exists(t, x) \in IET[j] : \\ (t = dv[j].inc)\ \wedge\ (x < dv[j].seq)$$

A received message is logged in the global volatile log $\underline{log}$ (**Receive**). This log totally orders all receives of all threads in their real time order, marking the end of receive sets by storing their $\underline{sii}$ values (**Start State Interval**). Periodically, this volatile log is flushed to stable storage (**Make Message Log**).

Periodically, checkpoints are also taken. The volatile log is flushed to the stable log and a checkpoint is appended (**Take Checkpoint**). The old $\underline{sii}$ value is logged before the checkpoint and incremented after the checkpoint. This ensures that every checkpoint is exactly between two state intervals. We assume that in addition to the application system state, the checkpoint includes the dependency vector dv for each thread.

The recovery mode protocol is listed in Figure 5. On a crash failure, the crashed process restores its last checkpoint in the stable log, $\underline{LOG}$ (**Restart After Crash**). All threads are replayed from this point using messages and $\underline{sii}$ values from $\underline{LOG}$ upto the last complete process state interval.

Next, the crashed process broadcasts an announcement of its failure to all threads of all other processes. This announcement includes its recovered state interval index indicating the end of that incarnation. This broadcast must be reliable in order to ensure that the system returns to a consistent state. Reliability may be ensured by repeating the broadcast periodically. The process then blocks, waiting for an acknowledgement from all threads. Once all of these are received, it starts a new incarnation by appropriately updating its state interval index.

When a thread receives a failure announcement, it first records the announcement in its incarnation end table IET (**Handle Fail_Announce**). This will be later used in normal mode to discard orphan messages (**Receive**). It then decides if it is an orphan based on its dependency vector

```
Thread $T_j$

Handle Fail_Announce($sender\_id$, $sender\_sii$):
  $IET[sender\_id] := IET[sender\_id] \cup \{sender\_sii\}$;
  if (orphan($dv$, $IET$))
    delete_orphans($\underline{log}$, $\underline{LOG}$, $IET$)
    restore last checkpoint of $T_j$ in $\underline{LOG}$;
    locally replay thread $T_j$ using messages and $\underline{sii}$
      values after last checkpoint of $T_j$ in $\underline{LOG}$ and $\underline{log}$;
  send $Ack\_Announce$ to $P_{sender\_id}$
```

```
Process $P_i$

Restart After Crash:
  restore last checkpoint in $\underline{LOG}$;
  globally replay all threads in $P_i$ using messages
      and $\underline{sii}$ values after last checkpoint in $\underline{LOG}$;
  reliably broadcast $Fail\_Announce(i, \underline{sii})$ to
      all threads in all other processes;
  wait for $Ack\_Announce$ from all;
  $\underline{sii}.inc := \underline{sii}.inc + 1$;
  $\underline{sii}.seq := 1$;
```

Figure 5. Protocol for Recovery-mode Operation

and its newly updated IET. If it is, it must delete all orphan entries from its stable and volatile logs. Next, it must restore the last thread checkpoint from the stable log. It then replays itself to its latest state using the messages and $\underline{sii}$ values in the stable and volatile logs. This will bring it to the latest state that is not an orphan with respect to the received failure announcement. Note that the other threads and, in particular, the global $\underline{sii}$ remain unaffected by this action. Finally, it sends an acknowledgement to the sender of the failure announcement.

To complete the protocol, we must add logging progress notification to accomodate output commit. However, we omit these details because they are identical to those in traditional optimistic protocols.

4.2. An Example

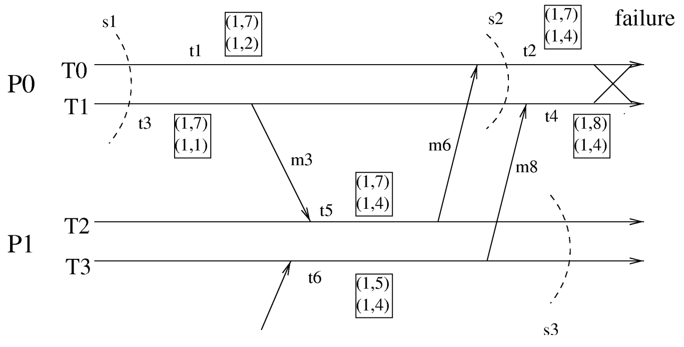


Figure 6. Example: Balanced Protocol

An example of our protocol in action is shown in Figure 6. Threads $T0$, $T1$ belong to process $P0$ and $T2$, $T3$ belong to $P1$. The dashed arcs show the ends of the process state intervals $s1$, $s2$ and $s3$. Thread state intervals are $t1$ to $t6$. State interval indices of $s2$ and $s3$ are (1,7) and (1,4) respectively. When $P0$ fails, it loses the state interval s2. It broadcasts a failure announcement containing the index (1,6), corresponding to the state interval index of $s1$. On receiving this announcement, thread $T2$ rolls back the orphan state interval $t5$. Thread $T3$ remains unaffected by this failure. Note that, if $P1$ were to fail instead of $P0$, and lose the state interval $s3$, then both $T0$ and $T1$ will detect that they are orphans due to the entry (1,4) in their dv's. This illustrates an important point of our protocol: in spite of belonging to the same process state interval and sharing a common index, thread intervals $t5$ and $t6$ act as independent rollback units and a single failure unit at the same time.

4.3. Comparative Evaluation

There are two factors of interest while comparing various protocols: false causality and dependency tracking overhead.

We have already discussed the false causality problem in Section 3.2.1. To summarize: the false causality problem arises in the process-centric approach because threads are forced to roll back together even when they have low interactions between them. False causality is particularly a problem for a large class of applications that have low interactions between threads. The observable effects of false causality are: (1) delayed output commits, and (2) unnecessary rollbacks after a failure. Both the thread-centric and balanced approaches avoid false causality by allowing threads to roll back independently.

The price paid for avoiding false causality is the higher dependency tracking overhead. This overhead is in three forms: space overhead, time overhead, and message size overhead. Table 1 summarizes the relative overheads of the various protocols. The overhead of checkpointing is common to all protocols and hence it is not shown in the table.

As discussed before, there have been two implementations of the process-centric approach: Slye & Elnozahy [14], and Goldberg et al. [7]. Slye & Elnozahy use a software counter to track the thread switches. Therefore, the space overhead consists of $O(s)$ space to log all thread switching information and $O(en)$ space to store dependency vectors for each receive event. The time overhead consists of the total extra time the recovery protocol requires to execute. This involves the time to save check-

	Space Overhead	**Time Overhead**	**Message Size Overhead**	**False Causality**
Process-centric(I)	$O(s + en)$	$O(c + en)$	$O(n)$	yes
Process-centric(II)	$O(o + en)$	$O(o + en)$	$O(n)$	yes
Thread-centric	$O(mn(o + e))$	$O(mn(o + e))$	$O(mn)$	no
Balanced	$O(n(o + e))$	$O(n(o + e))$	$O(n)$	no

Process-centric(I) is the process-centric protocol using Slye & Elnozahy [14]
Process-centric(II) is the process-centric protocol using Goldberg et al. [7]
n is the number of processes
m is the maximum number of threads and shared objects per process
e is the maximum number of message receive events per process execution
o is the maximum number of shared object accesses per process execution
s is the maximum number of thread switches per process execution
c is the time overhead for maintaining a software counter

Table 1. Comparative Evaluation of Overheads

points, log thread switching information, log dependency vectors. Therefore, the time overhead is proportional to the space overhead. The message size overhead is $O(n)$ since the dependency vector has n entries, one per process.

Goldberg et al. log the order of shared memory accesses so that they can be deterministically replayed. Therefore, the space overhead is $O(o + en)$ with the $O(o)$ component accounting for the log made on each shared memory access. The time overhead is proportional to space overhead. The message overhead remains $O(n)$ as before.

For the thread-centric approach, we assume that the overheads of checkpointing are similar for thread and process checkpoints. In practice, thread checkpoints may take additional time overhead to separate the thread local state from the process address space. Another method would be to simply take process checkpoints and extract the thread checkpoints when required. Since shared object accesses are treated as message receives, the space overhead to log the dependency vectors is $O(mn(o + e))$ since each vector has $O(mn)$ entries. The time overhead is similar. The message size overhead is now $O(mn)$.

The balanced approach reduces the vector size from $O(mn)$ in the thread-centric approach to $O(n)$. All overheads are similar to the thread-centric case replacing mn by n.

The saving in space, time and message size overhead of the balanced protocol with respect to the thread-centric protocol is substantial because mn is potentially a very large quantity compared to n. Each individual thread and shared object in the system is accounted for in mn. Since both protocols achieve the same elimination of false causality, the balanced protocol should always be preferred to the thread-centric protocol.

Compared to the process-centric protocol of Goldberg et al., the balanced protocol has the same message size overhead, but higher space and time overhead. This is because each shared object access in the balanced approach logs a $O(n)$ vector instead of constant information. With respect to the process-centric protocol using Slye & Elnozahy's technique, the space and time overhead is also expected to be higher since there are usually much fewer thread switches than shared memory accesses. However, as in Section 3.2.1, only the applications that have low thread interaction suffer greatly from false causality. For these applications, the increase in time and space overheads of balance protocol is low because the number of shared object accesses is low. Thus, the process-centric protocol should be used for applications with high thread interaction, and, therefore, low false causality effects. The balanced protocol should be used for the class of applications that have low thread interaction, where the extra space and time overhead is outweighed by the saving in false causality effects.

4.4. Implementation Issues

There are two new issues that arise when implementing optimistic recovery in multi-threaded systems as opposed to traditional systems: checkpointing threads, tracking causality through shared object accesses, and replaying threads independently.

Checkpointing threads is handled differently in different multi-threaded systems. In POSIX Threads, the thread state consists of the stack, register context (including program counter), thread local storage, and objects that are created on the heap by a thread. The global data is shared between all threads. This division may be simplistic, and it is possible for shared objects to exist in the stack of one process or on the heap. In such cases, a simplistic division would work correctly but would induce greater false causality. It is possible to track ownership of objects on the heap at the execution point when the object was created (e.g. by instrumenting at the *malloc* calls). It is important to note that the checkpoint can be made for the whole process as in existing schemes for multi-threaded systems [7]. This logically corresponds to all threads checkpointing at the same time. However, when required, the thread state must be extracted from the process checkpoint independently. In Java, object serialization allows the objects to be saved independently. The thread stacks are not exposed by the Java VM. In order to save thread stacks, existing schemes either modify the Java VM [12], or instrument the Java code [6].

An important issue in implementing shared objects is defining what exactly a shared object access is. This has impact on the number of accesses, o. As we have just seen, the smaller the value of o, the lower would be the overhead in the balanced protocol. Frequently, multi-threaded programs are written using a discipline that ensures that each shared object is accessed only through methods and all the methods are made mutually exclusive using an object lock. This is the model used by Java threads [8]. These mutually exclusive methods increase the granularity of a shared object access and provide a clear interface for tracking. Thus, instead of tracking every read and write to shared objects, we can now track each method invocation on a shared object.

Each method invocation on a shared object is treated as a message from the thread to the shared object and the return from the shared object is treated as a message from the shared object to the thread. Thus, the recovery manager must intervene at both these points. The log consists of the method and parameter values in case of the invocation and the returned values in the case of the return. While replaying a thread or a shared object, the logged values are replayed so that the thread or shared object recovers its state.

5. Final Note: Generalization

So far, we have considered the failure and rollback units in the context of processes and threads. We can select these units at even coarser granularities. For simplicity, let us consider a system consisting of non-threaded processes. Now, a process can be a rollback unit while a processor can be a failure unit. Compared to traditional optimistic protocols, this approach reduces the dependency tracking overhead while injecting false causality between processes. In case of a hardware crash, entire processor indeed crashes as a unit. In case of a software crash of a single process, all processes on the corresponding processor have to simulate a crash, and hence the false causality between them. Further generalizing, a local area network can act as a failure unit in a wide area environment. A process still acts as a rollback unit.

References

[1] L. Alvisi, and K. Marzullo. Deriving optimal checkpoint protocols for distributed shared memory architectures. *Selected Papers, International Workshop in Theory and Practice in Distributed Systems*, K. Birman, F. Mattern and A. Schiper editors, Springer-Verlag, 111-120, 1995.

[2] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1): 63-75, Feb. 1985.

[3] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously When Optimism Fails. In *Proc. IEEE Int. Conf. Distributed Comput. Syst.*, pages 108–115, 1996.

[4] O. P. Damani, A. Tarafdar and V. K. Garg. Optimistic Recovery in Multi-threaded Distributed Systems. Technical Report TR-PDS-1999-001, The Parallel and Distributed Systems Laboratory, The University of Texas at Austin (available via ftp or WWW at maple.ece.utexas.edu as technical report TR-PDS-1999-001).

[5] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Tech. Rep. No. CMU-CS-96-181, Dept. of Computer Science, Carnegie Mellon University (also available at ftp://ftp.cs.cmu.edu/user/mootaz/papers/S.ps), 1996.

[6] S. Funfrocken. Transparent Migration of Java-based Mobile Agents. *2nd Intl. Workshop on Mobile Agents*, Stuttgart, Germany, September 9 - 11, 1998.

[7] A. P. Goldberg, A. Gopal, K. Li, R. E. Strom, and D. F. Bacon. Transparent Recovery of Mach Applications. *1st USENIX Mach Workshop*, 1990.

[8] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison Wesley Longman, Inc, The Java Series, 1997.

[9] B. Lewis and D. J. Berg. *Threads Primer: A Guide to Multi-threaded Programming*. Sunsoft Press, 18-20, 1996.

[10] T. J. Leblanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. on Computers*, C-36(4):471-481, Apr. 1987.

[11] J. M. Mellor-Crummey and T. J. Leblanc. A Software Instruction Counter. *Proc. of the 3rd Symp. on Architectural Support for Programming Languages and Operating Systems*, 78-86, Apr. 1989.

[12] M. Ranganathan, A. Acharya, S. D. Sharma, and J. Saltz. Network-aware Mobile Programs. *Proc. of Usenix '97* Anaheim, CA, 1997.

[13] M. Russinovich and B. Cogswell. Replay for Concurrent Non-deterministic Shared-memory Applications. *Proc. ACM SIGPLAN Conf. on Programming Languages and Implementation (PLDI)*, 258-266, 1996.

[14] J. H. Slye and E. N. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. *Proc. 26th Fault Tolerant Computing Symposium*, 250-259, 1996.

[15] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Trans. on Computer Systems*, 204-226, August 1985.

[16] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. on Software Engineering*, 17(1):45-63, Jan. 1991.

[17] Y. M. Wang, O. P. Damani, and V. K. Garg. Distributed Recovery with K-Optimistic Logging. *Proc. 17th Intl. Conf. Dist. Comp. Sys.*, 60-67, 1997.