Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes*

Hong Va Leong Divyakant Agrawal
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Recovery from failures can be achieved through asynchronous checkpointing and optimistic message logging. These schemes have low overheads during failure-free operations. Central to these protocols is the determination of a maximal consistent global state, which is recoverable. Message semantics is not exploited in most existing recovery protocols to determine the recoverable state. We propose to identify messages that are not influential in a computation through message semantics. These messages can be logically removed from the computation without changing its meaning or result. We show that considering these messages in the recoverable state computation gives rise to recoverable states that dominate the recoverable state defined under conventional model. We then develop an algorithm for identifying these messages. This technique can also be applied to ensure a more timely commitment for output in a distributed computation.

Keywords: message semantics, commutativity, recovery, optimistic message logging, asynchronous checkpointing

1 Introduction

Fault-tolerance in distributed systems is an important issue. A system with only fail-stop failures [15] can be made fault tolerant by a checkpointing mechanism and a recovery mechanism. All other types of failures in a system can be translated to fail-stop failure with sufficient redundant resources [14]. A checkpoint is the system states of one or more processes stored on stable storage that survive processor crashes. Checkpointing provides information about non-trivial restarting points of computation after a failure has occurred whereas the recovery protocol manipulates the available information to determine a possibly maximal consistent system state, from which the computation can be resumed.

The simplest scheme is to checkpoint only consistent system states. Chandy and Lamport [5] proposed the marker algorithm that always collects a set of consistent system states. Koo and Toueg [9] used a two-phase scheme. In these *synchronous* schemes, recovery is merely the restoration of the most recent checkpoint. Synchronous checkpoints are also taken in distributed transaction processing systems [7, 17]. However, these schemes interfere with the underlying computation in general.

Asynchronous checkpoints can be taken independently to reduce the interference to the underlying computation [2, 4, 18, 8]. These checkpoints are no longer consistent and the recovery scheme must reconstruct a consistent state from this information after a failure. When a computation is forced to be discarded to maintain system consistency, it is said to be rolled back. The domino effect of cascading rollback can seriously damage the system performance. The protocol by Bhargava and Lian [2] rolls back processes only when necessary and does not suffer from the domino effect. However, the processes may be rolled back to the very beginning. To ensure progress in asynchronous checkpointing, the system needs additional information on the stable storage, such as the set of messages sent or received by a process. This is called *message logging* and is adopted in various recovery protocols [4, 18, 8].

Pessimistic message logging refers to the logging of a message before it is processed [4]. The advantage is the absence of cascading rollback. However, the waiting time for a message to be logged before processing is too long. Optimistic message logging allows messages to be processed independent of when they are logged. In the absence of failures, the only overhead is the asynchronous disk I/O by a background process. The drawback is that the recovery protocol becomes more complicated. Strom and Yemini [18] were the first to propose such a class of protocols, but their protocol suffers from *bounded* cascading rollbacks. Johnson and Zwaenepoel [8] developed a formal model and eliminated cascading rollback with a centralized protocol. Sistla and Welch [16] subsequently improved over [8] by imposing stronger assumptions and developed fully distributed protocols. Wang and Fuchs [19]

*This research is supported by the National Science Foundation under grant number IRI-9117094.

introduced the notion of *non-state* messages to reduce message logging. The recoverable state is limited to the most recent set of consistent checkpoints and message logging is used to recover lost messages. A non-state message will never become a lost message and hence needs not be logged.

Optimistic message logging schemes with asynchronous checkpointing suffer from rollback and incur a longer recovery period. We observe that certain messages involved in a distributed computation are irrelevant to the overall computation. Their existence does not change the meaning and result of the computation. By identifying such irrelevant messages and removing them logically, the message dependency is curtailed. Consequently the amount of rollback and recomputation after a failure is reduced. We can achieve comparable performance in the absence of failures, and faster recovery after a failure than Johnson and Zwaenepoel's protocol [8]. Since it is not necessary to log an irrelevant message in our scheme, this results in another small saving.

This paper is organized as follows. Section 2 gives a description of the model with some examples. In Section 3, the centralized recovery protocol is briefly described. We motivate our approach to improve the maximal consistent global state, based on the semantics of messages, in Section 4 with examples. In Section 5, we develop a protocol to compute the improved maximal recoverable state. Finally, we conclude our paper with a discussion on our protocol, decentralized recovery protocol, and related work on optimistic message logging and message semantics.

2 The Model

A distributed system comprises of a set of processes $\{P_1, P_2, ..., P_n\}$. Each process has a local state and performs computation based on the current state. Processes communicate by sending and receiving messages through a point-to-point communication network. An arriving message is buffered. When the receiver process is ready to receive a message and when all the bookkeeping tasks are completed, a buffered message is delivered. Delivery of a buffered message may be delayed to facilitate recovery (for example, log the message in pessimistic message logging), or to satisfy integrity constraints (for example, to ensure causal or atomic delivery [3]). Messages exchanged by the recovery protocol are not considered as part of the underlying computation.

The distributed computation model is based on that in [8]. Each process P_i maintains a count for the number of messages delivered to it. This count defines a *state interval* on P_i. In particular, the computation of P_i that occurs between the delivery of the k^{th} message and the $k+1^{st}$ message is called state interval σ_i^k and the value k is the *state interval index* of σ_i^k. The initial state interval is σ_i^0. The delivery of a message starts a new state interval on the message recipient. A message is *logged* if both its content and the index of the state interval that it starts have been saved on stable storage. The computation σ_i of P_i can be represented by a sequence of state intervals, $\sigma_i^0, \sigma_i^1, \sigma_i^2, ..., \sigma_i^l$. This computation $\sigma_i = \sigma_i^0 \cdot \sigma_i^1 \cdot \sigma_i^2 \cdot ... \cdot \sigma_i^l$, is the history of how the current state of P_i is reached and denotes also the current process state. Assuming that the local computation within a process is deterministic[1], it is always possible to reconstruct the current process state from the set of messages delivered. Occasionally a process takes a *checkpoint* and writes it to the stable storage. Each checkpoint belongs to a state interval σ_i^l. The *effective checkpoint* with respect to a state interval σ_i^k is the checkpoint with state interval σ_i^l and $l \leq k$ but there is no other checkpoint with state interval $\sigma_i^{l'}$ such that $l < l' \leq k$. A state interval σ_i^k is said to be *stable* if all the messages that start state intervals $\sigma_i^{l'}$ between σ_i^k and the effective checkpoint with respect to σ_i^k have been logged.

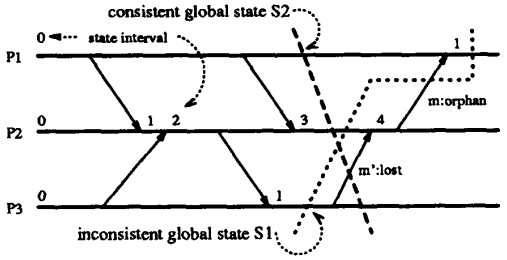
Processes interact through messages, which induce a *dependency* relation among processes. The message that starts the state interval σ_i^k on P_i has been sent by some other process P_j in state interval σ_j^l. We say that the state interval σ_i^k *directly depends* on the state interval σ_j^l. Also σ_i^k directly depends on all state intervals that $\sigma_i^{k'}(k' < k)$ directly depends upon. Define the *dependency vector* of σ_i^k to be $D_i^k = \langle d_1, d_2, ..., d_n \rangle$ such that for all j, σ_i^k directly depends on $\sigma_j^{d_j}$. Every state interval directly depends on itself and $d_i = k$. The dependency vector is an array of state interval indices and does not capture the notion of transitive dependency. The reason for not capturing the transitive dependency is to reduce the failure-free overhead (to be discussed in Section 3).

A *global system state* (or global state) S is a collection of process states σ_i, one from each process. In other words, $S = \langle \sigma_1, \sigma_2, ..., \sigma_n \rangle$. Each process state σ_i corresponds to a state interval $\sigma_i^{\delta_i}$, such that $\sigma_i = \sigma_i^0 \cdot \sigma_i^1 \cdot \sigma_i^2 \cdot ... \cdot \sigma_i^{\delta_i - 1} \cdot \sigma_i^{\delta_i}$. We abstract out the state interval indices to represent the global state S by $\langle \delta_1, \delta_2, ..., \delta_n \rangle$. Several global states may have the same state interval index and would be considered the same global state in this model. A global state S *dominates* another global state S' if the state of each process in S is the same as or more advanced than the state of the same process in S'. A global state S is *consistent* if and only if every message delivered in S has been sent in S. In terms of state in-

[1] Non-deterministic operations are still allowed, as long as the non-deterministic response of an operation is determined deterministically using only the process state, and non-temporal information such as the process identifier.

tervals, every message delivered must have been sent, or will be sent deterministically. The consistency of a global state $S = \langle \delta_1, \delta_2, ..., \delta_n \rangle$ can be determined by means of a *dependency matrix*. The dependency matrix D^S for S is generated by combining the corresponding dependency vectors row-wise from each process. Thus $D^S = (D_1^{\delta_1}, D_2^{\delta_2}, ..., D_n^{\delta_n})^T$, with each dependency vector forming a *row* of the matrix. A global state S is consistent if and only if for all i and j, $D^S[i,i] \geq D^S[j,i]$ [8]. In other words, a global state is consistent when all state intervals in the global state do not depend on any state interval not contained in the global state. A consistent global state S is *recoverable* if all state intervals $\sigma_i^{\delta_i}$ are stable. A recoverable global state (or recoverable state) is *always* reconstructible from information on the stable storage.

A message m is *orphan* in a global state S if it has been delivered but has not yet been sent (and will not be sent) in the current state interval of the sender. A process P_i is an *orphan process* in S if it has an orphan message delivered to it. More formally, P_i is orphan if its current state interval $\sigma_i^{\delta_i}$ depends on any of the state interval beyond S. No process is orphan in a consistent global state. A message m is a *lost message* in S if it has been sent in S but has not yet been delivered in S. Lost messages do not affect the consistency of a global state.



$$D^{S_1} = \begin{bmatrix} 1 & 4 & \perp \\ 0 & 3 & 0 \\ \perp & 2 & 1 \end{bmatrix} \qquad D^{S_2} = \begin{bmatrix} 0 & \perp & \perp \\ 0 & 3 & 0 \\ \perp & 2 & 1 \end{bmatrix}$$

Figure 1: Global State and Dependency Matrix

Figure 1 shows a computation with three processes. In S_1, $D^{S_1}[1,1] = 1$ since P_1 has only one message delivered. Also $D^{S_1}[1,2] = 4$ because P_1 has received a message from P_2 sent in state interval 4 of P_2. $D^{S_1}[1,3] = \perp$, since P_1 does not get any message from P_3. The message m is an orphan in S_1 since it has been delivered but not yet been sent. Since $D^{S_1}[1,2] > D^{S_1}[2,2]$, S_1 is inconsistent. A consistent global state S_2 is represented by the dependency matrix D^{S_2} where all diagonal elements dominate the columns. The message m' is a lost message but is not reflected in the dependency matrix.

The goal of a recovery scheme is to determine and construct a consistent global state from the stable storage that survives a failure. The computation then resumes from the recovered consistent global state. For any checkpointing and recovery protocol, it is required that such a state always exists and the maximal consistent global state reconstructible is monotonically increasing.

3 A Centralized Recovery Protocol

The centralized optimistic message logging and recovery protocol is presented in [8] and is summarized here. Messages delivered are logged asynchronously with respect to the processing of these messages. Checkpoints are taken independently and occasionally. As messages are logged and checkpoints taken, new state intervals become stable. A special coordinator process P_c is chosen to carry out the recovery related computation. Whenever a state interval σ_i^k becomes stable on P_i, its index and dependency vector are transmitted to P_c, which will incorporate σ_i^k into the current recoverable state. After a failure, all messages delivered to non-failed processes are logged and new stable state intervals incorporated to form the maximal recoverable state. All failed processes then execute the recovery protocol. Processes that are not orphans can immediately restart their computation. Orphan processes are forced to fail and execute the recovery protocol. The recovery protocol on P_i restores the effective checkpoint with respect to the stable state interval $\sigma_i^{\delta_i}$ contained in the maximal recoverable state. All logged messages are then replayed from the log, as if they were received and delivered normally, until the state interval $\sigma_i^{\delta_i}$ is reached. All the other logged messages are then discarded and normal processing resumes. Duplicate messages are identified (using unique message identifier) during the recovery stage and discarded. Lost messages are handled by the application program. The protocol uses only information on the stable storage. It is thus restartable and can tolerate multiple failures.

The only overhead during normal processing is the tagging of extra information in each message sent and the transmission of new stable intervals to P_c. When a message is sent, it is tagged with the state interval index $D_j[j]$ of the sender P_j. This value tracks the direct dependency of messages. If we do not use direct dependency, but use transitive dependency instead, we will have to transmit a vector of size n to maintain valid transitive information. The lower bound for tracking transitive information is proved to be a vector of size n [6]. Transitive dependency is computed only when a state interval becomes stable and the maximal recoverable state is updated. The computation is carried out by P_c in an incremental manner.

4 Reducing Rollback Computation Due to Orphans

Recovery protocols that take checkpoints and log messages asynchronously suffer from excessive rollback after a failure. In particular, orphan processes are forced to fail and rolled back to the maximal recoverable state. This often results in a significant rollback, even when only a single message is an orphan. Those orphan processes should be allowed to continue to execute, as long as the orphan messages are irrelevant with respect to the overall computation.

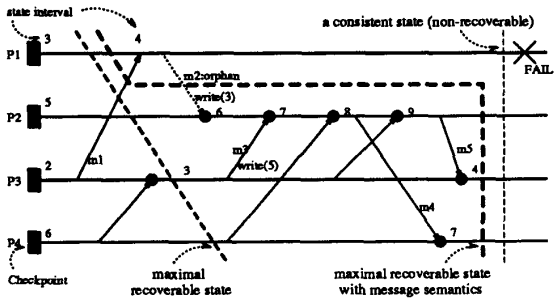


Figure 2: Reducing Orphan Rollback: Example 1

Consider Figure 2, where a rectangular block denotes a checkpoint and a circle denotes a logged message. Assume that P_1 is slow in logging messages, and m_1 has not yet been logged when P_1 fails. The maximal recoverable state is $RS = \langle 3, 5, 3, 6 \rangle$. Now m_2 is an orphan message and P_2 becomes an orphan process. Even worse, as P_2 becomes orphan, m_4 and m_5 also become orphan. As a result, P_3 and P_4 become orphan processes. This mishap is manifested by a cascading rollback in Strom and Yemini's protocol [18]. In Johnson and Zwaenepoel's protocol [8], the recoverable state remains unchanged, even as more and more state intervals in P_2, P_3 and P_4 become stable. But this can be improved. Suppose P_2 is a shared memory object accepting *read* and *write* messages. When we consider the semantics of the *write* messages m_2 and m_3, we discover that m_2 is irrelevant with respect to the overall computation, since the state of P_2 is overwritten when m_3 is delivered. Whether m_2 exists in the computation makes no difference to the computation and hence its orphanhood does not matter. Dropping m_2 out of the system, the consistent global state recoverable after the failure will be $RS' = \langle 3, 9, 4, 7 \rangle$. Thus all the three processes P_2, P_3 and P_4 are no longer orphans and rollback is not necessary.

Figure 3 illustrates another example with three processes, where P_2 is also a shared memory object. A similar observation can be made to the *read* message. The state that can be recovered is $RS = \langle 3, 5, 2 \rangle$, with traditional recovery protocols. However, we notice that m_2 causes no state change in P_2 and the reply message m_3 for request m_2 is sent to P_1. Since P_1

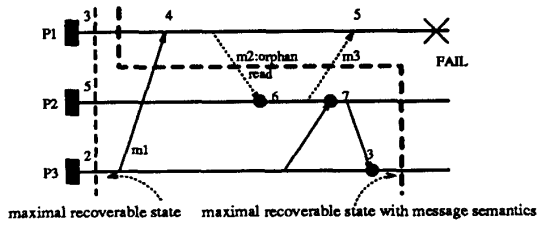


Figure 3: Reducing Orphan Rollback: Example 2

has failed, the reply message will be discarded. We therefore say that m_2 is irrelevant in the computation. Removing m_2 leads to a better recoverable state $RS' = \langle 3, 7, 3 \rangle$.

Let us define *explicit state change* to be the change in the content and value of the object/process and *implicit state change* to be the obligation to send out response messages, as a result of the processed message. A message m from P_j to P_i is *insignificant* if any one of the following conditions holds,

1. m is a lost message, or
2. m causes no explicit and implicit state change, or
3. m does not cause any explicit state change in P_i and m generates only insignificant messages, or
4. m causes an explicit state change in P_i but the explicit state change is overwritten by the next message m' to P_i and m generates only insignificant messages, or
5. m causes an explicit state change in P_i but the state of P_i is overwritten by some future message m'; furthermore, m generates only insignificant messages and all messages delivered to P_i between m and m' are insignificant.

Any message that is not insignificant is *significant*. The first condition is trivial, since lost messages do not affect the recoverable state. They may, however, enable some other messages to become insignificant, due to the other conditions. The second condition says that a "useless" message (a message causing no explicit nor implicit state change) is insignificant. The third condition captures the semantics of *read* messages and messages with similar property. The last two conditions capture the overwriting property of messages, such as *write* messages, in a direct and an indirect manner. In all conditions, P_i cannot send any significant messages, since those messages may introduce transitive dependency for the receivers. Define the *dependents* of a message m to be the set of messages sent out in the state interval that m starts. Call the dependents that have been delivered in a global state the *effective dependents* with respect to that state. We may express the above conditions in terms of dependents.

5 A Recovery Protocol with Semantics

We develop a protocol to realize the notion of insignificant messages in this section. Two issues should be resolved. The first issue is an efficient way to identify insignificant messages. The recursive definition of insignificant messages is too costly for classification. The second issue is to handle the changed dependency when such messages are removed since a gap in the state interval index, which counts the number of messages delivered, is left after a message is removed.

P_i maintains the dependency vector $D_i = (d_{i1}, d_{i2}, \ldots, d_{in})$ and logs delivered messages asynchronously.
Associated variables:
  depvec is dependency vector of messages
  in stores message tag, used when a message becomes insignificant
  type is possible message semantics (*read/write/other*)
  class is message classification (*tsig/insig/tinsig/unknown*) where *tsig* and *tinsig* are temporary classifications
  ctr counts number of messages sent out to other processes
When a message m is sent to P_j:
  tag m with the current state interval index d_{ii}
  send m to P_j; $ctr[d_{ii}] \leftarrow ctr[d_{ii}] + 1$
When a message m from P_j with index d is delivered:
begin
  $d_{ii} \leftarrow d_{ii} + 1$; $d_{ij} \leftarrow \max(d_{ij}, d)$
  $depvec[d_{ii}] \leftarrow D_i$; $in[d_{ii}] \leftarrow d$; $ctr[d_{ii}] \leftarrow 0$
  $class[d_{ii}] \leftarrow unknown$
  if m is state-overwriting **then** $type[d_{ii}] \leftarrow write$
  elseif m is non-state-modifying **then** $type[d_{ii}] \leftarrow read$
  else $type[d_{ii}] \leftarrow other$
  /* try to classify previous messages (only the simple cases) */
  if $type[d_{ii} - 1] = read$ and $ctr[d_{ii} - 1] = 0$ **then** (**)
    $class[d_{ii} - 1] \leftarrow insig$ (**)
  if $type[d_{ii}] = write$ **then** (**)
    $t \leftarrow d_{ii} - 1$ (**)
    /* do for previous intervals with no message sent */ (**)
    while $ctr[t] = 0$ **do** $class[t] \leftarrow insig$; $t \leftarrow t - 1$ (**)
  endif (**)
end
When state interval σ_i^k becomes stable:
  send the dependency vector to the coordinator

Figure 4: The Recovery Protocol: Normal Operation

Our recovery protocol is similar to [8]. The differences are the maintenance of extra information to facilitate the detection of insignificant messages and the logical removal of these messages. The complete recovery protocol is depicted in Figures 4, 5 and 6. The procedure *find_insig*(m) (Figure 7) is defined to determine whether a message m is insignificant. It applies the definition of an insignificant message. To improve efficiency, we perform some pre-processing before invoking this procedure. First, messages are classified online for simple cases. Second, a topological sort is performed on the ordering of messages to be classified.

It is easy to classify whether a message is insignificant if it has no dependent. This is handled by the receiver process when a new message is delivered. If the previous message is "read-only" and no message has been sent out, then it is insignificant. Similarly,

---

After a failure has occurred:
begin
  log all delivered messages onto stable storage
  identify the stable state intervals and send to the coordinator
  send information about insignificant messages to coordinator
end
The recovery protocol:
begin
  get the i^{th} component δ_i of the recoverable state from P_c
  restore the effective checkpoint with respect to δ_i
  switch mode to "recovering"
  get messages from stable storage log and perform computation
  when state interval reaches δ_i, discard logged messages, and
    set mode to "normal"
end

Figure 5: The Recovery Protocol: Failure Recovery

if the current message is a *write* message, all previous consecutive messages that have no dependent are insignificant. The relevant code can be found embedded in Figure 4 marked with (**). Messages that are insignificant conditional on whether their dependents are insignificant are not classified by the local process. Instead, this is done by the coordinator. An advantage of the centralized scheme is that it is not necessary to send the classification information among processes following the message dependency order. When the state interval for a message is current, we cannot determine whether it is significant, without knowing the future. We therefore introduce the status *tsig* (temporarily significant) and *tinsig* (temporarily insignificant) for it. With a non-terminating computation, it is not possible to conclude that a message is significant, due to the recursive nature of the definition. If existing evidence shows that a message is significant, unless proved otherwise in the future, we will say that it is *tsig*. In the same vein, a message is classified as *tinsig* if it is currently insignificant, but the classification is subject to revocation in the future. We would like to distinguish *tinsig* messages from *insig* (insignificant) messages, since the latter category is *stable*: once a message becomes *insig*, it will remain so forever. Lost messages are classified as *tinsig*, since they do not have any effect when the failure occurs. A message classified as *insig* cannot have caused a lost message to be sent in the state interval it creates.

Consider Figure 7, where m_{10} is a lost message. Several messages have been classified by local processes as simple cases. Messages m_9 and m_1 are *insig*, since they are before a *write* and do not cause any message to be sent. Indirectly, m_7 is *insig*. All the other seven messages are still unclassified. We start by finding a topological sort on the reversed dependency of the messages. It is clear that m_3 depends on m_6, and m_1 depends on m_3. Furthermore, the program order on the delivery events dictates that m_7 *happens-before* m_8 [10]. We reverse the direction of the dependency to form a prece-

```
Coordinator P_c maintains the current recoverable state RS,
    the dependency matrix D^RS, and a set of stable state intervals
    and their dependency vectors not yet contained in RS.
It identifies insignificant message dependency and computes new state interval.
Normal operation - when stable state interval σ_i^k is received:
    try to include stable state interval into the current recoverable
        state RS to form a better one, as in [8]
After a failure:
begin
    receive new stable state intervals from all non-failed processes,
        and information about insignificant messages
    form a recoverable state from available stable state intervals
    determine reversed message dependency on all messages beyond
        the current recoverable state
    perform a topological sort on the above messages
    foreach message m in topological ordering do
        call find_insig(m) to classify m
    foreach insig message m' in temporal order do
        let m' start the state interval σ_i^{k_i} on P_i, sent from P_j
        /* reconstruct dependency vectors of affected messages */
        depvec[k_i][j] ← depvec[k_i − 1][j]
        propagate the changed dependency forward
        recompute new dependency for any message sent from P_j,
            using array in and the previous dependency vector
    endfor
    if monotonicity is not required then
        repeat the above step for tinsig messages
    compute the maximal recoverable state from the stable state
        intervals and the revised dependency
    reset tsig and tinsig messages back to unknown
end
```

Figure 6: The Recovery Protocol: Coordinator

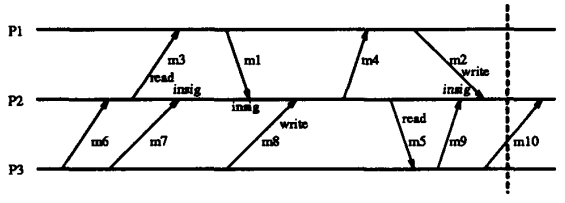


Figure 7: Classifying Messages: An Example

dence relation used in the topological sort: $m_{10} \rightarrow m_5, m_9 \rightarrow m_5, m_2 \rightarrow m_4, \ldots$ for reversed message dependency as well as $m_{10} \rightarrow m_9, \ldots$ for reversed program order. To allow an iterative implementation, the classification should be carried out in the reversed order of dependency. A topological sort yields $\{m_{10}, m_2, m_9, m_5, m_4, m_8, m_1, m_7, m_3, m_6\}$. Message m_{10} is lost, so it becomes *tinsig*. We can classify m_2 to be *tsig*, since there is no evidence that it is insignificant (line (5) of Figure 8). We already know that m_9 is *insig*. Since m_5 is a *read* message and all its dependents (m_9 and m_{10}) are *insig* and *tinsig* messages, it can be classified as *tinsig* (line (3)). Since m_4 has a dependent m_2 that is *tsig*, it is also classified as *tsig* by line (1). For the same reason, m_8 is *tsig*. Also m_1 and m_7 are known to be *insig* by the local process. All the dependents of the *read* message m_3 are *insig*. It is classified as *insig* by line (2). Finally, m_6 is *insig* by line (4), since all messages between it and

```
Procedure find_insig(m)
begin
    let m be sent from P_j and start state interval σ_i^{k_i} on P_i
    if class[k_i] ≠ unknown then return class[k_i]
    let num_dep be the number of effective dependents of m
    if any dependent of m is tsig then class[k_i] ← tsig                (1)
    elseif type[k_i] = read or type[k_i + 1] = write then
        if all dependents are insig and σ_i^{k_i} is not current then
            class[k_i] ← insig                                          (2)
        elseif all effective dependents are insig or tinsig then
            class[k_i] ← tinsig                                         (3)
    elseif there exists k > k_i, type[k] = write then
        foreach m' for state interval k' such that k_i < k' < k do
            test the class of m'
        if all of them and all their dependents are insig then
            class[k_i] ← insig                                          (4)
        elseif they and their effective dependents are insig or tinsig
            then class[k_i] ← tinsig
    elseif num_dep = 0 then /* no effective dependent */
        class[k_i] ← tinsig /* assume insignificant */
    endif
    if class[k_i] = unknown then /* not yet classified */
        class[k_i] ← tsig /* assume significant */                     (5)
    return class[k_i]
end
```

Figure 8: The Procedure *find_insig*()

the next *write* are *insig* and all their dependents are *insig*. Note that with several *write* messages, it is possible to generate many insignificant messages. A temporary status is assigned to messages that cannot be identified for sure.

After a failure, the coordinator obtains the necessary information from all other processes, computes the best possible recoverable state, and then identifies insignificant messages beyond this state. Some of the messages have been classified locally when they are delivered. The remaining unclassified messages are grouped together. The reversed dependency is then generated and a topological sort is carried out. Each message m is classified by a call to the procedure *find_insig*(m) in this order. Finally the insignificant messages are removed from the computation. To remove a message logically, we need to reset the dependency of the process on that message, as if the message had never been delivered. This can be achieved with the array *in*, where $in[k]$ contains the tag of the message which starts state interval σ_i^k on P_i, as well as the dependency vectors *depvec* on messages. The dependency of an insignificant message can be replaced by the dependency of the previous state interval. Suppose at P_i, the message m from P_j is to be removed. Let m start state interval $\sigma_i^{k_i}$ on P_i. The j^{th} entry of the dependency vector associated with m can be restored, by taking the value of the j^{th} component of the dependency vector of the previous message. This value is propagated forward for all delivered messages on P_i not sent by P_j, by changing the j^{th} component of their dependency vectors with that same value. If another message m' from P_j is encountered, this j^{th}

entry of the dependency vector is taken to be the maximum of the *in* array entry for m' and the j^{th} entry of the dependency vector of the previous message. The maximum is taken, as the dependency carried by this m' must be taken into account. Of course, if this m' is also going to be removed, it suffices to restore the previous value. After adjusting the dependency vectors due to insignificant messages, a new maximal recoverable state is computed. Finally, orphan processes with respect to this new recoverable state are rolled back, as in traditional recovery protocols.

We treat lost messages to be insignificant (*tinsig*) to improve the recoverable state. However, this definition results in a non-prefix-closed property. In particular, a non-state-modifying message that has only caused lost messages to be sent is insignificant. When a "lost" message is finally delivered, the message becomes significant. The recoverable state that is computed by assuming it to be insignificant may become invalid. Hence, the sequence of recoverable states generated is no longer monotonic. We can avoid this problem of non-monotonicity by being conservative: change the first condition in the definition of insignificant messages by assuming a lost message to be significant (*tsig*), unless the recipient of the lost message has failed. This can also be solved by considering all *tinsig* messages to be significant. Owing to the stability of an *insig* message, the recoverable states determined using only insignificant messages will continue to be valid and form a monotonic sequence.

6 Discussion

In this paper, we observe that messages irrelevant to the underlying computation can be removed logically, resulting in a better recovery behavior in a recovery protocol. We extend the basic [8] protocol to encompass the detection of insignificant messages and show how to use this information to improve the recoverable global state, thus reducing the amount of rollback required after a failure. The protocol is centralized.

The issue of exploiting message semantics were discussed in [13]. The authors considered reducing the checkpoint dependency based on whether a remote procedure call is state-modifying. They forced a checkpoint to be taken when dependency exists to avoid cascading rollback. They also discussed the dependency developed at a shared server process [1] (among several groups of independent processes with their own recovery protocol) and restricted the rollback effect by propagating checkpoint requests. Their approach can be considered as *semi-synchronous* checkpointing. During normal operation, checkpointing is asynchronous. When possible harmful dependency is developed, a checkpoint is forced to be taken on demand. This would increase the failure-free overhead. To avoid harmful dependencies, artificial delaying of message delivery is adopted in [20]. Our approach, based on the formal model of [8], for insignificant message detection is completely asynchronous. We also do not have to roll back processes whose current state interval is part of the maximal recoverable state, whereas [19] has to, since their notion of recoverable state is at the checkpoint level but not the state interval level. Inspired by [1], we examine server processes and apply richer semantics such as commutativity, as in [12] on the operations they support while retaining asynchronous checkpointing and asynchronous message logging schemes. Relevant results on server processes are reported in [11].

The protocols in Sections 3 and 5 are centralized. To improve fault tolerance, the protocols can be distributed. We may have the coordinator send the current recoverable state to other processes periodically so that they can become the new coordinator when the coordinator fails. Processes can also take turns in becoming the coordinator for better workload distribution and network traffic patterns. In a fully decentralized approach, each process maintains the same information as the coordinator and performs the computation of the recoverable state independently. This will allow completely independent recovery. Further issues for decentralization are discussed in [11].

Fully distributed recovery protocols are often costly and represent a tradeoff between fault-tolerance and efficiency. Sistla and Welch [16] developed distributed recovery protocols based on a more restrictive system model. In particular, they assumed that the message delivery mechanism is reliable and ensures FIFO ordering. Their recovery protocol (with transitive dependency) is a replicated version of a centralized protocol, since each process performs exactly the same computation to obtain the same recoverable state. The number of messages exchanged increases from $2n$ to $O(n^2)$, when going from centralized towards decentralized computation. Furthermore, a redundant n-folded computation is made. Without transitive dependency information appended to messages, $O(n^3)$ messages need to be sent, whereas $O(n^2)$ messages suffice in a centralized version. The number of messages sent by the [18] protocol is dependent on the execution status but in the worst case, it is exponential: $O(2^n)$. In the [8] protocol, this exponential number of messages caused by cascading rollback manifests itself as a lengthy computation inside the main loop in the computation of the recoverable state by the coordinator. The number of messages sent, however, is just $O(n)$. The coordinator must obtain information about stable state intervals for recovery state computation. Thus processes should send this information periodically. This information is also needed to commit an output. Our protocol is based on [8] and has the same message complexity. In the worst possible scenario, it is within a factor of 2 with respect to [8], since a sep-

arate message may need to be sent concerning with the fresh status of each message. A fully decentralized version of our protocol will result in an n-folded computation. More performance results are discussed in [11].

The identification of insignificant messages leads to new recoverable states that dominate the recoverable state under conventional definition. The same technique can be applied to commit output more timely, by advancing the recoverable state (recovery frontier) more rapidly, since output commitment can only be made after the send event of the output message is included in the recoverable state. There is a higher failure-free overhead because it is necessary to perform the computations more often. However, the same tradeoff has to be made in all optimistic message logging schemes, including [16, 8], to commit output.

Traditional definitions of consistent global state and recoverable global state focus only on the dependency of messages and fail to capture properties of the underlying messages. Our approach suggests a way to improve the effective recoverable global state, whereas still remaining in the domain of a "consistent" global state, by removing messages that have no impact on the overall computation.

References

[1] M. Ahamad and L. Lin. Using checkpoints to localize the effects of faults in distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 2–11. IEEE, 1989.

[2] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems—an optimistic approach. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems*, pages 3–12, October 1988.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 90–99, October 1983.

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

[6] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letter*, 39(1):11–16, July 1991.

[7] M.J. Fischer, N.D. Griffeth, and N.A. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982.

[8] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems. *Journal of Algorithms*, 11(3):462–491, September 1990. Preliminary version in PODC 1988.

[9] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

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

[11] H.V. Leong and D. Agrawal. Semantics-based failure recovery in distributed systems with optimistic message logging. Technical Report TRCS94-06, Department of Computer Science, University of California at Santa Barbara, 1994.

[12] H.V. Leong, D. Agrawal, and J.R. Agre. Using message semantics to reduce rollback in the time warp mechanism. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 309–323. LNCS, September 1993.

[13] L. Lin and M. Ahamad. Checkpointing and rollback-recovery in distributed object based systems. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 97–104. IEEE, 1990.

[14] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[15] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[16] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[17] S.H. Son and A.K. Agrawala. Distributed checkpointing for globally consistent states of databases. *IEEE Transactions on Software Engineering*, 15(10):1157–1167, October 1989.

[18] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):205–226, August 1985.

[19] Y. Wang and W.K. Fuchs. Optimistic message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 147–154. IEEE, 1992.

[20] Y. Wang and W.K. Fuchs. Scheduling message processing for reducing rollback propagation. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 204–211. IEEE, 1992.