Semantics-based Failure Recovery in Distributed Systems with Optimistic Message Logging*

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

Abstract

Recovery from failures is important in distributed computing. A common technique to support recovery is asynchronous checkpointing, coupled with optimistic message logging. These schemes have low overheads during failure-free operations and can provide an acceptable degree of fault-tolerance. 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 the computation through message semantics. These messages can be logically removed from the computation without changing its meaning or result. In this paper, we illustrate with examples how the removal of these messages improves the theoretical maximal consistent global state. Taking semantics into account, recovery protocols are then developed to realize the idea. The semantics in object-oriented databases is adapted to special processes acting as servers for further improvements. 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

*An abridged version of this paper appears in the *Proceedings of the 14th International Conference on Distributed Computing Systems* as Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes.

†This research is supported in part by the NSF under grant number IRI-9117094.

1 Introduction

As distributed computing becomes ubiquitous, the issue of fault-tolerance of distributed computation becomes prominent. Owing to the difference in the failure modes between a centralized and a distributed computer system, it is much more difficult to achieve fault-tolerance in a distributed system. Usually the components in a distributed system fail in an independent manner. This complicates the strategies to ensure consistency and integrity of the system when a failure occurs. The fault-tolerance in a computer system that fails by stopping [SS83] can be achieved by a checkpointing mechanism and a recovery mechanism. Neiger and Toueg [NT90] showed that all other types of failures can be translated to fail-stop failure with a sufficient amount of redundant resources. A checkpoint is the system states of one or more processes that are stored on stable storage that can 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 recovery scheme is to checkpoint only consistent system states. Chandy and Lamport [CL85] proposed the marker algorithm that always collects a set of consistent system states. Koo and Toueg [KT87] 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 for fault-tolerance [FGL82]. Unfortunately, taking a checkpoint synchronously essentially halts the underlying computation for an extended period and consumes significant communication bandwidth. This is in general very undesirable. In a database environment, a synchronous checkpoint that has little interference on the normal transactions is achievable [SA89] due to the discrete nature of transactions.

Asynchronous checkpoints can be taken independently to reduce the interference to the underlying computation in a synchronous checkpoint [BL88, BBG83, SY85, JZ90]. 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 [BL88] rolls back processes only when necessary and does not suffer from the domino effect. It is, however, possible to get rolled back to the very beginning. To ensure progress in asynchronous checkpointing, the system would maintain 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 [BBG83, SY85, JZ90].

Pessimistic message logging refers to the logging of a message received before being processed [BBG83]. The advantage of this scheme is the absence of cascading rollback, since there always exists a *non-decreasing* consistent system state constructible from the most recent checkpoints and the logged messages. However, the waiting time for a message to be logged before processing can be significant, compared with the computation. This is unacceptable in an environment where a large number of messages are exchanged. Optimistic message logging allows messages to be processed independent of when they are logged, in an asynchronous manner. In the absence of failure, the only overhead is the asynchronous logging of messages by a background process. The catch is that the recovery protocol becomes more complicated. Strom and Yemini [SY85] were the first to propose such a class of protocols, but their protocol suffers from cascading rollbacks, the number of which could be exponential in the number of processes. Johnson and Zwaenepoel [JZ90] developed a formal model and eliminated cascading rollback with a centralized protocol. Sistla and Welch [SW89] subsequently improved over [JZ90] by imposing stronger assumptions and developed fully distributed protocols. Reducing message logging improves the performance of an optimistic scheme. Wang and Fuchs [WF92a] therefore introduced the notion of *non-state* messages. 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 in a possible recoverable state and hence needs not be logged for proper recovery after a failure.

Optimistic message logging schemes with asynchronous checkpointing suffer from rollback and incur a longer recovery period due to the loss of useful forward computation. We observe that certain messages involved in a distributed computation are irrelevant to the overall computation (according to their semantics). Their existence does not change the meaning and result of the computation. For example, if a $write(x, 3)$ message is followed immediately by a $write(x, 5)$ message, the effect of the first message will be invisible to the computation. By identifying such irrelevant messages (referred to as *insignificant messages*) using message semantics and removing them logically, the dependency among processes imposed by messages is trimmed. Consequently the amount of rollback and recomputation after a failure is reduced. Thus we can achieve comparable performance in the absence of failure, and faster recovery after a failure than Johnson and Zwaenepoel's protocol [JZ90]. It is not necessary to log an insignificant message in our scheme, resulting in a small saving in computation. 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. Server processes as special cases are investigated in Section 6. Richer semantics such as commutativity are exploited for further improvement. Decentralized and distributed implementation issues are explored in Section 7. Performance of different protocols is discussed in Section 8. Finally, we conclude our paper with a discussion on our protocol and related work on optimistic message logging and message semantics.

2 The Model

A distributed system comprises of a set of n 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 and we assume a point-to-point communication at the abstract level. 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 delivery or atomic delivery [BSS91]). Messages sent and received by the recovery protocol are not considered as a part of the underlying computation.

The distributed computation is modeled after [JZ90]. 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, including the delivery event of the k^{th} 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 and terminates the current state interval on the message recipient. A state interval σ_i^k is *active* if the $k+1^{st}$ message has not yet been delivered. State interval σ_i^k is *safe* if the message that starts σ_i^k has been logged onto stable storage. 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. The final state interval σ_i^l is always active, since the $l+1^{st}$ message is not yet delivered. Assuming that the local computation within a process is deterministic[1], it is then always possible to reconstruct the current process state from the set of messages delivered. Occasionally a process takes a

[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 other non-temporal information such as the process identifier.

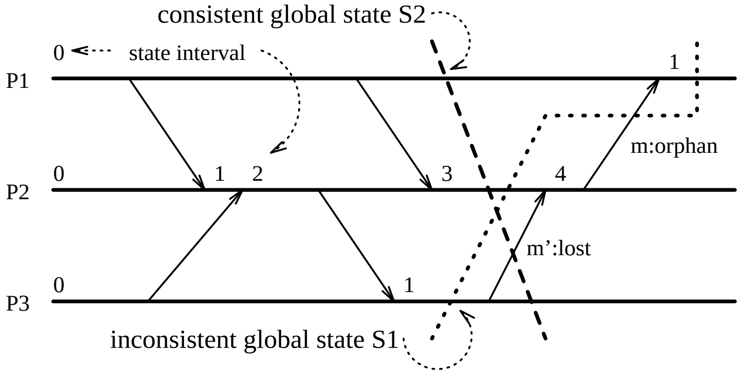
checkpoint that is a snapshot of its local state and writes it to the stable storage. Each checkpoint s_i belongs to a state interval σ_i^l. The *effective checkpoint* with respect to a state interval σ_i^k is the checkpoint with state interval σ_i^l and $l \leq k$ but there is no other checkpoint with state interval $\sigma_i^{l'}$ such that $l < l' \leq k$. A state interval σ_i^k is said to be *stable* if all state intervals $\sigma_i^{l'}$ between the effective checkpoint with respect to σ_i^k and σ_i^k (including σ_i^k) itself are safe.

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. For example, P_j at σ_j^2 sends m_1 to P_i, starting σ_i^8 and P_j at σ_j^3 sends m_2 to P_i, starting σ_i^6. Then σ_i^8 depends on both σ_j^3 as well as σ_j^2. This second part of the requirement takes into account non-FIFO messages (which is not allowed in the model of [SW89]). 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. If $P_{j'}$ does not send a message directly to P_i, but only to P_j which sends a message to P_i as a result of the message from $P_{j'}$, then we say σ_i depends on σ_j and σ_j depends on $\sigma_{j'}$ but do *not* say σ_i depends on $\sigma_{j'}$. 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 $\sigma_i = \sigma_i^0 \cdot \sigma_i^1 \cdot \sigma_i^2 \cdot ... \cdot \sigma_i^{\delta_i - 1} \cdot \sigma_i^{\delta_i}$, and $\sigma_i^{\delta_i}$ is active. We would 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 indices and would be considered the same global state. A global state S is said to *dominate* another global state S' if the state interval for each process in S is at least as advanced as the state interval for the same process in S'. In other words, the state $S = \langle\delta_1, \delta_2, ..., \delta_n\rangle$ dominates $S' = \langle\delta'_1, \delta'_2, ..., \delta'_n\rangle$ if for each i, $\delta_i \geq \delta'_i$. A global state S is *consistent* if and only if every message delivered in S has been sent in S. In terms of state intervals, 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 $D^S = \langle D_1^{\delta_1}, D_2^{\delta_2}, ..., D_n^{\delta_n}\rangle^T$, 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, $D^S[i,i] \geq D^S[j,i]$ [JZ90]. 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* with respect to a global state S if it has been delivered in the recipient 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* with respect to S if it has an orphan message delivered to it. More formally, P_i is orphan if its current state interval $\sigma_i^{\delta_i}$ is dependent upon any of the state interval beyond S. No process is orphan with respect to a consistent global state. A message m with respect to S is a *lost message* 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.

Figure 1 shows a computation with three processes. With respect to the inconsistent global state S_1, $D^{S_1}[1,1] = 1$ since P_1 has only one message delivered. Similarly, $D^{S_1}[2,2] = 3$ and $D^{S_1}[3,3] = 1$ as P_2 and P_3 have 3 and 1 messages delivered respectively. 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. $D^{S_1}[1,3]$ has a value of $\perp$, since P_1 does not get *any* message from P_3. The message m is an orphan message with respect to S_1 since it has been delivered but not yet been sent. This is reflected by $D^{S_1}[1,2] > D^{S_1}[2,2]$. The consistent global state S_2 is represented by the dependency matrix D^{S_2} in which all diagonal elements dominate the columns. The message m' is a lost message but cannot be



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

Figure 1: Inconsistent and Consistent Global States and Their Dependency Matrices

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. In our scheme, recovery is done by restoring a maximal recoverable state, which always exists.

3 A Centralized Recovery Protocol

The centralized message logging and recovery protocol is described in [JZ90] and is summarized here. Messages delivered are logged asynchronously with respect to processing of the message. Checkpoints are also taken independently and occasionally. As more and more messages are logged and checkpoints are 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 try to 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 to 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 RS. 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 must be 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 and is hence restartable.

The only overhead during normal processing is the tagging of extra information in each message sent and the occasional transmission of new stable intervals to the centralized process P_c. When a message is sent, the state interval index $D_j[j]$ of the sender P_j is tagged onto the message. 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. In [CB91], there is a proof on the lower bound of tracking transitive information to be a vector of size n. 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 and is done 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 must be rolled back to the maximal recoverable global state. They are forced to fail and then to execute the recovery protocol. Pessimistic as it might be, this often results in a significant rollback, even when only a single message is an orphan. However, those orphan processes can be allowed to continue to execute, as long as the orphan messages are irrelevant with respect to the overall computation.

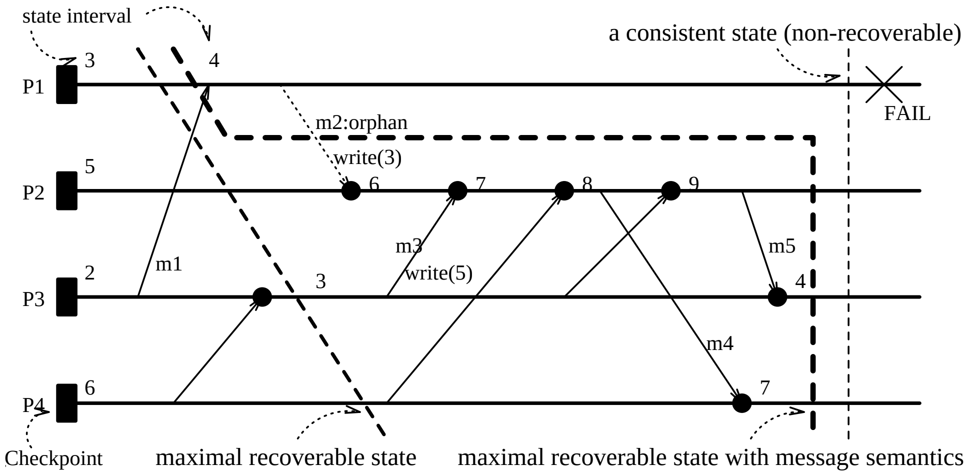


Figure 2: Reducing Orphan Rollback

Consider Figure 2 as an example, where a dark rectangular block denotes a checkpoint and a dark 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 [SY85]. In Johnson and Zwaenepoel's protocol [JZ90], 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 that can be recovered 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. In general, *read* and *write* messages correspond to state-inquiry and state-overwriting messages.

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

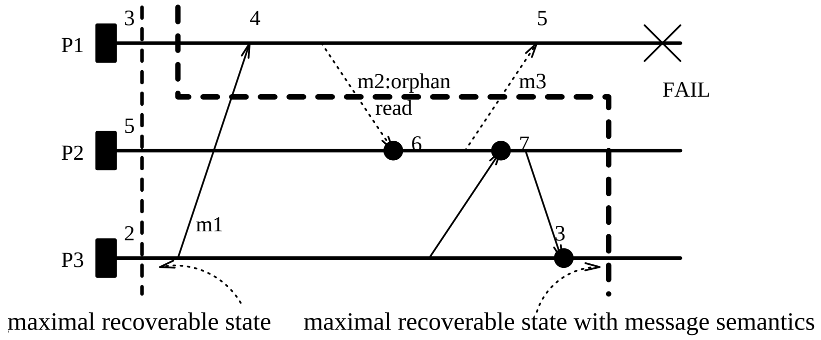


Figure 3: Reducing Orphan Rollback: Another Example

message m_3 for request m_2 is sent to P_1. Since P_1 has failed, the reply message will be discarded. We therefore say that m_2 is irrelevant in the computation[2]. Removing m_2 leads to a better recoverable state $RS' = \langle 3, 7, 3 \rangle$.

The improvements to the above examples are based on the detection of messages that are irrelevant with respect to the computation, defined as the *insignificant* messages. Insignificant messages can always be removed logically before the construction of a recoverable state. This results in a reduction of the dependency among messages, and serves to improve the recoverable state.

Let us use the term *explicit state change* to denote the change in the content and value of the object/process and *implicit state change* to mean the obligation to send out response messages, as a result of the processing of the delivered 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 last three conditions. For example, the lost message m_3 in Figure 3 causes m_2 to become insignificant. The second condition says that a "useless" message (i.e., a message causing no explicit nor implicit state change) is insignificant. The third condition captures the semantics of the *read* message and messages with similar property. The last two conditions capture the overwriting property of messages, such as the *write* message, in a direct and an indirect manner. In all conditions, P_i cannot send any significant messages, since

[2] Our approach extends beyond shared memory objects. When P_2 exhibits a request/response communication pattern (P_2 is a server), m_2 is insignificant regardless of m_3. However in general, when m_2 is received, it may generate more than one message (if P_2 is not merely a server). Whether m_2 is irrelevant depends also on the set of messages sent during the state interval started by m_2. If m_3 is not sent to P_1 but to P_3 and is logged, m_2 will *not* be irrelevant. This is expressed by the third condition for an insignificant message in the next section. If P_1 had not failed, m_1 would have been logged before the recovery protocol is executed and the orphan problem does not arise.

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 can therefore express the above conditions in terms of dependents. For example, the third condition can be rephrased as m causes no explicit state change and all its dependents are insignificant.

5 A Recovery Protocol with Semantics

Having demonstrated how the identification and removal of insignificant messages may lead to an improvement over the recoverable state and a reduction on the amount of rollback, we now develop a protocol to realize this idea. Two issues are to be resolved when insignificant messages are to be detected and logically removed from the recoverable state computation. The first issue is a mechanism to identify insignificant messages. The second issue is to handle the changed dependency when such messages are "removed". We can apply directly the definition of an insignificant message to classify every message of interest to us. This is, however, very inefficient, since the original definition is a recursive one. A more efficient algorithm must be developed. Also, when an insignificant message is removed, it will leave a "gap" in the state interval index, which is a count of the number of messages delivered. This can be remedied by replacing a removed message by a dummy message sent from the message receiver to itself. Such an "internal" message does not carry any dependency and has no impact on the detection of a better recoverable state. Changed dependency can be regenerated by keeping track of the tags of delivered messages in an array in.

P_i maintains the dependency vector $D_i = \langle d_{i1}, d_{i2}, ..., d_{in} \rangle$ and logs delivered messages asynchronously.
Associated variables: $depvec$ is the dependency vector of messages
    in saves the tag of an incoming message, used when the message becomes insignificant
    $type$ is the possible message semantics ($read/write/other$)
    $class$ is the message classification ($tsig/insig/tinsig/unknown$), where
        $tsig$ and $tinsig$ mean temporarily significant and temporarily insignificant, respectively
    ctr counts the 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} and send 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**   (**)
        $t \leftarrow d_{ii} - 1$ (**)
        while $ctr[t] = 0$ **do** $class[t] \leftarrow insig$; $t \leftarrow t - 1$ /* for all previous intervals with no message sent */   (**)
    endif   (**)
end
When state interval σ_i^k becomes stable:
    send the dependency vector to the coordinator

Figure 4: The Recovery Protocol with Insignificant Messages: Normal Operation

The modified recovery protocol is similar to [JZ90]. The differences are the maintenance of extra information to facilitate the detection of insignificant messages, the actual identification of these messages after a failure and the logical removal of these messages. For this purpose, we define the procedure $find\_insig(m)$ to determine whether a message m is insignificant or not. The procedure is straightforward: apply the definition of an insignificant message. To improve efficiency, we must perform some pre-processing before invoking this procedure. The first task is the online classification of messages in the simple cases, which can be done locally on the process which receives and processes the message. The second task is to perform a topological sort on the required ordering for the other messages to be classified. The complete recovery protocol is depicted in Figures 4, 5 and 6. Before explaining the protocol, we will illustrate the classification of messages.

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"
    accept messages from stable storage log and perform the computation
    when state interval reaches δ_i, discard all logged messages, set mode back to "normal"
end

Figure 5: The Recovery Protocol with Insignificant Messages: Failure Recovery

It is easy to classify whether a message is insignificant or not if it does not have any dependent. This is handled by the local process when a new message is delivered. If the previous message is "read-only" and no message has been sent out, then it must be insignificant. Similarly, if the current message is a *write* message, all previous consecutive messages that do not have any 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 significant 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 still active, we cannot determine whether it is significant or not, without knowing the future. We therefore introduce the status *tsig* (temporarily significant) and *tinsig* (temporarily insignificant) for these messages. With a non-terminating computation, it is not possible to conclude that a message is significant, due to the transitive nature of the definition. For example, you never know whether the state change caused by a particular message will be overwritten in the near future and whether all its dependents will become insignificant or not. It is not safe to say that it is significant. 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, whereas *tinsig* messages may be reclassified. 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, in which ten messages have been sent after the current recoverable state. Message

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 messages and computes a new state interval.
Normal operation - when stable state interval σ_i^k is received:
try to include the stable state interval into the current recoverable state RS to form a better one, as in [JZ90]
After a failure:
begin
receive the new stable state intervals from all non-failed processes, and information about insignificant messages
form a recoverable state from available stable state intervals
determine the 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' **do** in temporal order of the messages
let m' start the state interval $\sigma_i^{k_i}$ on P_i, sent from P_j
/* reconstruct the dependency vectors of affected messages */
$depvec[k_i][j] \leftarrow depvec[k_i - 1][j]$
propagate the changed dependency forward, recompute the new dependency for any message sent from P_j,
using the 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 with Insignificant Messages: Coordinator

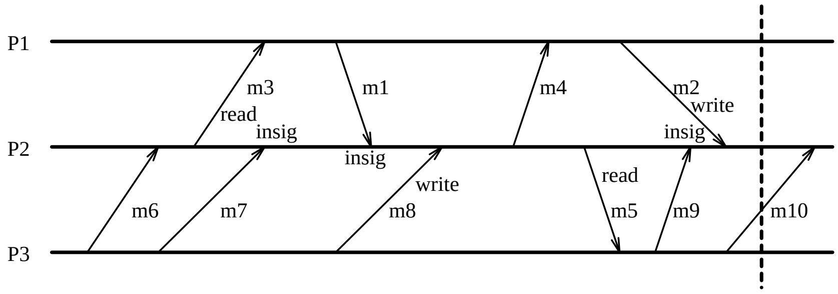


Figure 7: Classifying Messages: An Example

m_{10} is a lost message. Several messages have been classified by the local processes as simple cases. Messages m_9 and m_1 are *insig*, since they are before a *write* and do not result in responses. Indirectly, m_7 is *insig*. All the other seven messages are still unclassified. We start the classification by finding a topological sort on the reversed dependency of the messages. It is clear that message m_3 depends on m_6, since it is sent in the state interval that m_6 creates. Similarly, message m_1 depends on m_3. Furthermore, the program order on the delivery events dictates that m_7 *happens-before* m_8 [Lam78]. We reverse the direction of the dependency to form a precedence relation used in the topological sort: $m_{10} \rightarrow m_5, m_9 \rightarrow m_5, m_2 \rightarrow m_4, ...$ for reversed message dependency as well as $m_{10} \rightarrow m_9, ...$ for reversed program order. As per the definition of the insignificant messages (and hence significant messages), they are often classified according to the types of their dependents. To allow an iterative implementation, that uses only currently available information, the classification should be carried out in the reversed order of dependency. A topological sort on the reversed dependency determines this iterative classification order. Based on the example, the 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 it is unclassified and 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)), unless proven to be wrong in the future. 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 the next *write* are *insig* and all their dependents are *insig*. We can observe that with several *write* messages, it is possible to generate many insignificant messages. The classification procedure is depicted in Figure 8. The conditions in the procedure reflect the definition of an insignificant message. A temporary status is assigned to those messages that cannot be identified with full confidence.

Procedure *find_insig*(m)
begin
    let m be sent from P_j and start state interval $\sigma_i^{k_i}$ on P_i
    if $class[k_i] \neq unknown$ **then return** $class[k_i]$
    let num_dep be the number of effective dependents of m
    if any dependent of m is $tsig$ **then** $class[k_i] \leftarrow tsig$ (1)
    elseif $type[k_i] = read$ or $type[k_i + 1] = write$ **then**
        if all dependents are $insig$ and $\sigma_i^{k_i}$ is not active **then** $class[k_i] \leftarrow insig$ (2)
        elseif all effective dependents are $insig$ or $tinsig$ **then** $class[k_i] \leftarrow tinsig$ (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 are $insig$ and all dependents of these messages are $insig$ **then** $class[k_i] \leftarrow insig$ (4)
        elseif all of them and their effective dependents are $insig$ or $tinsig$ **then** $class[k_i] \leftarrow tinsig$
    elseif $num\_dep = 0$ **then** $class[k_i] \leftarrow tinsig$ /* no effective dependent, assume insignificant */
    endif
    if $class[k_i] = unknown$ **then** $class[k_i] \leftarrow tsig$ /* not yet classified, assume significant */   (5)
    return $class[k_i]$
end

Figure 8: The Procedure *find_insig*()

After a failure, the coordinator obtains the necessary information from all other processes, computes the first possible recoverable state, and then tries to discover 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 (this is carried implicitly in the information tagged with each message when it is being sent out) and a topological sort is carried out. Each message m is classified by a call to the procedure *find_insig*(m) in this topological order. Finally the insignificant messages are "removed" from the computation. To remove a message logically, we only 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 $in[\sigma_i^k]$ 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 not sent by P_j, by changing the j^{th} component of their dependency vectors with that same value. If another message m' (not going to be removed) 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 have treated lost messages to be insignificant and this helps to improve the recoverable state. However, this definition results in a non-prefix-closure property (non-monotonicity of recoverable states). In particular, a non-state-modifying message that has only caused lost messages to be sent is insignificant. When the "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 of recoverable state by being conservative: change the first condition in the definition of insignificant messages by assuming a lost message to be significant, 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[3] of an *insig* message, the recoverable states determined using only insignificant messages will continue to be valid and form a monotonic sequence. Despite adopting the more conservative precaution, the recoverable state obtained by our approach still dominates that obtained under traditional definition of consistent global state.

Output commitment depends on the recoverable state computation. When the state interval generating the output message to the environment becomes part of a recoverable state, the output can be committed to the outside world. The faster the recoverable frontier moves forward, the smaller latency the output suffers. Our approach is good in this sense. However, the overhead of recoverable state computation may be high, if the best possible recoverable state is needed. It is normally the case that only a running estimate of the recoverable state which could be suboptimal is used for output commitment. However, our recoverable state is always at least as good as those obtained through traditional protocols.

6 Processes Acting as Servers

In this section, we consider processes that act as servers for other processes. They exhibit a special communication pattern of request and response pairs. These special processes deserve our attention since they admit further optimization that is not possible with mere detection of insignificant messages.

Lessons learned from the database community in utilizing abstract data types (objects) to improve the level of concurrency in concurrency control protocols [HW91, Wei89] enable us to apply such methodologies on objects in general. Servers are objects, as they exhibit the same unique request/response pattern. They serialize concurrent requests by placing them in a queue or a bag, and serve them one by one sequentially. We would model a server process as an object providing operation types, which make up the interface between the server and its potential clients. For example, a *file server* supports the *read* and *write* operation types and a *counter* object provides the *increment*, *decrement*, *read* and *write* (*reset*) operation types. The action of a client invoking an operation can be modeled as remote procedure call. The request message from the client to the server is called the operation *invocation* and the response message from the server to the client as a result of the computation request represents the operation *response*. In terms of objects, messages are modeled as events. The two events, operation invocation and operation response, together constitute an *operation* p, which is represented as $\langle inv(p), resp(p)\rangle$. Operations of an object may *commute* with each other. We say that two operations p and q on an object commute if the resultant object state and the responses generated for the two operations are identical no matter in which order p and q are executed. The commutativity relation on a counter object with *increment, decrement, read* and *write* operations is depicted in Table 1 where a $\surd$ indicates commutativity. For example, two increment operations $\langle inc(3), ok\rangle$

[3] A property p is stable if once it becomes true, it remains true forever. In other words, $p \Rightarrow \Box p$.

and $\langle inc(2), ok\rangle$ commute since the object state remains the same whatever order they are executed. The same holds between an increment and a decrement operation.

Counter	$\langle inc(x), ok\rangle$	$\langle dec(x), ok\rangle$	$\langle read(), x\rangle$	$\langle write(x), ok\rangle$
$\langle inc(y), ok\rangle$	$\surd$	$\surd$		
$\langle dec(y), ok\rangle$	$\surd$	$\surd$		
$\langle read(), y\rangle$			$\surd$	
$\langle write(y), ok\rangle$				

Table 1: Commutativity Relation for the Counter Object

Restricting the communication pattern of a process to the pattern exhibited by a server improves the recoverable state. As an example, consider the counter object C serving three processes P_1, P_2 and P_3 in Figure 9. Suppose we place the implementation requirement on the server that when it recovers from failure, it must poll all existing processes for requests that are lost during the failure. This is similar to the assumption that lost messages are handled and resent by the recovery mechanism, as in many practical protocols. The maximal recoverable state was $RS = \langle 3, 5, 2, 1\rangle$, when C fails. Taking insignificant messages into account as described in Section 5 does not improve anything under this situation, as none of the messages Q_1, Q_2 or Q_3 is *read* or *write*. Since all the three operations carried by the requests Q_1, Q_2 and Q_3 are commutative with one another, we have an alternative. The problem can be solved by observing that the order in which these three operations are executed does not change the final object state or the three responses generated (and hence the effect or dependency created on the client processes by these responses). As long as C recovers, the maximal recoverable state can be considered as $RS' = \langle 3, 7, 4, 3\rangle$. This is especially useful if P_1 and P_2 have also sent some messages to other processes (say, P_4) and those messages have been logged. Our approach will reduce orphan messages and their cascading effect on these other processes like P_4. After the failure, C recovers and reexecutes the three operations carried by Q_1, Q_2 and Q_3 in *any* order to restore the current object state.

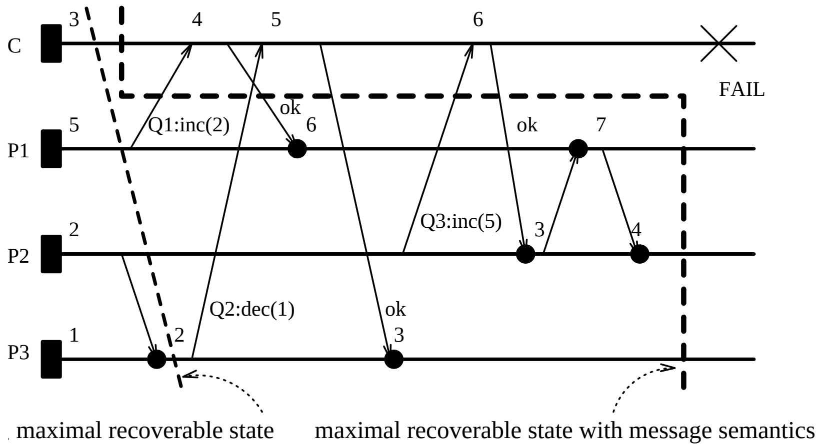


Figure 9: Object Semantics Model: Example 1

Checkpoints which do not correspond to stable state intervals can also benefit from the commutative nature of operations. This can be illustrated in Figure 10. In this situation, P_3 fails. C also fails after taking a checkpoint. The maximal recoverable state would be $RS = \langle 5, 6, 2, 2\rangle$. If P_3 can recover, the maximal recoverable state could be $RS' = \langle 6, 7, 4, 2\rangle$. The reason is that when Q_2 is regenerated, the response to the operation for Q_2 can be regenerated at the object state at state interval 3. The same response will be generated as when the operation is carried out in state interval 5 (the correct state interval), again due to the commutativity of the operations. Now what if P_3 cannot regenerate Q_2 when it recovers? This is shown in Figure 11, where the message from P_2 to P_3 has not yet been logged. The maximal recoverable state would be $RS = \langle 4, 6, 2, 1 \rangle$. If we have the inverse operation defined, we can still take the maximal recoverable state to be $RS'' = \langle 6', 7, 4, 1 \rangle$, where state interval $6'$ is generated by performing the inverse operation of Q_2 on the object state at state interval 6 (the checkpoint). Again the commutativity of operations allows Q_2 to be swapped with Q_3. The object C is indistinguishable whether Q_2 is executed before Q_3 or after Q_3. Performing the inverse for Q_2 (denoted by $dec^{-}(1)$, which is equivalent to $inc(1)$) at the checkpoint cancels the effect of Q_2 effectively as if Q_2 were never received.

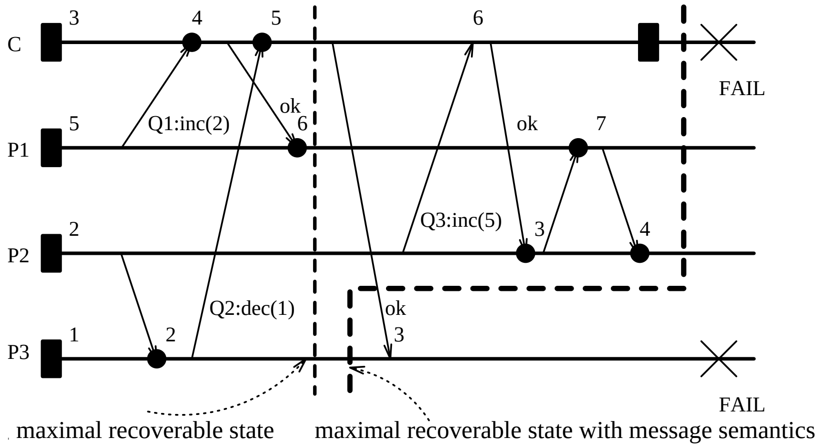


Figure 10: Object Semantics Model: Example 2

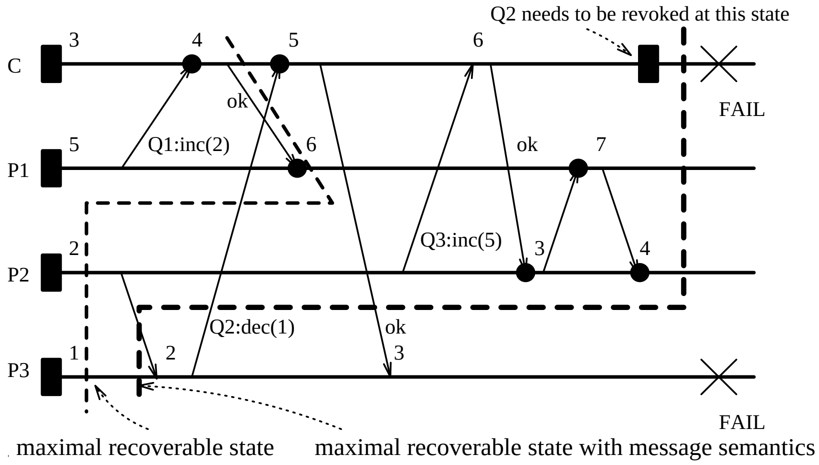


Figure 11: Object Semantics Model: Example 3

Commutativity is too strong a requirement in our application in this section. We should opt for a weaker notion that would enable us to do the same kind of improvement. The notion of right backward commutativity [Wei89] is particularly useful. Informally, we say that an operation q *right commutes backward* with another operation p, if whenever q can be executed at the state after p has been executed, it is possible to swap these two operations so that q is executed before p. Furthermore, the final states of execution in either q after p or p after q are identical. Note that this definition of right backward commutativity is *not* symmetric, whereas the usual definition of commutativity (described above) is symmetric. The formal definition can be found in [Wei89] and an alternative definition in [LAA93].

Bounded Counter	$\langle inc(x), ok\rangle$	$\langle dec(x), ok\rangle$	$\langle dec(x), no\rangle$	$\langle read(), x\rangle$	$\langle write(x), ok\rangle$
$\langle inc(y), ok\rangle$	$\surd$	$\surd$			
$\langle dec(y), ok\rangle$		$\surd$	$\surd$		
$\langle dec(y), no\rangle$	$\surd$		$\surd$	$\surd$	
$\langle read(), y\rangle$			$\surd$	$\surd$	
$\langle write(y), ok\rangle$					

Table 2: Right Backward Commutativity Relation for the Bounded Counter Object

The right backward commutativity relation for the counter object happens to be identical to its commutativity relation. However, it can be verified that right backward commutativity is at least as weak as commutativity. This means two operations that commute must also right commute backward with each other. However, two operations that right commute backward with each other may not happen to commute. Consider the *bounded* counter object, which is a special counter that will refuse an decrement operation if the decrement operation causes the counter value to fall below zero. Thus executing $dec(8)$ on a bounded counter with value 10 causes the value to be changed to 2 and a response of *ok* is returned, whereas executing the same operation invocation when the value is 5 generates a response of *no* and the counter value is unchanged. The right backward commutativity relation for a bounded counter is depicted in Table 2. A $\surd$ indicates that the operation along the row (with parameter y) right commutes backward with the operation along the column (with parameter x). For example, $\langle inc(5), ok\rangle$ right commutes backward with $\langle dec(3), ok\rangle$ since the increment operation can be pushed backward before the decrement operation, so that the decrement operation still generates a response of *ok* and the final counter value remains unchanged. However, $\langle inc(5), ok\rangle$ does not right commute backward with $\langle dec(3), no\rangle$, since executing the increment operation before the decrement operation will render the response *no* for the decrement operation invalid. The correct response will then be *ok*. Similarly, it can be verified that $\langle dec(3), no\rangle$ right commutes backward with $\langle inc(5), ok\rangle$, though the converse is not true. This illustrates that the right backward commutativity relation is not symmetric. As a comparison purpose, the commutativity relation for the bounded counter object is shown in Table 3, which is stronger than the corresponding right backward commutativity relation. The notion of commutativity when the response of an operation is taken into account is an extension from the original definition in [Kor83]. State equivalence and validity of operations are required.

Bounded Counter	$\langle inc(x), ok\rangle$	$\langle dec(x), ok\rangle$	$\langle dec(x), no\rangle$	$\langle read(), x\rangle$	$\langle write(x), ok\rangle$
$\langle inc(y), ok\rangle$	$\surd$				
$\langle dec(y), ok\rangle$		$\surd$			
$\langle dec(y), no\rangle$			$\surd$	$\surd$	
$\langle read(), y\rangle$			$\surd$	$\surd$	
$\langle write(y), ok\rangle$					

Table 3: Commutativity Relation for the Bounded Counter Object

Generalizing from the examples we have illustrated and using right backward commutativity, we can define the operation invocations (messages) which satisfy the commutativity property to be *weakly significant*. To be more specific, we say that a message m_1 is weakly significant with respect to m_2 if the operation represented by m_1 (as well as the response generated by m_1) right commutes backward with the operation represented by m_2. For example, we say that the invocation message $dec(2)$ of the operation $\langle dec(2), ok\rangle$ is weakly significant with respect to the invocations of $\langle inc(5), ok\rangle$ and $\langle dec(8), ok\rangle$. A message m is said to be *held* by another message m' if m' is unlogged and precedes m. Furthermore, m does not right commute backward with m' and there is no checkpoint between m' and m. We also say that m' holds m. A message is weakly significant if it is not being held by any message. Weakly significant messages are basically insignificant, but they may affect the checkpointed state, as in Figure 11, and may require the inverse operation to be performed. Consider an example on weakly significant messages in Figure 12. In the example, m_5 is weakly significant with respect to m_4 and m_6 is weakly significant with respect to both m_4 and m_5, since the respective right backward commutativity relations hold. However, m_7 is weakly significant with respect to every preceding message, except m_4. Finally, m_8 is weakly significant with respect to all except m_5 and m_6. Simple table lookup can determine the status of a message. A message that is not weakly significant is held by some message. We say m_7 is held by m_4 and m_8 is held by m_5 and m_6. When all the messages holding a message m are logged, m becomes weakly significant.

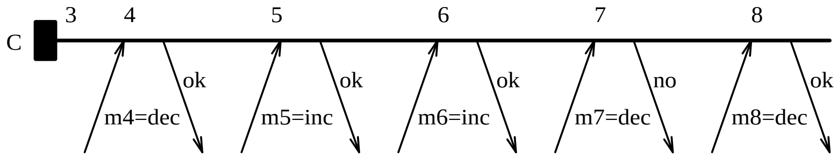


Figure 12: Weakly Significant Messages

The protocol is a modification to the protocol in Section 5. In addition to insignificant messages, weakly significant messages are also taken into account for server processes. The status of the invocation messages are kept track of by the server. They are reported to the coordinator when those messages become weakly significant. During a recoverable state computation, weakly significant messages can be considered insignificant, and they can be removed from the computation logically as before. Furthermore, weakly significance is a stable property and will not result in non-monotonic recoverable states. To utilize checkpoints that are ahead of the current recoverable state, as in Figures 10 and 11, the following steps can be taken. If all unlogged messages from the current recoverable state to the next checkpoint are weakly significant, they can be considered logged and the next checkpoint used to compute a better recoverable state. If a logged message at the server is an orphan in a particular recoverable state, there are two options: either to rollback the server process as in traditional approaches, or to eliminate the effect of the orphan message for an improved system state. If that orphan message is not holding any subsequent message (up to the next checkpoint) in case that it is eliminated (becomes unlogged), its effect can be reversed by performing its inverse operation on the next available checkpoint and the message itself removed logically, in the hope that the modified checkpoint and dependency can form a better recoverable state. The modification to the protocol to cater for checkpoint usage is more complicated and would not be effective when many messages are not weakly significant.

7 Decentralized Recovery Protocols

The protocols in Sections 3 and 5 are centralized in dedicating a particular site as the coordinator. This offers an advantage of a simpler algorithm in the computation of the recoverable state, since the coordinator has all the necessary information. Also the computation needs to be performed only once, instead of being repeated at every process. The network traffic of sending the stable state intervals is also reduced. However, a centralized algorithm is not fault-tolerant and the coordinator is prone to failures. When the coordinator fails, the recoverable state must be recomputed from scratch, wasting computation already made. The coordinator may also become a bottleneck of computation. It would therefore be natural to quest for a decentralized recovery protocol, which supports distributed identification of the recoverable state and independent recovery after a failure.

A first step to decentralize these protocols is through the replication of recovery information, which can be exchanged in background. We can have the coordinator send the current recoverable state to other processes periodically. For example, the coordinator can send the first version of recoverable state to P_1, the second version to P_2 and so on, in a round robin manner. This ensures that each process has at least a copy of the recoverable state, though not up-to-date. After the failure of the coordinator, the process holding the most recent copy can be elected as the new coordinator. It gathers the stable state interval information from other processes, and computes the recoverable state. Processes can also take turn in becoming the coordinator, for better workload distribution and network traffic patterns. Note that the existence of a coordinator in this scenario is to provide a central computation agent to compute the recoverable state, which is the only information required by each process to perform independent recovery.

The above modification only causes the information to be replicated, without decentralizing the computation. A further step can be taken by having each process execute the same algorithm to determine their versions of the recoverable state. Due to the asynchrony of background stable intervals exchange, each process may reach a different recoverable state. This does no harm for the purpose of committing output, since the structure of maximal recoverable system state forms a lattice [JZ90]. The current maximal recoverable state dominates the least upper bound of all the local recoverable states computed by the processes. Thus the local version of the recoverable state is good enough to guarantee that no rollback is possible beyond the output commitment point. However, for the purpose of distributed recovery, it is important that each process gets the same view of the current recoverable state. It is therefore necessary to compute this least upper bound before each process performs the recovery action. This approach is simple and fast, but the resulting least upper bound can be suboptimal since each process does not have a complete knowledge of the recoverable state with respect to all the stable state intervals. A second approach is to have all the processes exchange the stable state intervals before computing the recoverable state. This algorithm usually results in more messages to be exchanged after the failure, but then each process will compute an identical maximal recoverable state. After the first phase of message exchanging, the processes can perform independent recovery, without further communication. When output commitment is considered, the first approach will be sufficient, since suboptimality is not a major concern. The second approach will lead to excessive messages to be exchanged and is not preferred for merely output commitment purpose.

Incorporating insignificant messages into the recoverable state computation may lead to the non-monotonicity of recoverable states, as discussed in Section 5. We can still apply the above two approaches for full decentralization. Unfortunately, owing to the non-monotonicity nature, both approaches may result in an incorrect recoverable state, since the lattice structure is no longer preserved in the presence of insignificant messages that can later become significant. To amend for the deficiency, we should compute insignificant messages in a more conservative manner. We assume all messages that are temporarily insignificant to be significant in the computation of the recoverable state, as discussed at the end of Section 5. Since the insignificance of messages is a stable property, the resulting recoverable states form a monotonic sequence. The lattice property is preserved and correctness guaranteed.

8 Performance Issues

Centralized protocols are generally more efficient in terms of message complexity and computation overhead. They are, however, less fault-tolerant and unfair. The failure of the coordinator implies recomputation of the recoverable state from scratch. A fully distributed recovery protocol that supports independent recovery is often too costly. There is a tradeoff between fault-tolerance and efficiency. Speaking about performance, it is necessary to quantify the overhead in providing fault-tolerance. There are two kinds of overheads: failure-free overhead and recovery overhead. Failure-free overhead is the fixed overhead imposed on the system even when there is no failure (fortunately). Recovery overhead is the cost paid in order to be able to restart the system from a certain state. Assuming computation is much faster than communication, we can concentrate on the message overhead, on both message size and message count. We call the messages of the underlying computation *application messages* and the messages of the recovery protocol *protocol messages*.

Sistla and Welch [SW89] 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. Messages are logged asynchronously in the sequential order they are received. Their recovery protocol (with transitive dependency) is a replicated version of a "centralized" protocol, since each process performs exactly the same computation to arrive at the same recoverable state. There is an overhead of $O(n)$ for tagging transitive dependency information for each application message, the only failure-free overhead. The number of messages exchanged after a failure increases from $2n$ to $O(n^2)$, when going from centralization towards distribution. Each message is of size $O(1)$, containing the best stable state interval index (making use of the FIFO property of message logging and message delivery). Furthermore, a redundant n-folded computation is made. Without transitive dependency information appended to messages, their second protocol has a failure-free overhead of $O(1)$ per application message, but requires $O(n^3)$ messages of size $O(1)$ to be sent for recovery, whereas $O(n^2)$ messages suffice in the corresponding centralized version. The failure-free overhead for the [SY85] protocol is $O(n)$ per message. The size of a recovery message is $O(1)$. However, the number of recovery messages is dependent on the execution status, and in the worst case, it is exponential $O(2^n)$ due to cascading rollback. Cascading rollback is limited by the existence of a nondecreasing maximal recoverable state and also by the number $2^n - 1$.

In the [JZ90] protocol, the failure-free overhead is $O(1)$ per application message and the exponential number of recovery messages caused by cascading rollback in the [SY85] protocol manifests itself as a prolonged computation inside the main loop in the computation of the recoverable state by the coordinator. The number of recovery messages, each of constant size, is $O(n)$: $n-1$ to collect the stable state interval index and $n-1$ to announce the result. Additional failure-free overhead is needed for this protocol as protocol messages are exchanged for incremental recoverable state computation and output commitment. The coordinator must be aware of each stable state interval and the number of protocol messages sent becomes a linear function of the number of application messages delivered to each process. To avoid an excessive overhead, protocol messages can be combined together and sent as a batch. They may even be piggybacked on application messages. Let the average number of application messages delivered to a process be a, and the batch size is b (when b state intervals become stable, the information is sent in a single message to the coordinator). Assume that the recoverable state computation is invoked r times during the parallel computation (the algorithm is invoked when there is a failure, or when it is necessary to commit the output to the outside world). To be more exact, the total number of failure-free protocol messages sent in the protocol is $O(a'n)$, where $a' = \frac{a}{br}$. Each message is of size $O(b)$. Note that the factor of a has been amortized over the batch and the number of times the recoverable state algorithm is invoked. Johnson and Zwaenepoel did not consider distribution of their protocol, but a natural replication of the algorithm on all the processes would lead to $O(n^2)$ recovery messages to be sent, after which independent recovery can begin on each process and an n-folded computation. Failure-free protocol message complexity would then be $O(a'n^2)$.

Our protocol is based on [JZ90] and has the same message complexity of $O(a'n)$, where the factor of a is amortized in the same way over batch size and number of recoverable state computation. In the worst case, the total number of failure-free protocol messages is within a factor of 2 with respect to [JZ90]. The size of each protocol message is also doubled. This factor of 2 in the number of messages comes from the reporting of the status of insignificant messages. When the state intervals are reported to the coordinator to be stable, the status of the messages is also reported. The status of the messages may change in the future due to a new message. For example when a *write* message is received, some of the previous messages m' may become insignificant as a result. A separate message is then needed to notify the coordinator about this, if the information about the stable interval started by m' has already been sent, possibly doubling the message count in the worst case, though ths factor of 2 becomes close to 1 as b increases. The failure-free overhead per application message is also $O(1)$. Protocol message complexity for centralized recovery state computation remains $O(n)$. A fully distributed version of our protocol will result in an n-folded computation, and a message complexity of $O(a'n^2)$, the same as Johnson and Zwaenepoel's protocol.

9 Discussion

In this paper, we observe that some messages irrelevant to the underlying computation can be removed logically, resulting in a better recovery behavior in a recovery protocol. We extend the basic [JZ90] 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. Extending the model to include client-server model leads to the definition of weakly significant messages. Considering them in the computation further improves the recoverable state.

The issue of exploiting message semantics were discussed in [LA90]. 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 [AL89] (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 [WF92b]. Our approach, based on the formal model of [JZ90], 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 [WF92a] has to, since their notion of recoverable state is at the checkpoint level but not the state interval level. Inspired by [AL89], we examine the server processes and apply richer semantics like commutativity as in [LAA93] on the operations they support while retaining asynchronous checkpointing and asynchronous message logging schemes.

The identification of insignificant and weakly significant messages improves the recoverable state. The same technique can be applied to make output commitment more timely, by advancing the recoverable state more rapidly, since commitment to output can only be made after the send event of the output message is included in the recoverable state. This will incur more computation overheads even when there is no failure because it is necessary to perform the computation more often. However, the same tradeoff has to be made in all optimistic message logging schemes, including [SW89, JZ90], to commit output.

Traditional definitions of consistent global state and recoverable global state have been focused 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 remaining in the domain of a "consistent" global state, by removing messages that have no impact on the overall computation. Towards another dimension, does there exist an alternative definition for a "consistent" global state, when taking message semantics into account? Moving towards a more practical setting, experimental results are needed to verify the effectiveness of the approach.

References

[AL89] Mustaque Ahamad and Luke 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.

[BBG83] 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.

[BL88] 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.

[BSS91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[CB91] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letter*, 39(1):11–16, July 1991.

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

[FGL82] 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.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems. *Journal of Algorithms*, 11(3):462–491, September 1990. Preliminary version in PODC 1988.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[KT87] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[LA90] Luke Lin and Mustaque 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.

[LAA93] 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.

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

[NT90] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[SA89] 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.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[SW89] 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.

[SY85] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):205–226, August 1985.

[Wei89] William E. Weihl. The impact of recovery on concurrency control. In *Proceedings of the 8th ACM Annual Symposium on Principles of Database Systems*, pages 259–269. ACM, 1989.

[WF92a] Yi-Min Wang and W. Kent 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.

[WF92b] Yi-Min Wang and W. Kent Fuchs. Scheduling message processing for reducing rollback propagation. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 204–211. IEEE, 1992.