Distributed Recovery with K-Optimistic Logging

Om P. Damani
Dept. of Computer Sci.
Uni. of Texas at Austin
damani@cs.utexas.edu

Yi-Min Wang
Microsoft Research
Redmond, WA 98052
ymwang@microsoft.com

Vijay K. Garg
Dept. of Elect. & Comp. Eng
Uni. of Texas at Austin
garg@ece.utexas.edu

Abstract

Fault-tolerance techniques based on checkpointing and message logging have been increasingly used in real-world applications to reduce service down-time. Most industrial applications have chosen pessimistic logging because it allows fast and localized recovery. The price that they must pay, however, is the high failure-free overhead. In this paper, we introduce the concept of K-optimistic logging where K is the degree of optimism that can be used to fine-tune the trade-off between failure-free overhead and recovery efficiency. Traditional pessimistic logging and optimistic logging then become the two extremes in the entire spectrum spanned by K-optimistic logging. Our results generalize several previously known protocols.

Our approach is to prove that only dependencies on those states that may be lost upon a failure need to be tracked on-line, and so transitive dependency tracking can be performed with a variable-size vector. The size of the vector piggy-backed on a message then indicates the number of processes whose failures may revoke the message, and K corresponds to the upper bound on the vector size. Furthermore, the parameter K is dynamically tunable in response to changing system characteristics.

1 Introduction

Log-based rollback-recovery [11] is an effective technique for providing low-cost fault tolerance to distributed applications [5, 13, 22, 33]. For fault-resilience, a process periodically records its state on a stable storage [25]. This action is called *checkpointing* and the recorded state is called a *checkpoint*. The checkpoint is used to restore a process after a failure. However, information stored in volatile memory is lost in a failure. This loss may leave the restored system in an *inconsistent* state [6]. The goal of a recovery protocol is to bring back the system to a *consistent* state after one or more processes fail. A *consistent* state is one where the sending of a message is recorded in the sender's state if the receipt of the message has been recorded in the receiver's state.

In log-based recovery schemes, besides checkpoints, the contents and processing orders of the received messages are also saved on the stable storage as message logs. Upon a failure, the failed process restores a checkpointed state and replays logged messages in their original order to deterministically reconstruct its pre-failure states. Log-based rollback-recovery is especially useful for distributed applications that frequently interact with the outside world [11]. It can be used either to reduce the amount of lost work due to failures in long-running scientific applications [11], or to enable fast and localized recovery in continuously-running service-providing applications [18].

Depending on when received messages are logged, log-based rollback-recovery techniques can be divided into three main categories: pessimistic logging [5, 18], optimistic logging [22, 33], and causal logging [2, 12]. Pessimistic logging either synchronously logs each message upon receiving it, or logs all delivered messages before sending a message. It guarantees that any process state from which a message is sent is always recreatable, and therefore no process failure will ever revoke any message to force its receiver to also roll back. This advantage of localized recovery comes at the expense of a higher failure-free overhead. In contrast, optimistic logging first saves messages in a volatile buffer and later writes several messages to stable storage in a single operation. It incurs a lower failure-free overhead due to the reduced number of stable storage operations and the asynchronous logging. The main disadvantage is that messages saved in the volatile buffer may be lost upon a failure. Other states that are dependent on these *lost* states are called *orphan* states and they need to be explicitly rolled back. Causal logging provides orphan-free recovery without incurring the overhead of synchronous logging. It limits the rollback to the most recent checkpoint on stable storage. It avoids synchronous access to the stable storage except during output commit. These advantages come at the expense of more complex recovery protocols. It logs messages in processes other than the receivers. So synchronization is required during recovery.

In this paper, we focus on optimistic message logging protocols. These protocols have the advantage of low failure-free overhead. In addition, there are many scenarios in which optimistic logging schemes are desirable.

1. **Non-crash failures:** Traditional logging protocols are based on the assumption that processes fail by simply crashing, without causing any other harm such as sending incorrect messages. In practice, there is some latency between a fault-occurrence and the fault-detection. Optimistic protocols can handle this problem, when possible, by identifying and rolling back the faulty states [33].

2. **Software bugs:** Traditional logging protocols assume that successive failures of a process are independent. On restarting a failed process, the cause of the last crash is not expected to lead to another crash. However, when a software bug crashes a program, deterministically recreating the pre-failure computation results in the same bug leading to the same crash. A way to avoid this is to replay the last few messages in a different order, thereby potentially bypassing the bug that caused the original crash [33, 36].

3. **Optimistic computations:** Many applications employ techniques similar to optimistic logging and require rollback capability. (e.g., optimistic distributed simulation [19].) In such applications, the fault-tolerance overhead can be reduced by employing the same dependency tracking mechanism for both the application and the recovery system.

4. **Distributed debugging:** If a program needs to be tested under different message orderings, a technique similar to optimistic recovery can be used. After the result for a particular message ordering is available, a failure can be simulated and a different message ordering can be tried.

5. **Input message cancellation:** Traditional recovery protocols assume that messages from the environment are irrevocable. However, many new classes of distributed applications are emerging that allow the environment to revoke input messages but still do not allow the environment to be modeled as one of the application process. One example is an application based on the integration of log based techniques with transaction processing. For such applications, revoking of an input message can be modeled as a failure in an optimistic system.

In Section 4.6 we modify a pessimistic protocol to introduce roll back capability. In [1], the author states that a causal protocol can be modified to have rollback capability. As of now, however, we are unaware of any causal protocol with explicit rollback capability.

Although pessimistic and optimistic protocols together provide a trade-off between failure-free overhead and recovery efficiency, it is only a coarse-grain trade-off: the application has to either tolerate the high overhead of pessimistic logging, or accept the potentially inefficient recovery of optimistic logging. In practice, it is desirable to have a flexible scheme with tunable parameters so that each application can fine-tune the above trade-off based on the load and failure rate of the system. For example, a telecommunications system needs to choose a parameter to control the overhead so that it can be responsive during normal operation, and also control the rollback scope so that it can recover quickly upon a failure. Furthermore, since system load and failure rate may fluctuate during the execution, it is desirable to have dynamically tunable parameters.

To address this issue, we introduce the concept of *K-optimistic logging* where K is an integer between 0 and n (the total number of processes). Given any message m in a K-optimistic logging system, K is *the maximum number of processes whose failures can revoke m.* Clearly, pessimistic logging corresponds to 0-optimistic logging because messages can never be revoked by any process failures, while traditional optimistic logging corresponds to n-optimistic logging because, in the worst case, any process failure can revoke a given message. Between these two extremes, the integer K then serves as a tunable parameter that provides a fine-grain trade-off between failure-free overhead and recovery efficiency. Our protocol generalizes several previously known protocols.

The trade-off provided is probabilistic in nature. In the worst case, for any value of K, the recovery time can be as bad as that for a completely optimistic protocol. This is not surprising because in the worst case, pessimistic protocols can have as bad a recovery time as optimistic protocols.

The outline of the rest of the paper is as follows. Section 2 presents the related work. Section 3 describes the system model and the recovery problem. Formal definition of the dependency relation and orphan are also given in this section. Section 4 motivates and defines the concept of K-optimistic logging, and gives a description of the protocol. It also discusses several optimizations and implementation issues. Section 6 concludes the paper. The correctness proof and the properties of our protocol are presented in the appendix.

2 Related Work

Strom and Yemini [33] started the area of optimistic message logging. The protocol presented in this paper is similar in spirit to their protocol. They, however, did not define orphans properly and did not distinguish between failures and rollbacks. As a result, their protocol suffers from the *exponential rollback* problem, where a single failure of a process can roll back another process exponential number of times. For the same reason, they could not omit dependency tracking on stable states. They also assumed FIFO message ordering which is not required in optimistic protocols.

Johnson and Zwaenepoel [22] present a centralized optimistic protocol. Unlike most optimistic protocols including ours, their protocol does not require every received message to be logged. This can be advantageous in case average message size is much larger than average checkpoint size. They also use direct dependency tracking instead of transitive dependency tracking. This implies that instead of dependency vectors, only a small constant amount of information is piggybacked on each message. These advantages come at the expense of a centralized recovery manager, which itself needs to be made fault-tolerant.

Smith, Johnson and Tygar [31] present the first completely asynchronous optimistic protocol. Their protocol is completely asynchronous in that, in their system, neither a process is ever blocked from progressing, nor a message is ever blocked from being delivered. This is achieved by piggybacking on each message a data structure that is similar to our incarnation end table. This results in high failure-free overhead. In their protocol, no failure announcement is used. Since the incarnation end table is piggybacked on every message, each process learns about a failure when a message path is established between the restarted failed process and every other process. We believe that rather than letting orphan computation continue for a long time, it is better to announce failure and let every process know as soon as possible. In Strom and Yemini's and in our protocol, a process does not block after a failure, but a message may be blocked from delivery.

To address the scalability issue of dependency tracking for large systems, Sistla and Welch [32] divided the entire system into clusters and treated inter-cluster messages as output messages. Lowry et al. [24] introduced the concept of *recovery unit gateways* to compress the vector at the cost of introducing false dependencies. Direct dependency tracking techniques [32, 22, 21] piggyback only the sender's current state interval index, and are more scalable in general. The trade-off is that, at the time of output commit or recovery, the system needs to assemble direct dependencies to obtain transitive dependencies.

In Table 1 we present a comparison of our protocol with other optimistic protocols that track transitive dependencies. Since no other protocol bridges the gap between optimism and pessimism, we consider our protocol for K equal to n. Note that, using our ideas presented in [8], Smith and Johnson reduced the size of dependency vectors in their algorithm [30].

Our protocol generalizes several previously known protocols. For K equal to n, our protocol reduces to the optimistic protocol presented in [8], while for K equal to 0, it reduces to the pessimistic protocol presented in [20]. The protocol in [3] can be thought of as a variant of our protocol where the server processes set K to 0 and the clients set K to n.

Although no parallel exists to our protocol in the area of message logging, in the area of

	Message ordering	Number of integers piggybacked	Number of concurrent failures	Number of rollbacks per failure	Asynchronous Recovery
Strom and Yemini 85	FIFO	$O(n)$	n	$O(2^n)$	Mostly
Sistla and Welch 89	FIFO	$O(n)$	1	1	No
Johnson and Zwaenepoel 90	None	$O(1)$	n	1	No
Peterson and Kearns 93	FIFO	$O(n)$	1	1	No
Smith, Johnson and Tygar 95	None	$O(n^2 f)$	n	1	Yes
Damani and Garg 96	None	$O(n)$	n	1	Mostly
Smith and Johnson 96	None	$O(nf)$	n	1	Yes

Table 1: Comparison with related work. (n is the number of processes in the system and f is the maximum number of failures of any single process.)

checkpoint-based rollback-recovery, the concept of lazy checkpoint coordination [35] has been proposed to provide a fine-grain trade-off in-between the two extremes of uncoordinated checkpointing and coordinated checkpointing. An integer parameter Z, called the laziness, is introduced to control the degree of optimism by controlling the frequency of coordination. The concept of K-optimistic logging can be considered as the counterpart of lazy checkpoint coordination for the area of log-based rollback-recovery.

3 Theoretical Framework

In this section, we introduce the formal model of the system. We formally define what it means for a state to be dependent on another state. This dependency relation is an extension of the Lamport's happened before relation [23] to failure prone computations.

3.1 Abstract Model

A process execution is a pair $(S, \prec)$. S is a set of elementary entities called state interval (*si* for short). There exists a boolean-valued function *init* that takes a *si* as its input and returns *true* for exactly one of the members of S. The relation $\prec$ is an acyclic binary relation on S satisfying following conditions:

- $\forall s : |\{u : init(s) \wedge u \prec s\}| = 0$
- $\forall s : |\{u : \neg init(s) \wedge u \prec s\}| = 1$

The relation $\prec$ induces a tree on S. If $u \prec s$, u is a parent of s and s is a child of u.

In an online execution, new elements can be added to S at any time with an accompanying strengthening of the relation $\prec$.

A *si* can have one of three labels: *useful*, *lost* and *rolled_back*. Every *si* starts as *useful*. A newly added *si* becomes child of a *useful si* that has no *useful* child. The label of only a *useful*, non-*init si* can be changed. When the label of a *si* is changed, labels of all its *useful* children are also changed in the same way. This change propagates recursively to all descendents.

Consider a system consisting of n processes $P_1, .., P_n$. Let the execution of P_i be $(S_i, \prec_i)$. The system execution is a triplet $(H, \prec, \rightsquigarrow)$. The set H is defined as $H \equiv \bigcup_i S_i$. The acyclic binary relation $\prec$ is defined on H as $\prec \equiv \bigcup_i \prec_i$. The relation $\rightsquigarrow$, another acyclic binary relation defined on H, satisfies the following conditions[1]:

- $\forall s : |\{u : init(s) \wedge u \rightsquigarrow s\}| = 0$
- $\forall s : |\{u : \neg init(s) \wedge u \rightsquigarrow s\}| = 1$

Let the relation $\rightarrow$ be the transitive closure of $\prec \cup \rightsquigarrow$. Two system executions are considered equivalent if their $\rightarrow$ relations, restricted to *useful si*, are same.

3.2 Physical Model and the Recovery Problem

We consider an application system consisting of n processes communicating only through messages. The communication system used is unreliable in that it can lose, delay, duplicate, or reorder the messages. The environment also uses messages to provide inputs to and receive outputs from the application system. Each process has its own volatile storage and also has access to stable storage [25]. The data saved on volatile storage is lost in a process crash, while the data saved on stable storage remains unaffected by a process crash.

The state of a process consists of values of all program variables and the program counter. A *state interval* is a sequence of states between two consecutive message receipts by the application process. The execution within each interval is assumed to be completely deterministic, i.e., actions performed between two message receives are completely determined by the content of the first message received and the state of the process at the time of the first receive. For the purpose of recovery, we are interested in state intervals only and not in states, and therefore for convenience, we use the term *state* instead of *state interval*.

A state interval here corresponds to a state interval (*si*) in the abstract model. If in the abstract model, $s \prec u$, then the interval corresponding to s immediately precedes the interval corresponding to u. If $s \rightsquigarrow u$ then a message is send in the interval corresponding to s and the receive of that message results in the interval that corresponds to u. From now on, when there is no confusion, we use the term 'state s' instead of saying 'state interval that corresponds to *si* s.'

Although an abstract process execution is a tree, a physical process execution is a sequence of state intervals in real time. All n process executions together constitute a system execution. Two physical system executions are considered equivalent if their abstract counterparts are equivalent.

We assume perfect failure detection [7], i.e. each non-failed process eventually learns about all failures in the system and no process falsely assumes that a non-failed process has failed. A process fails by simply crashing. In a crash failure, a process stops executing and loses the

[1] As an aside we note that just like $\prec$, $\rightsquigarrow$ also induces n disjoint trees on H.

data in its volatile storage. The process does no other harm, such as sending incorrect message. Pre-failure states of a process that cannot be recreated after a failure are called lost states. A lost state gets the label *lost* in the abstract model.

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

If an application state is rolled back by the recovery system then that state is called *rolled_back*.

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

From here on, when there is no confusion, instead of saying 'the system does something for the corresponding process', we will say 'a process does something'. We next give a general description of optimistic protocols in this model.

3.3 Optimistic Recovery

Optimistic recovery is a special class of log-based rollback recovery, where the recovery system employs checkpointing and message logging to control the application [11]. In optimistic recovery, received messages are logged in volatile storage. The volatile log is periodically written to stable storage in an asynchronous fashion. By asynchronous, we mean that a process does not stop executing while its volatile log is being written to stable storage. Each process, either independently or in coordination with other processes, takes periodic checkpoints [11].

After a crash, a process is *restarted* by restoring its last checkpoint and replaying logged messages that were received after the restored checkpoint. Since some of the messages might not have been logged at the time of the failure, some pre-failure states, called *lost* states, cannot be recreated. States in other processes that causally depend on lost states are called *orphan*. Causal dependency corresponds to the $\rightarrow$ relation in the abstract model. A message sent by a lost or orphan state is called an orphan message. If the current state of a process is orphan then the process itself is called orphan. All orphan states are rolled back. All orphan messages are also discarded. Each restart or rollback starts a new *incarnation* of the process. A failure or a rollback does not start a new interval. It simply restores an old interval.

Traditional optimistic protocols treat rollback of a failed process as if the process has failed and restarted. We note the distinction between a restart and a rollback. A failed process restarts whereas a rollback is done by a non-failed process. Information stored in volatile memory before a failure is not available at restart. In a rollback, no information is lost. Unlike in traditional protocols, in our protocols, a process informs other processes about its failures only and not about rollbacks.

In all optimistic protocols (or all log-based recovery protocols), the recovered state could have happened in a failure-free execution of the application, with relatively slower processor speed and relatively increased network delays. Therefore, in an asynchronous system, optimistic protocols solve the recovery problem.

3.3.1 Output Commit

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

In optimistic recovery, an output can be committed when the state intervals that the output depends on have all become *stable* [33]. An interval is said to be stable if it can be recreated from the information saved on stable storage. To determine when output can be committed, each process periodically broadcasts a logging progress notification to let other processes know which of its state intervals have become stable. Such information is accumulated at each process to allow output commit.

Example

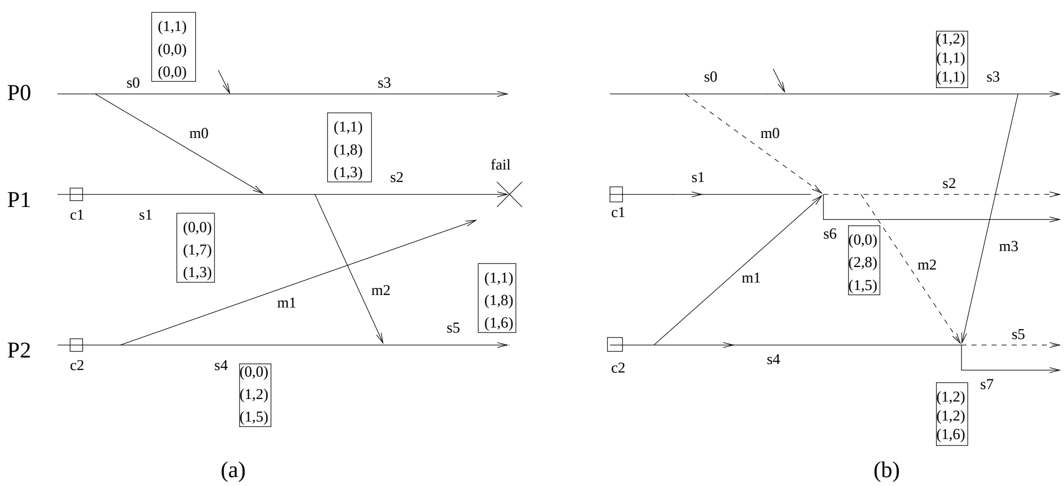


Figure 1: Example: Optimistic Recovery in Action

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

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

Figure 1(b) shows the post-failure computation. On restarting after the failure, $P1$ restores its last checkpoint $c1$, replays all the logged messages and restores the interval $s1$. It then broadcasts a failure announcement (not shown in Figure 1). It continues its execution and starts interval $s6$ by processing $m1$. $P2$ receives the failure announcement in interval $s5$ and realizes that it is dependent on a lost state. It rolls back, restores its last checkpoint $c2$, and replays the logged messages until it is about to process $m2$, the message that made it dependent on a lost state. It discards $m2$ and continues its execution by processing $m3$. The message $m2$ is not regenerated in post-failure computation. $P0$ remains unaffected by the failure of $P1$.

Notations

We next define notations that are used throughout the paper.

- i, j, k refer to process identification numbers.
- t refers to the incarnation number of a process.
- s, u, v, w, z refer to a state (or a state interval).
- P_i refers to the i'th process.
- $P_{i,t}$ refers to incarnation t of P_i.
- $s.p$ denotes the identification number of the process to which s belongs, that is, $s.p = i \Rightarrow s \in S_i$.
- x, y refer to state sequence numbers.
- $(t, x)_i$ refers to the x'th state of the t'th incarnation of process P_i.
- m refers to a message.
- c refers to a dependency vector (defined in Section 3.5).

3.4 Causal Dependency Between States

In the previous section, we talked about one state being dependent on another. The application state resulting from a message delivery depends on (is determined by) the content of the message delivered and therefore depends on the state sending the message. This dependency relation is transitive. It corresponds to the $\rightarrow$ relation defined in the abstract model. Lamport defined the *happened before* relation [23] for a failure-free computation. Our dependency relation is an adaptation of the happened before relation to a failure-prone systems. The physical meaning of the abstract relation $\rightarrow$ is as follows. In a failure-prone system, happened before ($\rightarrow$) is the transitive closure of the relation defined by the following two conditions:

- $u \rightarrow v$, if the processing of an application message in state u results in state v, (for example, $s1 \rightarrow s6$ in Figure 1(b)),

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

We say that u is transitively dependent or simply dependent on s if s happened before u. By $s \underline{\rightarrow} u$, we mean $s \rightarrow u$ or $s = u$. By $s \not\rightarrow u$ we mean s did not happen before u. For example, in Figure 1(b), $s2 \not\rightarrow s6$.

Only application messages contribute to the happened before relation. The recovery protocol might also send some messages. These messages do not contribute to the happened before relation.

Earlier we mentioned that a state dependent on a lost sate is called orphan. We can now formally define *orphan* as:

Definition 1 $orphan(s) \equiv \exists u : lost(u) \ \wedge \ u \underline{\rightarrow} s$

To detect orphans, we need a mechanism to track dependencies between states.

3.5 Dependency Tracking Mechanism

We use dependency vectors to track transitive dependencies between states in a failure-prone system. Although dependency vectors have been used before [33], their properties have not been discussed.

A dependency vector has n entries, where n is the number of processes in the system. Each entry contains an *incarnation number* and a *state sequence number* (or simply *sequence number*). Let us consider the dependency vector of a process P_i. The incarnation number in the i'th entry of P_i's dependency vector (its own incarnation number) is equal to the number of times P_i has failed or rolled back. The incarnation number in the j'th entry is equal to the highest incarnation number of P_j on which P_i depends. Let entry e correspond to a tuple (incarnation t, sequence number seq). Then, $e_1 < e_2 \equiv (t_1 < t_2) \vee [(t_1 = t_2) \wedge (seq_1 < seq_2)]$.

A process sends its dependency vector along with every outgoing message. Before delivering a message to the application, a process updates its dependency vector with the message's dependency vector by taking the componentwise maximum of all entries. The process then increments its own sequence number.

To start a new incarnation, a process increments its incarnation number (it leaves the sequence number unchanged). A new incarnation is always started after a rollback or a failure.

The dependency tracking mechanism is given in Figure 2. An example of the mechanism is shown in Figure 1. The dependency vector of each state is shown in a rectangular box near it. The row i of the dependency vector corresponds to P_i (P_i is shown as Pi in Figure 1) .

3.5.1 Properties of Dependency Vectors

Dependency vectors have properties similar to Mattern's vector clocks [26]. They can be used to detect transitive dependencies between *useful* states (states which are neither lost nor orphan).

We define an ordering between two dependency vectors $c1$ and $c2$ as follows.

$$c1 < c2 \equiv (\forall i : c1[i] \leq c2[i]) \wedge (\exists j : c1[j] < c2[j])$$

```
Process P_i :

type entry =     (int inc, int seq)         // incarnation, sequence number
var c :          array[n] of entry          // n : number of processes in system

Initialize :
      ∀ j : c[j] := (0,0) ;
      c[i] := (1,1) ;
Send_message :
      send (data, c) ;
Process_message (m) :
      // P_i receives the dependency vector 'm.c' with incoming message
      ∀ j: c[j] := max(c[j],m.c[j]) ;
      c[i].seq := c[i].seq + 1 ;
Start_incarnation :
      // A new incarnation is started after a failure or a rollback
      c[i].inc := c[i].inc + 1 ;
```

Figure 2: Dependency vector algorithm

Let $s.c$ denote the dependency vector of $P_{s.p}$ in state s. The following lemma gives a necessary condition for the $\not\rightarrow$ relation between two *useful* states.

Lemma 1 *Let s and u be distinct useful states (neither lost nor orphan). Then, $s \not\rightarrow u \Rightarrow u.c[s.p] < s.c[s.p]$*

Proof: Let $s.p = u.p$. Since s and u are distinct useful states, it follows that $u \rightarrow s$. During processing of a message, $P_{s.p}$ takes the maximum of dependency vectors and then increments the sequence number of its own component. On restart after a failure or a rollback, $P_{s.p}$ increments its incarnation number. Since for each state transition along the path from u to s, the local dependency vector is incremented, $u.c[s.p] < s.c[s.p]$.

Let $s.p \neq u.p$. As $s \not\rightarrow u$, $P_{u.p}$ could not have seen $s.c[s.p]$, the local dependency vector of $P_{s.p}$. Hence $u.c[s.p] < s.c[s.p]$. ■

As shown in the next theorem, the above condition is also sufficient for the $\not\rightarrow$ relation. The next theorem shows that, despite failures, dependency vectors keep track of causality for *useful* states.

Theorem 1 *Let s and u be useful states in a distributed computation. Then, $s \rightarrow u$* **iff** *$s.c < u.c$*

Proof: If $s = u$, then the theorem is trivially true. Let $s \rightarrow u$. Since both s and u are useful, there is a message path from s to u such that none of the intermediate states are either lost or orphan. Due to monotonicity of dependency vectors along each link in the path, $\forall j : s.c[j] \leq u.c[j]$. Since $u \not\rightarrow s$, from Lemma 1, $s.c[u.p] < u.c[u.p]$. Hence, $s.c < u.c$.

The converse follows from Lemma 1. ■

Dependency vectors do not detect the causality for either lost or orphan states. To detect causality for lost or orphan states, we use an *incarnation end table*, as explained in Section 4.3.

4 K-Optimistic Protocol

In this section, we first prove several fundamental properties about optimistic recovery. Using these properties, we design a K-optimistic protocol that bridges the gap between optimism and pessimism. This protocol provides a trade-off between recovery time and failure-free overhead. For K equal to n, the protocol reduces to the optimistic protocol presented in [8], while for K equal to 0, it reduces to the pessimistic protocol presented in [20].

4.1 Motivation

Traditional pessimistic logging and optimistic logging provide a coarse-grain trade-off between failure-free overhead and recovery efficiency: the application has to either tolerate the high overhead of pessimistic logging or accept the potentially inefficient recovery of optimistic logging. For long-running scientific applications, the primary performance measure is the total execution time. For these applications, minimizing failure-free overhead is more important than improving recovery efficiency because failures are rare events. Hence, optimistic logging is a better choice. In contrast, for continuously-running service-providing applications, the primary performance measure is the service quality. Systems running such applications are often designed with extra capacity which can absorb reasonable overhead without causing noticeable service degradation. On the other hand, improving recovery efficiency to reduce service down time can greatly improve service quality. As a result, many commercial service-providing applications have chosen pessimistic logging [18].

The above coarse-grain trade-off, however, may not provide optimal performance when the typical scenarios are no longer valid. For example, although hardware failures are rare, programs can also fail or exit due to transient software or protocol errors such as triggered boundary conditions, temporary resource unavailability, and by-passable deadlocks. If an application suffers from these additional failures in a particular execution environment, slow recovery due to optimistic logging may not be acceptable. Similarly, for a service-providing application, the initial design may be able to absorb higher run-time overhead incurred by message logging. However, as more service features are introduced in later releases, they consume more and more computation power and the system may no longer have the luxury to perform pessimistic logging.

These observations motivate the concept of K-optimistic protocol where K is the degree of optimism that can be tuned to provide a fine-grain trade-off. The basic idea is to ask each message sender to control the maximum amount of risk placed on each message. A sender can release a message only after it can guarantee that failures of at most K processes can possibly revoke the message (see Theorem 6 in the appendix).

This protocol provides a trade-off between recovery time and logging overhead, with traditional optimistic and pessimistic protocols being two extremes. As the value of K moves from n to 0, the recovery time goes down with a corresponding increase in the logging overhead. The parameter K can be dynamically changed to adjust to a changing environment.

4.2 Theoretical Basis

In Section 3.2, we presented the distinction between restart due to a process's own failure and rollback due to some other process's failure. Traditional optimistic recovery protocols [31, 33] blur this distinction and refer to lost states as rolled back states. In order to relate our results to those in the literature, we use the following terminology. A state satisfies predicate $rolled\_back$ if it has been either lost in a failure or explicitly rolled back by the recovery protocol. In traditional protocols, any state dependent on a rolled back state is called an *orphan*. The following predicate formally defines an orphan state for these protocols.

Definition 2 $orphan(s) \equiv \exists u : rolled\_back(u) \ \wedge \ u \underline{\rightarrow} s$

We have presented above definition only for an understanding of the traditional protocols and for the proof of the theorems in this section. In rest of the paper, we use the orphan definition given in Section 3.4. For emphasis, we reproduce that definition here:

Definition 1 $orphan(s) \equiv \exists u : lost(u) \ \wedge \ u \underline{\rightarrow} s$

For orphan detection, traditional optimistic protocols usually require every non-failed rolled back process to behave as if it itself has failed [31, 33]. After each failure or rollback, a process starts a new incarnation and announces the rollback. A process P_i announces a failure or a rollback by broadcasting index $(t, x')_i$ where all states of incarnation t of P_i with sequence number greater than x' have been lost in the corresponding failure or rollback. We observe that announcing failures is sufficient for orphan detection. We give a proof of this observation in the following theorem.

Theorem 2 *With transitive dependency tracking, announcing only failures (instead of all rollbacks) is sufficient for orphan detection.*

Proof. Let a state interval v be orphan because of rollback of another interval u. Interval u rolled back either because $P_{u.p}$ failed or because a rollback of another interval z made u orphan. By repeatedly applying this observation, we find an interval w whose rollback due to $P_{w.p}$'s failure caused v to become orphan. Because of transitive dependency tracking, $P_{v.p}$ can detect that v depends on w. Therefore, $P_{v.p}$ will detect that v is orphan when it receives the failure announcement from $P_{w.p}$. ∎

The above observation was first used in [8] and later used in [30]. We carry this observation even further in Theorem 3, by proving that any dependencies on stable intervals can be omitted without affecting the correctness of a recovery protocol which tracks transitive dependencies. A state interval is said to be *stable*, if it can be reconstructed from the information saved in stable storage.

We say that v is *commit dependent on* w if v is transitively dependent on w and w is not stable. A system is said to employ *commit dependency tracking* if it can detect the commit dependency between any two state intervals. The following theorem suggests a way to reduce dependency tracking for recovery purposes. It states that if all state intervals of P_j, on which P_i is dependent, are stable then P_i does not need to track its dependency on P_j.

Theorem 3 *Commit dependency tracking and failure announcements are sufficient for orphan detection.*

Proof. Once a state interval becomes stable, it cannot be lost in a failure. It can always be reconstructed by restarting from its previous checkpoint and replaying the logged messages. Following the proof in Theorem 2, an orphan interval v must transitively depend on an interval w that is lost in $P_{w.p}$'s failure. This implies that w had not become stable when the $P_{w.p}$'s failure occurred. By definition of commit dependency tracking, $P_{v.p}$ can detect that v transitively depends on w. Therefore, on receiving the failure announcement from $P_{w.p}$, $P_{v.p}$ will detect v to be orphan. ■

A process can explicitly inform other processes of new stable state intervals by periodically sending logging progress notifications. Such information can also be obtained in a less obvious way. A failure announcement containing index $(t, x')_i$ indicates that all states of incarnation t of P_i with sequence number greater than x' have been lost in a failure. Since the state with sequence number x' has been restored after a failure, the announcement also serves as a logging progress notification that interval $(t, x')_i$ has become stable. Corollary 1 summarizes this result.

Corollary 1 *Upon receiving a failure announcement containing index $(t, x')_i$, a process can omit the dependency entry $(t, x)_i$ if $x \leq x'$.*

Corollary 1 is implicitly used by Strom and Yemini [33] to allow tracking dependency on only one incarnation of each process so that the size of dependency vector always remains n: *when process P_j receives a message m carrying a dependency entry $(t, x)_i$ before it receives the rollback announcement for P_i's incarnation $(t-1)$, P_j should delay the delivery of m until that rollback announcement arrives.* This in fact implicitly applies Corollary 1.

We can further apply corollary 1 to eliminate unnecessary delays in message delivery. Suppose P_j has a dependency on $(t-2, x)_i$ when it receives message m carrying a dependency on $(t, x+10)_i$. According to Theorem 3, P_j only needs to be informed that interval $(t-2, x)_i$ has become stable. It does not need to be informed anything about incarnation $(t-1)$ before it can acquire the dependency on $(t, x+10)_i$ and overwrite $(t-2, x)_i$. P_j can obtain that information when it receives either a logging progress notification or a failure announcement from P_i. A more interesting and useful special case is when P_j does not have any dependency entry for P_i at all and so the delay is altogether eliminated.

Based on these results, we have developed an efficient optimistic protocol, which is described next.

4.3 The Protocol

4.3.1 Data Structures

The variables maintained by a process in this protocol are shown in Figure 3. The integer K is the degree of optimism. Dependency tracking is done by the dependency vector c. The messages received from the communication subsystem but not yet delivered to the application are kept in the *Receive_buffer*. The messages sent by the application, but not yet delivered to the communication subsystem are stored in the *Send_buffer*. In *Log_prog*, logging progress

Process P_i :			
type	entry:	(inc int, seq int)	// incarnation, sequence number
var	K:	int;	// the degree of optimism
	c :	array[n] of entry;	// transitive dependency vector
	State_list :	list of state;	// list of non-stable states
	Receive_buffer :	set of message;	// messages received but not yet delivered
	Send_buffer :	set of message;	// messages to be sent
	Log_prog :	array[n] of set of entry;	// logging progress information
	// The following data structures are stored in stable storage		
	cur_inc :	int ;	// current incarnation number
	iet :	array[n] of set of entry;	// incarnation end table
	Stable_state_list :	list of state;	// states saved on stable storage

Figure 3: Variables maintained by a process

information is maintained by keeping an entry for the highest known stable interval of each known incarnation of each process. The received failure announcements are stored in an incarnation end table (*iet*). Variable *cur_inc* stores the current incarnation number in stable storage. This avoids the loss of incarnation number information in a failure. A simplification that we use to clarify the correctness proof is that of replacing checkpointing and message logging with saving of entire states. Our implementation indeed uses checkpointing and message logging. We discuss this point in detail in Section 4.3.10.

4.3.2 Auxiliary Functions and Predicates

Figure 4 shows predicates and functions used in the protocol. We next explain each of them.

- *knows_orphan*: If a state s knows that a state u is orphan then the predicate $knows\_orphan(s, u)$ is true. This is the case, when the *iet* of s shows u to be dependent on a lost state.
- *stable*: If s belongs to Stable_state_list of $P_{s.p}$, then $stable(s)$ is said to be true.
- *seq_num*: This function takes a set of entries and an incarnation number and returns the sequence number associated with the given incarnation number in the set.
- *knows_stable*: A state u is said to correspond to entry e if $u.c[u.p]$ is equal to e. If a state s knows that P_j's state corresponding to entry e is stable then predicate $knows\_stable(s, e, j)$ is true.
- *admissible*: The predicate $admissible(m, s)$ is true if a message m can be processed in a state s. The message can be processed if no dependency on any unstable interval will be overwritten in taking maximum of $m.c$ and $s.c$.
- *get_state*: This function takes a process id and an entry and returns the state interval of the given process that corresponds to that entry.

- *Insert*: This function inserts an entry (t, x) in a set se. If an entry (t, y) for incarnation t already exists in se, then that entry is replaced by $(t, max(x, y))$. This ensures that the set se contains the latest information about incarnation t.
- NULL: A NULL entry is defined to be lexicographically smaller than any non-NULL entry.

knows_orphan(s,u) $\equiv$ $\exists$j: $\exists$ (t,x) $\in$ s.iet[j]: (t = u.c[j].inc) $\wedge$ (x < u.c[j].seq)

stable(s) $\equiv$ s $\in$ Stable_state_list

seq_num(se, t): **return** x **where** (t,x) $\in$ se

knows_stable(s,e,j) $\equiv$ seq_num(s.Log_prog[j], e.inc) $\geq$ e.seq

admissible(m,s) $\equiv$ $\forall$j: [s.c[j].inc $\neq$ m.c[j].inc $\Rightarrow$
    knows_stable(s, min(s.c[j], m.c[j]), j)]

get_state(j,e): **return** s **where** s.p = j $\wedge$ s.c[j] = e

Insert(se, (t,x)): **if** $\exists$ y: (t,y) $\in$ se **then** se := (se - {(t,y)}) $\cup$ {(t,max(x,y))}
    else se := se $\cup$ {(t,x)} ;

e = NULL $\equiv$ (e.inc = NULL $\wedge$ e.seq = NULL)

x = NULL $\equiv$ ($\forall$y $\neq$ NULL: x < y)

Figure 4: Predicates and functions used in the protocol

In the protocol, unspecified state variable s stands for the current state unless otherwise stated. In a predicate, if a message m is used instead of a state u then $u.c$ in predicate definition is replaced by $m.c$.

4.3.3 Initialization

We next describe the actions taken by a process P_i upon the occurrence of different events. The initialization routine is given in Figure 5.

Initialize: Upon starting the execution, a process has no dependency on any other process. Therefore, P_i sets all dependency vector entries, except its own, to NULL. Since each process execution can be considered as starting with an initial checkpoint, the first state interval is always stable. Therefore, P_i updates its *Log_prog* accordingly. We show the initial state being added to the *State_list*. In practice, this is not done as the program itself serves as the initial state.

```
Initialize :
    ∀j : c[j] := NULL;
    c[i] := (1,1);
    ∀j : iet[j] := Log_prog[j] := {} ;              // empty set
    Insert(Log_prog[i],(1,1)); cur_inc := 1 ;
    State_list := {s} ; Stable_state_list := {} ;
```

Figure 5: K-optimistic protocol: Initialization routine

```
Send_message(data) :
    Send_buffer := Send_buffer ∪ {(data, c)} ;
    Check_send_buffer ;

Receive_message(m) :
    if ¬knows_orphan(s,m) then
        Receive_buffer := Receive_buffer ∪ {m} ;

Process_message(m) :
    if admissible(m) then
        c := max(c,m.c) ; c[i].seq := c[i].seq + 1 ;
            // Message m delivered to the application
        State_list := State_list ∪ {s} ;
```

Figure 6: K-optimistic protocol: Routines that manipulate messages

4.3.4 Message Manipulation

Routines that manipulate messages are given in Figure 6.

Send_message: To send a message, the current dependency vector is attached to the message and the message is added to *Send_buffer*. The message is held in *Send_buffer* if the number of non-NULL entries in its dependency vector is greater than K. Messages held in *Send_buffer* are sent in the routine *Check_send_buffer* (in Figure 7).

Receive_message: A received message is discarded if it is known to be orphan. Otherwise, it is added to *Receive_buffer*.

Process_message: When the application needs to process a message, any of the admissible messages among the received ones is selected. A message is admissible, if its delivery does not result in the overwriting of any non-stable entry in the dependency vector. In other words, if delivering a message to the application would cause P_i to depend on two incarnations of any process, P_i waits for the interval with the smaller incarnation number to become stable. This information may arrive in the form of a logging progress notification or a failure announcement. Such situation may arise only for a small time interval after a failure and failure are expected to be rare, hence such blocking will rarely occur. After application processes a message, the current state is included in volatile log. In Section 4.5, a detailed example is given. Delivery of message $m4$ to the application is delayed till the corresponding failure announcement is received.

Check_orphan :
    // Discard orphan messages from the receive and the send buffer.
    Send_buffer := { m ∈ Send_buffer | ¬knows_orphan(s,m)} ;
    Receive_buffer := { m ∈ Receive_buffer | ¬knows_orphan(s,m)} ;

Check_send_buffer :
    // Check and send messages held in Send_buffer, if possible
    ∀ m ∈ Send_buffer: ∀ j: **if** knows_stable(s,m.c[j],j) **then** m.c[j] := NULL ;
    ∀ m ∈ Send_buffer:
        if Number of non-NULL entries in m.c is at most K **then** send m ;

Broadcast_log_prog :
    Broadcast(Log_prog) ;

Log_state :
    Stable_state_list := Stable_state_list ∪ State_list ;
    State_list := {} ;
    Insert(Log_prog[i],c[i]) ;
    Check_send_buffer ;

Figure 7: K-optimistic logging protocol: Routines invoked periodically.

4.3.5 Routines Executed Periodically

We now describe the routines in Figure 7. These routines are executed periodically.

Check_orphan: This routine is called to discard orphan messages from the receive and the send buffers.

Check_send_buffer: This routine updates the dependency vectors of messages in Send_buffer. It is invoked by the events that can announce new stable state intervals, including: (1) *Receive_log_prog* for receiving logging progress notification; (2) *Receive_failure_ann* (according to Corollary 1); and (3) *Log_state*. When a message's dependency vector contains K or less non-NULL entries, it is sent.

Broadcast_log_prog: P_i informs other processes about its logging progress by broadcasting its *Log_prog*. However, logging progress notification is in general less frequent than the logging of states.

Log_state: This routine is called to save volatile states on stable storage.

```
Receive_log_prog(mlog_prog) :
    ∀ j,t : (t,x) ∈ mlog_prog[j] : Insert(Log_prog[j],(t,x)) ;
    ∀ j ≠ i: if knows_stable(s,c[j],j) then c[j] := NULL ;
    // i'th entry is not set to NULL as it will be needed to start the next interval
    Check_send_buffer ;
```

Figure 8: K-optimistic logging protocol: Routine for receiving logging notification.

4.3.6 Handling a Logging Notification

On receiving a logging progress notification, the routine in Figure 8 is called.

Receive_log_prog: Upon receiving a logging notification, a process updates its *Log_prog*. It also sets the stable entries in its dependency vector to NULL. The *Log_prog* is periodically flushed to stable storage. As some part of the *Log_prog* may get lost in a failure, a process needs to collect the logging information from other processes on restarting after a failure.

4.3.7 Handling a Failure

We next describe the routines in Figure 9. These routines are executed in case of a failure.

Restart: On restarting after a failure, P_i restores its last stable state and broadcasts the index of this state as a failure announcement. We assume that the reliable broadcast of a failure includes the execution of the routine *Receive_failure_ann* by all processes. P_i starts a new incarnation by incrementing its incarnation number in the routine *Start_incarnation*.

Receive_failure_ann: On receiving a failure announcement, P_i updates its incarnation end table. As explained in Section 4.2, this announcement also serves as a logging progress notification. P_i also discards orphan messages in Send_buffer and Receive_buffer by calling *Check_orphan* (in Figure 7). If the current state of P_i has become orphan due to this failure, then P_i rolls back by calling *Rollback*.

Rollback: Before rolling back, P_i logs its volatile states in stable storage. Clearly, an implementation will log only the non-orphan states. The highest non-orphan stable state is restored and the orphan states are discarded from stable storage. A new incarnation is started. No rollback announcement is send to other processes, which is a distinctive feature of our protocol.

Start_incarnation: This routine increments the current incarnation number, which is saved in stable storage as the variable *cur_inc*. This ensures that the current incarnation number is not lost in a failure. This routine also updates the dependency vector.

```
Restart :          // after failure
    s := head(Stable_state_list) ;
    Insert(iet[i], c[i]) ;
    Broadcast_failure(c[i]) ;
    Start_incarnation ;

Receive_failure_ann (t,x,j) :      // called by state s on receiving (t,x) from P_j :
    Insert(iet[j],(t,x)) ; Insert(Log_prog[j],(t,x)) ;
    Check_orphan ;
    Check_send_buffer ;
    if knows_orphan(s,s) then Rollback ;

Rollback :
    Log_state ;
    s := maximum{u ∈ Stable_state_list | ¬knows_orphan(s,u) } ;
    Stable_state_list := Stable_state_list - {u ∈ Stable_state_list | s → u} ;
    Start_incarnation ;

Start_incarnation
    cur_inc := cur_inc + 1 ;
    c[i].inc := cur_inc ;
```

Figure 9: K-optimistic logging protocol: Routines involving failure.

4.3.8 Adapting K

Note that there is nothing in the protocol to prevent a change in the value of K. Therefore, the value of K can be changed dynamically in response to changing system characteristics. Also, different processes can have different value of K. A process that is failing frequently may choose to become completely pessimistic by setting its K value to 0 while other processes in system may continue to be optimistic. On the other hand, if the stable storage manager becomes busy, a process may choose to increases its K value.

4.3.9 Output Commit

If a process needs to commit output to external world, it maintains an Output_buffer like the Send_buffer. This buffer is also updated whenever the Send_buffer is updated. An output message is released when all entries in message's dependency vector become NULL. It is interesting to note that an output can be viewed as a 0-optimistic message, and that different values of K can in fact be applied to different messages in the same system. In our implementation described in Section 5, we do not use the Output_buffer. Instead, we attach a K value with each message with 0 being assigned to output messages.

In practice, the concept of K-output commit may also be useful. Although strict output commit may be necessary for military or medical applications, most service-providing applications can revoke an output, if absolutely necessary, by escalating the recovery procedure to a higher level which may involve human intervention. Therefore, K-output commit can be useful to provide a trade-off between the commit latency and the degree of commitment.

4.3.10 Using Checkpoints and Message Logs

```
Log_message :
    Stable_message_list := Stable_message_list ∪ Message_list ;
    Message_list := {} ;
    Insert(Log_prog[i],c[i]) ;
    Check_send_buffer ;

Checkpoint :
    Log_messages ;
    Stable_state_list := Stable_state_list ∪ {s} ;
```

Figure 10: K-optimistic protocol routines that use checkpointing.

A simplification that we have used to clarify the correctness proof is that of replacing checkpointing and message logging with the saving of entire states. In our presentation, we save all states in volatile and stable storage. This is useful only in the unlikely case of the average state size being much smaller than the average message size. Otherwise, an implementation should save the received message instead of the states in volatile memory. Periodically, the current state should be saved on stable storage as a checkpoint. Any state can be reconstructed by restoring the highest checkpoint prior to that state and replaying the messages that have been received between the checkpoint and the state. Instead of a volatile *State_list*, a volatile *Message_list* is used. A *Stable_message_list* is also used. Checkpoints are stored in *Stable_state_list*. Instead of routine *Log_state*, two new routines are used. These routines are given in Figure 10. The old routines that are modified by this implementation strategy are shown in Figure 11.

```
Process_message(m) :
    if admissible(m) then
        c := max(c,m.c) ; c[i].seq := c[i].seq + 1 ;
        Message_list := Message_list ∪ {m} ;
            // Application acts on the message

Restart :          // after failure
    s := head(Stable_state_list) ;
    replay the logged messages that follow;
    Insert(iet[i], c[i]) ;
    Broadcast_failure(c[i]) ;
    Start_incarnation ;

Rollback :
    Log_message ;
    s := maximum{u ∈ Stable_state_list | ¬knows_orphan(s,u) } ;
    Stable_state_list := Stable_state_list - {u ∈ Stable_state_list | s → u} ;
    Replay the messages logged after s in the original receipt order,
        till the current state remains non-orphan;
    Among remaining logged messages, discard orphans and
        add non-orphans to Receive_buffer ;
    Start_incarnation ;
```

Figure 11: Modified K-optimistic logging protocol routines.

So far, we have discussed the design of the K-optimistic protocol. There are a number of implementation issues that have been avoided for clarity. We now take a look at these issues.

4.4 Implementation Notes

There are a number of policy decisions and optimizations that are available to a system designer.

4.4.1 Policy Decisions

1. While broadcasting the logging progress information, a process can choose to broadcast either its own logging information only or the information about all processes that it knows of. Similarly, at the time of failure announcement, logging information about all processes can be broadcast.

2. In general, logging progress need not be broadcast reliably. For a given incarnation, logging progress is monotonic. Therefore, future notification will supply the missing information. However, if an implementation does not broadcast the information about previous incarnations, then in the routine *Start_incarnation*, logging information of previous incarnation needs to be broadcast reliably.

3. We maintain the dependency vector as a vector of n entries. However, dependency vector can also be viewed as a set of triplets of the form (process number, incarnation number, sequence number). Depending on the relative values of K and n, more efficient form should be used.

4.4.2 Optimizations

1. In Figure 11, in routine *Restart*, failure broadcast is done after replaying the messages. An implementation will compute the index of the maximum recoverable state and broadcast it before replaying the messages.

2. In Figure 11, routine *Log_message* is called in the routine *Rollback* to log all unlogged messages. An implementation will log only non-orphan messages.

3. When a process sets its K value to 0, it needs to reliably broadcast a logging progress notification. After that it does not need to send a logging notification as no other process will be commit dependent on its future intervals. With this optimization, our protocol behaves like the pessimistic protocol in [20] for K equal to 0.

4.4.3 Other Issues

In this paper, our focus is on the design of efficient optimistic protocols. There are a number of implementation issues that are not addressed here. We next give a partial list of these issues. These and many other issues are discussed in detail in [11].

- **Failure detection:** In theory, it is impossible to distinguish a failed process from a very slow process [14]. In practice, many failure detectors have been built that work well for practical situations [17]. Most of these detectors use a timeout mechanism.

- **Garbage collection:** Some form of garbage collection is required to reclaim the space used for checkpoints and message logs [33].

- **Stable storage:** Logging protocols require some form of stable storage that remains available across failures. In a multi-processor environment local disk can be used, because as other processors can access the local disk even if one of the processors fails. In a networking environment, the local disk may be inaccessible when the corresponding processor fails. Therefore, a network storage server is required. The storage server itself can be made fault-tolerant by using the techniques presented in [25].

- **Network Address:** When a failed process is restarted, it may have a different network address. Therefore, location independent identifiers need to be used for the purpose of inter-process communication.

- **Environment Variables:** If a failed process is restarted on a processor different from the one used before the failure then some inconsistency may arise due to mismatch of the values of environment variables in pre- and post-failure computation. In such scenario, logging and resetting of environment variables is required.

4.5 A Detailed Example

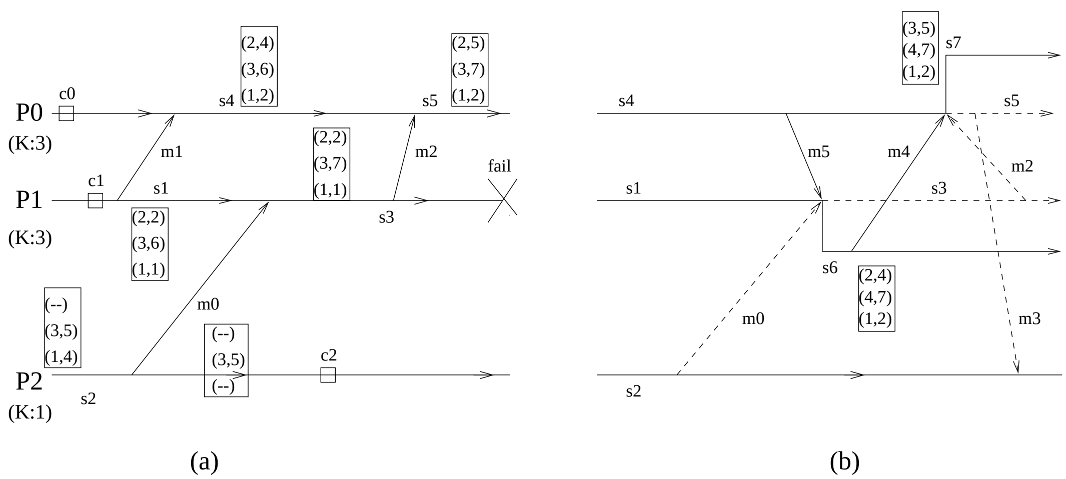


Figure 12: K-optimistic recovery: (a) Pre-failure computation (b) Post-failure computation

Figure 12 shows an example of the protocol execution. Dependency vectors are shown only for some states and messages. To avoid cluttering the figure, some messages causing state transitions are not shown. The K value for $P0$ and $P1$ is 3 and that for $P2$ is 1.

In Figure 12(a), $P1$ sends the message $m1$ to $P0$ in the state interval $s1$. $P0$ processes this message, starts the state interval $s4$ and sends a message $m5$ to $P1$ ($m5$ is shown in the Figure 12(b) only). In the state interval $s2$, $P2$ sends the message $m0$ to $P1$. However, the recovery layer delays the sending of $m0$ as it is dependent on two non-stable intervals. The message is sent after $P2$ makes its own interval (1,4) stable. $P1$ processes this message and sends the message $m2$ to $P0$. It performs some more computations and fails (shown by a cross). At the time of failure, it has logged all the messages received till the interval $s1$ and has not logged the message $m0$. During this time, $P0$ acts on the messages $m2$ and starts the interval $s5$.

The post-failure computation is shown in the Figure 12(b). On restart, $P1$ restores its last checkpoint $c1$, replays the logged messages and recreates the interval $s1$. It broadcasts the failure announcement (3,6) to other processes and increments its own incarnation number. $P1$ now processes message $m5$ resulting in the interval $s6$. $P1$ sends message $m4$ to $P0$. During this time, $P0$ sends the message $m3$ to $P2$. The recovery layer of $P0$ receives the message $m4$ before it receives the failure announcement from $P1$. Note that the message $m4$ is received by the recovery layer in state $s5$, but it is not delivered to the application. In the figure, arrows point to the state in which a message is delivered and not the state in which they are received. The second entry in dependency vector of $m4$ is $(4,7)$, while the second entry in $P0$'s dependency vector in state $s5$ is $(3,7)$. Therefore, $P0$ decides that $m4$ is inadmissible. Later, when $P0$ receives the failure announcement in state s5, it inserts the entry $(3,6)$ in its $iet[1]$ (see Figure 4.3.6). Then the predicate $knows\_orphan(s5, s5)$ becomes true for $j = 1, t = 3$, and $x = 6$. Hence $P0$ rolls back. $P0$ restores the checkpoint $c0$ and replays the logged messages, until, in state $s4$, it is about to process the message $m2$ that made it orphan. It discards $m2$ and increments its incarnation number. It does not send out any rollback announcement. Now message $m4$ is processed and interval $s7$ is started. On receiving message $m3$, $P2$ detects that $m3$ is orphan and discards it.

4.6 Variations of the Basic Protocol

The K-optimistic protocol presented in previous sections is one of the possible applications of Theorem 3. This theorem can be used to implement many different policies. For example, P_i may be unwilling to roll back due to a failure of P_j. This can be enforced by P_i by blocking the delivery of any message that is commit dependent on any interval of P_j till that interval becomes stable. Interestingly, P_i may choose to become commit dependent on P_k while avoiding commit dependency on P_j, even though P_k may become commit dependent on P_j. This is because, on receiving a message from P_k, P_i can detect that P_k is passing an unstable dependency on P_j. That message delivery can be blocked by P_i till the dependency on P_j becomes stable.

Simulating a Failure

As discussed in Section 1, there are many scenarios like non-crash failures and software error recovery, where recreating pre-failure states is undesirable. This poses a problem for our protocol, because we are setting stable entries to NULL under the assumption that they can never be lost in a failure. But, now we need to simulate the loss of stable intervals. To do this, we add one more bit (initially 0) to each entry in the dependency vector. Instead of setting an entry to NULL, we simply set the corresponding bit to 1. Lexicographic comparison operation still remains the same. Incarnation numbers and state indices are compared to determine the maximum of two entries. Everything else in the protocol remains the same except that for the purpose of orphan detection, all entries including the stable ones need to be inspected. For example, suppose the second entry of $P1$ dependency vector is (2,6,0). It corresponds to entry (2,6) in the old notation. Now $P1$ receives the logging notification (2,8) from $P2$. Instead of setting (2,6,0) to NULL, it is changed to (2,6,1). Later on, if $P2$ were to simulate a failure and send the announcement (2,4), $P1$ will know that it is an orphan by comparing (2,4) to the entry (2,6,1) in its dependency vector.

There is a clear limitation of this approach. It does not work when an entry from a lower incarnation is overwritten by an entry from a higher incarnation. This means that failures can be simulated only within the current incarnation.

An alternative approach to failure simulation is that in addition to logging on stable storage, an application may also need to satisfy some other conditions before it can declare an interval stable. For example, with latent errors, an interval becomes stable only after the maximum error detection latency period.

5 Experimental Results

So far we have mainly discussed the theoretical issues related to K-optimistic logging. This section presents the experimental results of a prototype implementation.

To our knowledge, there is no general answer to the question: What value of K should one use? It depends entirely on the application characteristics and the application. Some of the factors that play a crucial role are: communication pattern, message size distribution, message arrival rate, network bandwidth, stable storage server load, and failure probability. Given the wide range of these parameters, it is not possible to come up with a table showing the failure-free overhead and the recovery time for combinations of particular values of these parameters. Instead, we recommend that a prototype of the application be run with different values of K and be tested for different failure scenarios. Based on the observed behavior and the application requirements regarding maximum down time and failure-free overhead, appropriate value of K can be chosen. In the following sections, we discuss some particular applications and present the failure-free overhead and recovery time for the single failure case.

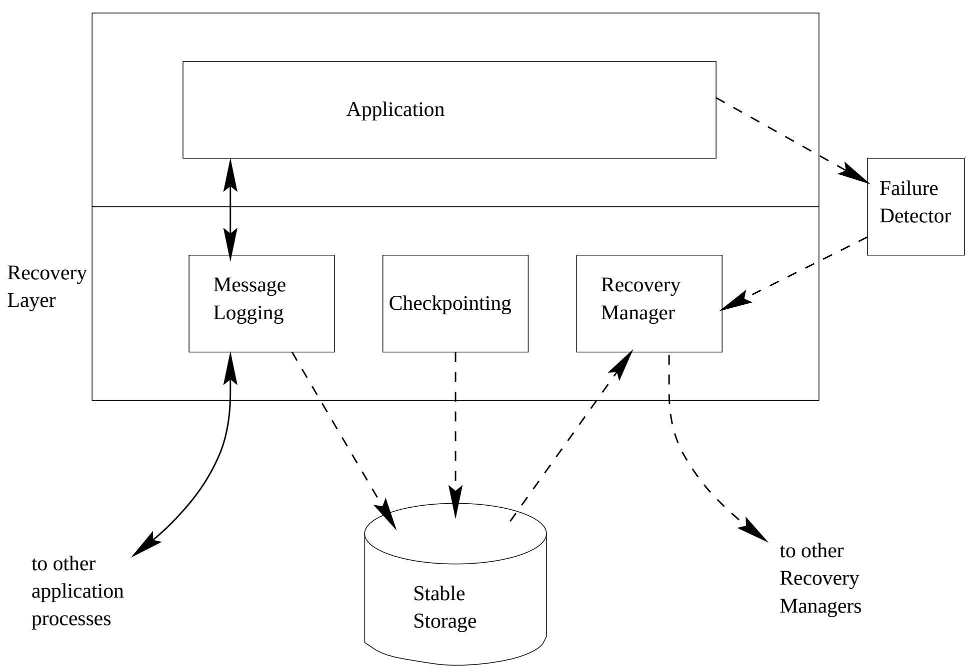


Figure 13: Recovery layer architecture.

5.1 Architecture

We have implemented a prototype of the K-optimistic protocol. Our architecture is shown in the Figure 13. An application is compiled with a recovery library that consists of a logging server, a checkpointing server and a recovery manager. Solid arrows show the flow of application messages while the dashed arrows show the messages required for the fault-tolerance. Application should periodically send an *I-am-alive* message to the failure detector. As per the diagram, recovery manager and application belong to the same process. Therefore, on detecting many consecutive missing I-am-alive message, the failure detector will start a new recovery manager which will load the latest checkpoint and pass the control to the application. Since our focus is on the message logging part, we have not implemented the checkpointing server and the failure detector. We simulate them appropriately, as explained in the Section 5.5.

5.2 Message Logging Policy

In traditional optimistic schemes, the volatile log is periodically flushed to stable storage. However logging at fixed interval does not work well for the K-optimistic scheme. This is because for the lower values of K, the logging needs to take place more often to give acceptable performance. For example, consider the case of K being 0. If messages are logged at a fixed interval of say 300 millisecond, then no process can send out messages faster than once in 300 millisecond.

The application progress is affected in a non-linear fashion with the varying logging frequency. Higher logging frequency may result in non-optimal use of the file server. Also, the application and the logging server may compete for the processor cycles and the network bandwidth. On the other hand, lower logging frequency may result in messages being held in *Send_buffer* for a long time. This implies that for a given value of K, one needs to experiment with different values of logging frequency to select the optimal value. However, this method of determining logging frequency does not work in presence of different message sizes, changing message arrival rates and varying system load.

To solve this problem, we have designed a novel message logging policy. Our policy asynchronously logs the very first message, right after it is received. After that, whenever a notification from the file server is received that the previous logging completed successfully, all the received messages since the previous logging are submitted to the file server for asynchronous logging. This policy automatically adapts to the changing system load. For a lightly loaded system, messages will be logged frequently. As the system load increases, logging frequency decreases correspondingly.

For K value of n, above logging policy is similar in spirit to the logging policy used in traditional optimistic protocols. Even for K value of 0, this policy works like the pessimistic protocol in [20]. In that protocol, the logging overhead is reduced by delaying the logging till the point where a message dependent on unlogged messages needs to be sent.

A related issue is that of logging progress notification frequency. It offers trade-offs similar to those discussed for the logging frequency. However, as the size of the logging notification message is much smaller than a typical application message (8 to $8n$ bytes, depending on the implementation), frequent notification results in negligible overhead compared to frequent logging. We also piggyback the logging progress information about the highest known incarnation of each process on every outgoing message. We found that this adds very little to the message processing overhead but helps in fast logging notification.

We have chosen a period of 500 millisecond for the logging notification, except when K equals n, for which notification period is 1 second. In the latter case, logging notification is needed only for the output commit and not for the progress of the computation.

5.3 Test Scripts

In our experiments, each process receives a message, sleeps for a while, sends a message and then blocks for the next message receive. In the beginning, an initiator process sends a message to all other processes. For a given experiment, the message size is fixed but compute time is chosen uniformly from a range. Compute time is the time between the processing of a message and the send of next message. It is inversely related to message frequency.

5.4 Application Parameters

We vary following parameters in our tests: message size, compute time and communication pattern. We consider these parameters because they are the main determinant of the trade-off provided by the K-optimistic protocol. The trade-off depends on two factors: how fast messages can be logged and how fast messages need to be logged. How fast messages can be logged depends on the message size. How fast the messages need to be logged depends on the message frequency. We later show that communication pattern also determines how fast messages can be logged.

5.4.1 Communication Pattern

We have tested our protocol for two different communication patterns. In test *Random*, the receiver of a message is chosen randomly by the sender, whereas in test *Neighbor*, processes are arranged in a ring and they alternately send messages to their left and right neighbors only. Tests *Random* and *Neighbor* were chosen because they are representatives of the many different applications studied in the literature [13, 29]. They represent two extreme communication patterns for distributed applications and if our protocol works well for these extremes, then it should work well for the patterns in between the extremes.

In *Random*, each process receives messages from all other processes. Therefore, if a single process were to fail, other processes will not slow down much as they will still be able to communicate with each other. On the other hand, in *Neighbor*, failure of a single process changes the topology from a ring to a doubly-linked list. The neighbors of the failed process are expected to slow down as their message intake is reduced by half. As a result, other neighbors of these neighbors will also slow down and so on, resulting in a slowdown of the entire application. In *Neighbor*, all processes receive equal number of messages and so they block and compute for approximately equal periods. On the other hand, in *Random*, some processes receive a little more messages than average while others receive a little less. This implies that some process may be blocked waiting for a message to process while some other process may always have a message to process when it needs one.

5.4.2 Message Size and Compute Time

We have selected specific message sizes and compute time to illustrate a wide range of applications. If the message logging time for most messages is less than the minimum computation time, then most of the messages will be logged before the application finishes processing them. As a result, very few messages will get lost in a failure and the recovery time will be dominated by the checkpoint restoration and the message replay time for the failed process. Therefore, the recovery time for different values of K should be similar. Also, during failure-free computation, very few messages should be held in the send buffer. Therefore, the overhead for different values of K should be similar and little. This overhead can be made arbitrarily small by selecting very high compute times. If the message logging time for most messages is more than the maximum compute time, then the overhead for lower values of K can be arbitrarily large depending on the actual values of logging and computation time.

For message size of 1K, compute time of 80 to 100 millisecond is more than the average logging time for both the tests. For message size of 10K, compute time of 80 to 100 millisecond is less than the average logging time for both the tests. For message size of 4K, compute time of 50-70 millisecond is less than the average logging time for *Random* and more than the average logging time for *Neighbor*.

5.5 Performance Evaluation

5.5.1 Performance Metrics

We measure the failure-free overhead by running the test with and without K-optimistic logging for different values of K. We measure recovery time as the difference in the average values of execution time without any failure and the execution time with a single failure. This definition implies that the recovery time may be similar for different values of K, even though the number of processes rolling back are different. This is because processes may roll back concurrently. Also, when one process rolls back, other processes may block. Therefore, the recovery time may change little if the blocked processes were to actually roll back a little.

5.5.2 Experimental Settings

We use a network of IBM-250/25T workstations connected by a 10Mb/s Ethernet. Each workstation runs AIX 4.1.5 and is equipped with a 66-MHz PowerPC 601 processor, 32KB of data cache and 256MB of memory. A highly available NFS is used for stable storage.

5.5.3 Measurement Methodology

All measurements are made with applications distributed across 4 machines with 2 processes per machines. All tests involved 20 trials. All results presented here are averages of middle 10 values for each test. The duration of a trial for the case of no message logging ranged from 24 to 40 seconds for different combinations of test parameters. Standard deviations for most measurements are under 1% of the average.

The occurrence of a failure is simulated after a process has processed 30 messages since a designated checkpoint. The latency of failure detection is heavily dependent on the implementation of the failure detector and the application environment. Since we consider only the single failure case, we have chosen to ignore this latency in comparing the recovery time for different values of K. Since the checkpoint size in our tests is very small, we simulate the cost of restoring a large checkpoint and reading the message log by blocking the application for 2 seconds while recovering from a failure. The value of 2 second was chosen based on the values reported in [29].

5.5.4 Results

The results of our experiments are shown in Figures 14 and 15. Part (a) of the figures shows the failure-free overhead in percentage terms for various values of K. Part (b) shows the recovery time in seconds for the corresponding values of K. First entry in the legends is an experiment number that we use to discuss the results. Next two entries show the message size in bytes and the range in millisecond from which computation time for each message is uniformly chosen.

The failure-free overhead for traditional optimistic ($K = 8$) and pessimistic ($K = 0$) logging ranges from 6-14% and 8-66% respectively. These ranges are completely arbitrary and they can be much larger or smaller, depending on the application characteristics. For example, in [36], overhead of 3% is reported for completely pessimistic logging with a compute time of 2 seconds. The recovery time measurement should be used for illustration only, and not as the absolute values since we simulate the failure.

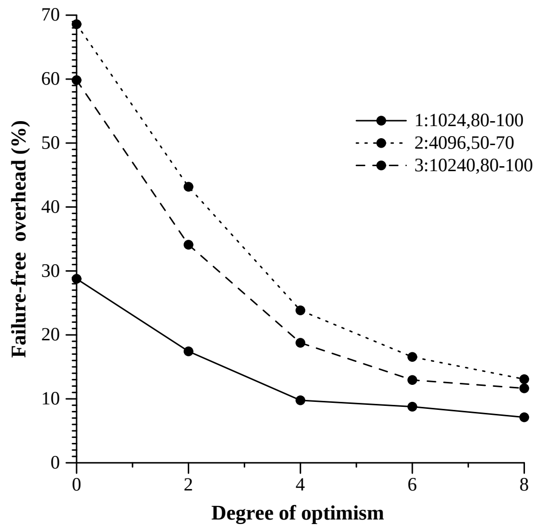
5.5.5 Discussion

Both failure-free overhead and recovery time graphs for experiment 4 are almost flat. This is to be expected, since in this experiment, almost all messages are logged before the application finishes processing them. Same logging pattern implies same failure-free overhead. Since no messages are lost in a failure, recovery time is also same for different values of K. In general, whenever compute time is much more than the average message logging time, we should expect the failure-free overhead and the recovery time graphs to be flat.

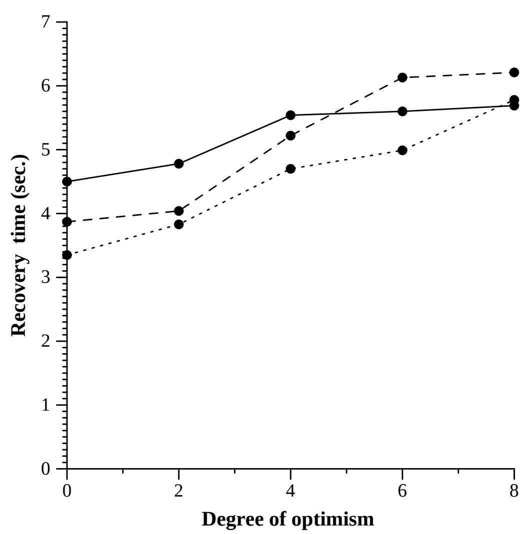
All other parameters being equal, the failure-free overhead for *Neighbor* is always more than that for *Random*. In *Neighbor*, all processes block and compute for similar period of time. Therefore, there are times when no process is trying to log messages while at other times, many processes try to log messages concurrently. Compared to this, access to stable storage is more uniformly distributed over time for *Random*. Another reason is that in *Neighbor*, if one process is waiting for the current logging to finish to release its messages from the send buffer, then its neighbors also slow down. However, in *Random*, neighbors can receive messages from any other processes and can make progress.

We also note that the recovery time for *Neighbor* is always more than the recovery time for *Random*. This is because while a process is recovering, its neighbors receive messages only from their other neighbor, while in *Random*, they receive messages from many other processes.

For lower values of K, the time to restore the checkpoint and replay the message logs is always greater than the recovery time, which is defined as the difference in the running time of a failure-free run and a single-failure run. This is because while a process is recovering, other

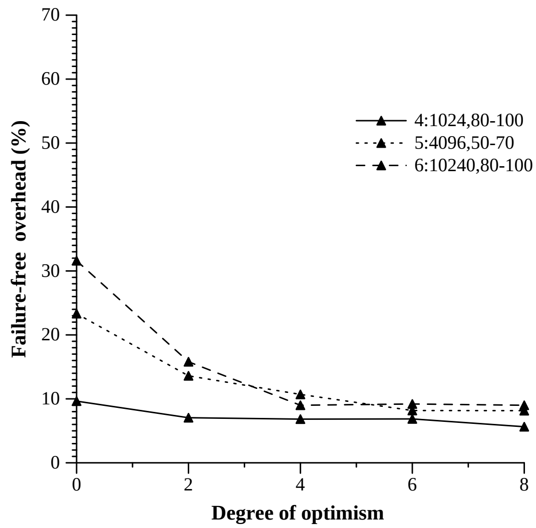


(a)

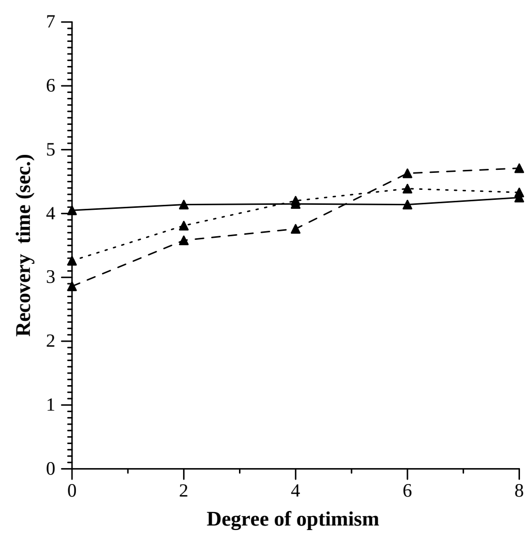


(b)

Figure 14: Performance results for application *Neighbor*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.



(a)



(b)

Figure 15: Performance results for application *Random*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.

processes are making progress. Also, other processes are sending messages to the recovering process. Therefore, when the failed process finishes its replay, it will have many more messages to process without blocking for the want of a message to process.

Another interesting trend is that for the lower values of K and the same value of compute time, computation with smaller message size takes longer to recover. This is contrary to what one would expect if one were to define recovery time as the time to restore the checkpoint plus the time to replay the message logs. We explain this with reference to the experiments 1 and 3. But before that, let us understand the effect of a failure on an application completion time.

In general, processes compute most of the time and when they have no messages to process, they remain idle. When a failed process is recovering, it is also receiving messages from other processes. After the end of the replay, the failed process acts on these received messages and does not remain idle for quite some time. In this period, other processes receive message from the failed process at a rate faster than normal. As a result, overall computation proceeds at a rate faster than normal. More time a computation takes to finish in a failure-free run, less is the increase in completion time caused by a failure because the computation has more time to adjust to the disturbance caused by a failure.

Since messages of size 1K are logged faster than messages of size 10K, in the absence of any failure, experiment 1 takes less time to complete than experiment 3. Experiment 1 takes 37 seconds whereas experiment 3 takes 49 seconds. A failed process takes almost same time (5 seconds) to restore a checkpoint and replay messages in both experiments. Compared to experiment 1, experiment 3 has more time to adjust to the disturbance caused by this blocking. Therefore, the extra time taken to finish is more in experiment 1.

For higher values of K and same compute time, recovery time is lower for experiments with lower message sizes. This is expected because for lower message sizes, fewer messages are lost in a failure and as a result, fewer processes roll back compared to higher message sizes.

The failure-free overhead varies with varying message sizes in an intuitive manner. For *Random*, the failure-free overhead does not increase much as the K changes from 8 to 4. This is probably because most messages are dependent on at most 4 non-stable intervals. Therefore, for K value of 4 to 8, most messages are never held in the send buffer, resulting in similar failure-free overhead. However, logging progress in *Neighbor* is slow compared to *Random* for the reasons discussed earlier. Therefore, for experiments 2 and 3, failure-free overhead changes as K changes from 8 to 4. For experiment 1, small message size results in little change of failure-free overhead for higher values of K.

Finally, note that the configurations that give similar failure-free performance (5,6) give different recovery characteristics.

5.6 Selecting K

At the beginning of this section, we proposed that to select the appropriate value of K for a given application, a prototype of the application should be run with different values of K and failure-free overhead and recovery time graphs should be obtained. After that, K can be chosen based on the constraints on the failure-free overhead or recovery time. For example, let us consider the results of the experiment 2, shown in Figure 14. If the system designer wants to minimize

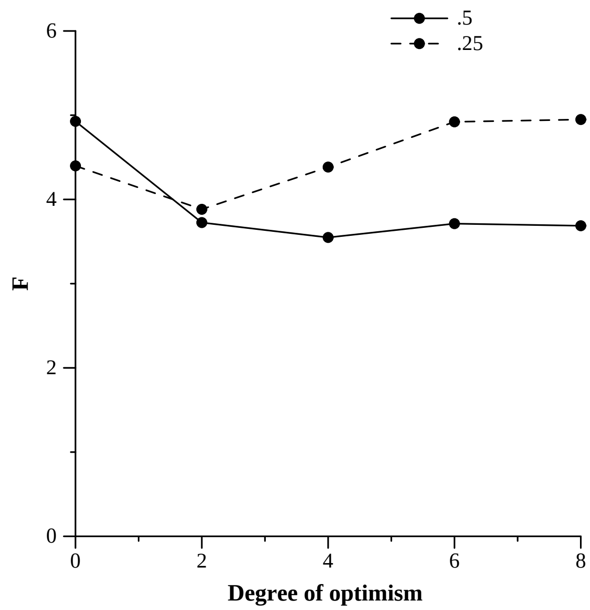


Figure 16: Selecting a K value for a given α. Numbers in the legend represent the α values.

the recovery time while keeping failure-free overhead under 20% then he can choose K equal to 6. On the other hand if the goal is to have the maximum recovery time of 4 seconds while minimizing the failure-free overhead then K value of 2 can be used.

Another approach is to minimize an objective function of the failure-free overhead and recovery time. For example, let the objective function F be:

$$F(O, R) = \alpha O + (1 - \alpha)R$$

Arguments O and R represent the failure-free overhead and the recovery time in some normalized form and α is a parameter to be chosen by the system designer. As α changes from 0 to 1, we should expect the optimum K value to change from 0 to n. In Figure 16, we plot the function F for two different values of α. Argument O is obtained by dividing the failure-free overhead for experiment 2 by 10% and argument R is taken as recovery time. As both curves have a unique minimum, we select the K value of 4 for α equal to .5 and K value of 2 for α equal to .25.

6 Conclusion

We have proved two fundamental results. First, with transitive dependency tracking, only the failures and not all rollbacks need to be announced. Second, only the dependencies on non-stable intervals need to be tracked. Based on these results, we have introduced the concept of K-optimistic logging that allows a system to explicitly fine-tune the trade-off between failure-free overhead and recovery efficiency. In such a system, given any message, the number of processes whose failures can revoke the message is bounded by K, and therefore K indicates the maximum amount of risk that can be placed on each message or equivalently the degree of optimism in the system. Traditional pessimistic logging and optimistic logging then become the two extremes in the spectrum spanned by K-optimistic logging.

References

[1] L. Alvisi. Understanding the Message Logging Paradigm for Masking Process Crashes. Ph. D. Thesis, Dept. of Computer Science, Cornell University, January 1996.

[2] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. *Proc. 23rd Fault-Tolerant Computing Symp.* , 145-154, 1993.

[3] M. Ahamad and L. Lin. Using Checkpoints to Localize the Effects of Faults in Distributed Systems. *Proc. 8th Symp. on Reliable Distributed Systems*, 66-75, 1989.

[4] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. *Proc. 9th ACM Symp. on Operating Systems Principles*, 90-99, 1983.

[5] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Trans. Comput. Syst.*, 7(1):1-24, February 1989.

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

[7] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, March 1996.

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

[9] O. P. Damani and V. K. Garg. Fault-Tolerant Distributed Simulation. *Proc. 12th Workshop on Parallel and Distributed Simulation*, 38-45, 1998.

[10] O. P. Damani, Y. M. Wang and V. K. Garg. Optimistic Distributed Simulation Based on Transitive Dependency Tracking. *Proc. 11th Workshop on Parallel and Distributed Simulation*, 90-97, 1997.

[11] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. em ACM Computing Surveys, 34(3):375-408, Sept. 2002.

[12] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Trans. on Computers*, 41(5):526-531, May 1992.

[13] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. In *Proc. 24th IEEE Fault-Tolerant Computing Symp.*, 298-307, 1994.

[14] M. J. Fischer, N. Lynch and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2), 374-382, April 1985.

[15] V. K. Garg and A. I. Tomlinson. Using Induction to Prove Properties of distributed Programs. *Proc. 5th IEEE Symp. on Parallel and Distributed Processing*, 478-485, 1993.

[16] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 3, pp. 299-307, March 1994.

[17] Y. Huang and C. Kintala. Software Implemented Fault-Tolerance: Technologies and Experience. *Proc. IEEE Fault-Tolerant Computing Sym.*, 2-9, 1992.

[18] Y. Huang and Y. M. Wang. Why Optimistic Message Logging has not been Used in Telecommunications Systems. *Proc. IEEE Fault-Tolerant Computing Symp.*, 459-463, June 1995.

[19] D. R. Jefferson. Virtual Time. *ACM Trans. Prog. Lang. and Sys.*, 7(3):404-425, 1985.

[20] P. Jalote. Fault Tolerant Processes. *Distributed Computing*. 1989.

[21] D. B. Johnson. Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 86-95, 1993.

[22] D. B. Johnson and W. Zwaenpeoel. Recovery in Distributed Systems using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11:462-491, Sept. 1990.

[23] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, 558-565, 1978.

[24] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic Failure Recovery for Very Large Networks. *Proc. IEEE Symposium on Reliable Distributed Systems*, 66-75, 1991.

[25] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System. Unpublished Technical Report, Xerox Palo Alto Research Center. Apr. 1979. *http://research.microsoft.com/users/blampson/Publications.html*

[26] F. Mattern. Virtual Time and Global States of Distributed Systems. *Parallel and Distributed Algorithms: Proc. of the Intl. Workshop on Parallel and Distributed Algorithms, Elsevier Science Publishers B. V.(North Holland)*, 215-226, 1989.

[27] S. L. Peterson and P. Kearns. Rollback Based on Vector Time. *Proc. 12th IEEE Symp. on Reliable Distributed Systems*, 68-77, 1993.

[28] M. L. Powell and D. L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. *Proc. 9th ACM Symp. on Operating Systems Principles*, 100-109, 1983.

[29] S. Rao, L. Alvisi and H. Vin. The Cost of Recovery in Message Logging Protocols. Tech. Rep. No. TR-98-02, Dept. of Computer Sciences, Univ. of Texas at Austin, 1998.

[30] S. W. Smith and D. B. Johnson. Minimizing Timestamp Size for Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. *Proc. 15th Symp. on Reliable Distributed Systems*, 66-75, 1996.

[31] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. *Proc. 25th Intl. Symp. on Fault-Tolerant Computing*, 361-370, 1995.

[32] A. P. Sistla and J. L. Welch. Efficient Distributed Recovery Using Message Logging. *Proc. 8th ACM Symp. on Principles of Distributed Computing*, 223-238, 1989.

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

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

[35] Y. M. Wang and W. K. Fuchs. Lazy Checkpoint Coordination for Bounding Rollback Propagation. *Proc. IEEE Symposium on Reliable Distributed Systems*, 78-85, 1993.

[36] Y. M. Wang, Y. Huang, W. K. Fuchs, C. Kintala, and G. Suri. Progressive Retry for Software Failure Recovery in Message-Passing Applications. *IEEE Trans. on Computers*, 46(10):1137-1141, Oct. 1997.

Appendix: Properties of the Protocol

In this section, we prove our protocol correct and discuss its properties. All the lemmas and invariants presented before the Theorem 4 are used to for the correctness proof in the Theorem 4. The Theorem 5 shows that our protocol does not indefinitely postpone the sending of a message. The Theorem 6 establishes the meaning of K, i.e., given any message m released by its sender, the number of processes whose failure can revoke m is at most K.

In Figure 4, the function *get_state* returns a state interval of a given process corresponding to a given entity. This function is well defined only if the entries corresponding to the states of a process are unique. The following observation shows that the function *get_state* is indeed well defined.

Observation 1 *Different states of the same process have different entries.*

Proof. Whenever a new state is started in the routine *Process_message*, sequence number is incremented. In *Start_incarnation*, incarnation number is incremented. Storing *cur_inc* on stable storage prevents the loss of incarnation information in a failure. A failure before the completion of the routine *Start_incarnation* will result in the restoration of the same state. Therefore, reuse of the incarnation number does not matter. ■

The converse of this lemma is not true. A state can have more than one entry. This happens to a state restored after a failure or rollback when the state's incarnation number is incremented.

The following lemma shows that stability is monotonic with respect to the happened before relation within the same process.

Lemma 2 $(s \underline{\rightarrow} w \wedge s.p = w.p \wedge stable(w)) \Rightarrow stable(s)$

Proof. First note that whenever an interval is started, it is added to the *State_list*. Also note that either *State_list* is empty or all the states in it belong to the same incarnation and are consecutive. This is because routine *Log_state* is called in the routine *Rollback*. Lemma is trivially true when s is same as w. Now we prove the lemma by induction on the number of states between distinct s and w.

Base Case (0 states): If s is not made stable in the call to *Log_state* that made w stable, then s must have been made stable in an previous call to *Log_state*.

Induction Step: Let $s \rightarrow w$. Let u be the state immediately preceding w. Now $stable(w)$ implies $stable(u)$ by induction hypothesis. Also, $stable(u)$ implies $stable(s)$ by induction hypothesis. Hence the result. ■

The following lemma shows that the predicate *knows_stable* correctly detects a stable state.

Lemma 3 $knows\_stable(s, e, j) \Rightarrow stable(get\_state(j, e))$.

Proof. A state enters its entry into its *Log_prog* in the routine *Log_state*. It does so, only after adding the current state to the stable state list. Therefore, lemma is true when $s.p$ is equal to j. Now we consider the case when $s.p$ is different from j.

$knows\_stable(s, e, j)$

$\Rightarrow$ { definition of *knows_stable* }

$seq\_num(s.Log\_prog[j], e.inc) \geq e.seq$
$\Rightarrow$ { Definition of seq_num }
$\exists x : (e.inc, x) \in s.Log\_prog[j] \ \wedge \ x \geq e.seq$
$\Rightarrow$ { e belongs to $s.Log\_prog[j]$ implies that P_j has broadcast it. }
$\exists w : w.p = j \ \wedge \ w.c[j] = (e.inc, x) \ \wedge \ stable(w)$
$\Rightarrow$ { Definition of get_state }
$\exists w : stable(w) \ \wedge \ w.p = j \ \wedge \ get\_state(j, e).c[j].inc = w.c[j].inc$
$\wedge \ get\_state(j, e).c[j].seq \leq w.c[j].seq$
$\Rightarrow$ { Lemma 2 }
$stable(get\_state(j, e))$ ■

The following lemma proves a similar result in the other direction. Before we continue, we need one more predicate definition. A set or an entry satisfies the predicate *broadcast* if it has been broadcast by some state.

Lemma 4 $stable(get\_state(j, e)) \Rightarrow \forall i : \exists s \in P_i : knows\_stable(s, e, j)$

Proof. Let w be the state returned by $get\_state(j, e)$. Let u be the first state of P_j after w in which *Log_state* is called. We next prove that w and u have the same incarnation number. A new incarnation is started only due to a failure or a rollback. By our choice of u, a failure before u means that w is not stable. A rollback before u means that u is not the first state after w in which *Log_state* is called as *Log_state* is called in *Rollback*. Therefore, u and w have same incarnation number. Now,

$stable(w)$
$\Rightarrow$ { Above argument, *Log_state* is atomic }
$u.c[j].inc = e.inc \ \wedge \ u.c[j].seq \geq e.seq \wedge u.c[j] \in u.Log\_prog[j]$
$\Rightarrow$ { Future insertions maintain the following property }
$\forall v : v.p = j \ \wedge \ u \underline{\rightarrow} v : [\exists x : (e.inc, x) \in v.Log\_prog[j] \wedge x \geq e.seq \ ]$
$\Rightarrow$ { *Broadcast_log_prog* is called eventually. }
$\exists Log\_prog : broadcast(Log\_prog) : [\exists x : (e.inc, x) \in Log\_prog[j] \wedge x \geq e.seq]$
$\Rightarrow$ {*Receive_log_prog* is called eventually }
$\forall i : \exists s \in P_i : \exists x : (e.inc, x) \in s.Log\_prog[j] \wedge x \geq e.seq$
$\Rightarrow$ { Definition of seq_num }
$\forall i : \exists s \in P_i : seq\_num(s.Log\_prog[j], e.inc) \geq e.seq$
$\Rightarrow$ { Definition of $knows\_stable$ }
$\forall i : \exists s \in P_i : knows\_stable(s, e, j)$ ■

The following invariant is at the heart of our protocol. It ensures that dependencies on unstable states are never set to NULL. Therefore when an unstable state is lost in a failure, this invariant helps in the detection of orphan states.

Invariant 1 $\forall s, u : (s \underline{\rightarrow} u \ \wedge \ \neg stable(s)) \Rightarrow ((u.c[s.p].inc = s.c[s.p].inc)$
$\wedge \ (u.c[s.p].seq \geq s.c[s.p].seq))$

Proof. Invariant holds trivially if s is same as u. Therefore, we consider s, such that s happened before u. We show that the above invariant holds initially. We also show that the invariant holds after the execution of a routine, if it holds before the routine is called. It is sufficient to consider the routines that create a new interval or modify the dependency vector.

Initialize: Invariant is trivially true, because for any initial state u and any state s, $s \not\rightarrow u$.

Process_message: Let the message m be sent by an interval w. Let interval v process m and start interval u. Incrementing the sequence number in the second operation in the routine leaves the invariant unaffected. Therefore, it is sufficient to show that the invariant holds after the *max* operation.

In general, dependency vector of w and m can be different as some entries in the dependency vector of m might be set to NULL in the routine *Check_send_buffer*. Happened before relation is defined between w and u, whereas dependency vector of u is updated by taking maximum of dependency vectors of v and m. In order to reason about happened before relation, sometimes we would like to use the dependency vector of w instead of dependency vector of m. The following claim shows that dependency vectors of m and w agree on the dependencies on the non-stable intervals.

Claim 1: $s \underline{\rightarrow} w \ \wedge \ \neg stable(s) \Rightarrow m.c[s.p] = w.c[s.p]$

Proof. $s \underline{\rightarrow} w \ \wedge \ \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine is called }

$(w.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (w.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { Definitions of *get_state* and happened before }

$s \underline{\rightarrow} get\_state(s.p, w.c[s.p])$

$\Rightarrow$ { $\neg stable(s)$, Lemma 2 }

$\neg stable(get\_state(s.p, w.c[s.p]))$

$\Rightarrow$ { Lemma 3 }

$\forall r : \neg knows\_stable(r, s.p, w.c[s.p])$

$\Rightarrow$ { First if condition in *Check_send_buffer* is not satisfied }

$m.c[s.p] = w.c[s.p]$ { End of Claim }

Now, $s \rightarrow u$

$\Rightarrow$ {By definition of happened before}

$s \underline{\rightarrow} v \vee s \underline{\rightarrow} w$ ... (I)

We consider the following three cases separately:

1) $v.c[s.p].inc = m.c[s.p].inc$
2) $v.c[s.p].inc > m.c[s.p].inc$
3) $v.c[s.p].inc < m.c[s.p].inc$

Case 1: $v.c[s.p].inc = m.c[s.p].inc$. From (I), there are two subcases to consider: 1.a) $s \underline{\rightarrow} v$, 1.b) $s \underline{\rightarrow} w$.

Subcase 1.a: $s \underline{\rightarrow} v \ \wedge \ \neg stable(s)$

$\Rightarrow$ { Invariant holds before this routine is called. }

$(v.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (v.c[s.p].seq \geq s.c[s.p].seq)$

$\Rightarrow$ { $u.c[s.p] = max(v.c[s.p], m.c[s.p])$, case 1 }

$(u.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (u.c[s.p].seq \geq s.c[s.p].seq)$

Subcase 1.b: By Claim 1, $m.c[s.p]$ is equal to $w.c[w.p]$. The rest of the proof is similar to the subcase 1.a.

Case 2: $v.c[s.p].inc > m.c[s.p].inc$. From (I), there are two subcases to consider: 2.a) $s \underline{\rightarrow} v$, 2.b) $s \underline{\rightarrow} w$.
Subcase 2.a: It is similar to subcase 1.a.

Subcase 2.b: $s \underline{\rightarrow} w \ \wedge \ \neg stable(s)$
$\Rightarrow$ { Invariant holds before this routine is called }
$(w.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (w.c[s.p].seq \geq s.c[s.p].seq)$
$\Rightarrow$ { Claim 1: $m.c[s.p] = w.c[s.p]$ }
$(m.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (m.c[s.p].seq \geq s.c[s.p].seq)$
$\Rightarrow$ { $admissible(m, v)$, case 2 }
$(m.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (m.c[s.p].seq \geq s.c[s.p].seq)$
$\wedge \ knows\_stable(v, m.c[s.p], s.p)$
$\Rightarrow$ { Definition of $knows\_stable$ }
$knows\_stable(v, s.c[s.p], s.p)$
$\Rightarrow$ { Lemma 3 }
$stable(s)$

Case 3: $v.c[s.p].inc < m.c[s.p].inc$. Proof is similar to that of case 2.
This concludes the proof for the routine *Process_message*.

Start_incarnation: Let u be the state in which this routine is called. This routine is called inside routines *Restart* and *Rollback* only. Therefore, u is stable. We consider the following two cases: 1) $s.p = u.p$, 2) $s.p \neq u.p$.
Case 1: $s.p = u.p$
$s \rightarrow u \Rightarrow s \underline{\rightarrow} u$
$\Rightarrow$ { u is stable, stability is monotonic }
$stable(s)$

case 2: $s.p \neq u.p$
$s \rightarrow u$
$\Rightarrow$ { Antecedent: $\neg stable(s)$ }
$s \rightarrow u \ \wedge \ \neg stable(s)$
$\Rightarrow$ { Invariant holds before this routine, $s.p \neq u.p$, only $u.c[u.p]$ is modified in this routine }
$(u.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (u.c[s.p].seq \geq s.c[s.p].seq)$

Receive_log_prog: Let u be the interval calling this routine. Modification of dependency vector of u can cause violations of invariant involving two kinds of states: 1) $s \rightarrow u$, 2) $u \rightarrow s$. As $u.p$'th entry is not modified in this routine, so it is sufficient to consider the first case. We need to consider state s only if the $s.p$'th entry is modified in this routine.
Now, $s.p$'th entry is modified.

$\Rightarrow$ { Test in the if condition }
$knows\_stable(u, u.c[s.p], s.p)$
$\Rightarrow$ { Antecedent, invariant holds before this routine is called }
$(u.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (u.c[s.p].seq \geq s.c[s.p].seq)$
$\wedge \ knows\_stable(u, u.c[s.p], s.p)$
$\Rightarrow$ { Definition of $knows\_stable$ }
$knows\_stable(u, s.c[s.p], s.p)$
$\Rightarrow$ { Lemma 3 }
$stable(s)$ ■

The following invariant shows that a process never sets its own entry to NULL, as its own entry is needed to start its next interval.

Invariant 2 $\forall s : s.c[s.p] \neq NULL.$

Proof. Process $P_{s.p}$ starts with a non-NULL $c[s.p]$. It updates it by taking maximum with another entry or by incrementing it. Hence the result. ■

The following invariant ensures that our dependency tracking mechanism does not introduce any false dependencies.

Invariant 3 $s \underline{\not\rightarrow} u \Rightarrow ((s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq))$

Proof. If distinct s and u belong to the same incarnation of a process and s did not happen before u then u happened before s. Now within the same incarnation, sequence number is only increased. Hence the invariant holds in this case. Therefore, we assume that s and u belong to different processes.

We show that the above invariant holds initially. We also show that the invariant holds after the execution of a routine, if it holds before the routine is called. It is sufficient to consider the routines that create a new interval or modify the dependency vector.

Initialize: Let u be the initial interval.
$\Rightarrow$ { Assumption: $s.p \neq u.p$ }
$u.c[s.p] = NULL$
$\Rightarrow$ { Invariant 2 }
$u.c[s.p].inc \neq s.c[s.p].inc$

Process_message: Let the message m be sent by an interval w. Let interval v process m and start interval u. Incrementing the sequence number in the second operation in the routine leaves the invariant unaffected as $s.p$ is not equal to $u.p$. Therefore, it is sufficient to show that the invariant holds after the max operation.

Now, $u.c[s.p] = max(v.c[s.p], m.c[s.p])$
$\Rightarrow$ { Definition of max }
$u.c[s.p] = v.c[s.p] \vee u.c[s.p] = m.c[s.p]$

We consider the following two cases: 1) $u.c[s.p] = v.c[s.p]$, 2) $u.c[s.p] = m.c[s.p]$

Case 1: $u.c[s.p] = v.c[s.p]$.

Now, $s \not\rightarrow u$
$\Rightarrow$ { Definition of happened before }

$s \not\rightarrow v$
$\Rightarrow$ { Invariant holds before this routine is called }
$(s.c[s.p].inc \neq v.c[s.p].inc) \vee (s.c[s.p].seq > v.c[s.p].seq)$
$\Rightarrow$ { Case 1 }
$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$

Case 2: $u.c[s.p] = m.c[s.p]$. If $m.c[s.p]$ is NULL then $u.c[s.p]$ is NULL. This implies that $v.c[s.p]$ is also NULL and the case 1 applies. Therefore, we consider the case when $m.c[s.p]$ is not NULL. This implies that $m.c[s.p]$ is same as $w.c[s.p]$. Rest of the proof is similar to the case 1 .

Start_incarnation: let u be the state in which this routine is called.
Now, $s \not\rightarrow u$
$\Rightarrow$ { Invariant holds before this routine is called }
$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$
$\Rightarrow$ { Assumption $s.p \neq u.p$, only $u.c[u.p]$ is modified in this routine }
$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].seq > u.c[s.p].seq)$

Receive_log_prog: Let u be the interval calling this routine. Modification of dependency vector of u can cause violations of invariant involving two kinds of states: 1) $s \not\rightarrow u$, 2) $u \not\rightarrow s$. As $u.p$'th entry is not modified in this routine, so it is sufficient to consider the first case. We need to consider state s only if the $s.p$'th entry is set to NULL in this routine. By Invariant 2, $s.c[s.p]$ is non-NULL. Hence the result follows. ■

The following lemma shows that the predicate *knows_orphan* correctly detects an orphan state.

Lemma 5 $knows\_orphan(s, u) \Rightarrow orphan(u)$.

Proof. $knows\_orphan(s, u)$
$\Rightarrow$ { Definition of *knows_orphan* }
$\exists j : \exists (t, x) \in s.iet[j] : (t = u.c[j].inc) \ \wedge \ (x < u.c[j].seq)$

Let w be the minimum state lost in the failure of P_j that resulted in the failure announcement entry (t, x). Now $w.p$ is same as j.

Then, $w.c[w.p] = (t, x + 1)$
$\Rightarrow$ { Definition of $knows\_orphan(s, u)$ }
$w.c[w.p].inc = u.c[w.p].inc \ \wedge \ w.c[w.p].seq \leq u.c[w.p].seq$
$\Rightarrow$ { Invariant 3 }
$\neg(w \underline{\not\rightarrow} u)$
$\Rightarrow$ { w is a lost state }
$w \underline{\rightarrow} u \ \wedge \ rolled\_back(w)$
$\Rightarrow$ { Definition of *orphan* }
$orphan(u)$ ■

The following lemma proves a similar result in the other direction.

Lemma 6 $orphan(u) \Rightarrow \forall i : \exists w \in P_i : knows\_orphan(w, u)$

Proof. Let v be lost in a failure and s be the state restored after that failure. Then we prove that s and v have the same incarnation number before the routine *Start_incarnation* is called by s. If s and v have different incarnation number then the routine, *Start_incarnation* must have been called by an intermediate state. Since s is the first state to call *Restart*, after loss of v, the routine *Rollback* must have been called in some intermediate state. As *Log_state* is called in *Rollback*, so all unlogged states including v are made stable. Then v cannot be lost in a failure. Therefore, s and v have the same incarnation number.

Now, $orphan(u)$
$\Rightarrow$ { Definition of orphan }
$\exists v : lost(v) \wedge v \underline{\rightarrow} u$
$\Rightarrow$ { Above argument, *Restart* is atomic, idempotent. }
$\exists v, s : lost(v) \wedge v \underline{\rightarrow} u \wedge broadcast(s.c[v.p]) \wedge s.c[v.p].inc = v.c[v.p].inc \wedge v.c[v.p].seq > s.c[v.p].seq$
$\Rightarrow$ { Reliable broadcast includes the execution of *Receive_failure_ann.* }
$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = v.c[v.p].inc \wedge v.c[v.p].seq > e.seq$
$\Rightarrow$ { Invariant 1, not stable v }
$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = u.c[v.p].inc \wedge u.c[v.p].seq > e.seq$
$\Rightarrow$ { Definition of $knows\_orphan$ }
$\forall i : \exists w \in P_i : knows\_orphan(w, u)$ ∎

Two executions of a process are considered equivalent if their stable states and the sets of messages sent to other processes are same.

Lemma 7 *Given an arbitrary execution of a process and an arbitrary point of failure, there exists an equivalent execution in which the failure occurs just before the routine* Log_state *is called.*

Proof. A failure during execution of any routine that modifies volatile storage only is same as the failure before the execution of that routine.

Using the techniques in [25], all routines that modify stable storage can be made atomic. These routines are idempotent as well. A repeated execution of any routine in Figure 9 has the same effect as that of executing it only once, provided no new failure announcements are received during the repeated execution. If new failure announcements are received then repeated execution is equivalent to an execution in which all failure announcements are received before executing the routine under consideration. This follows from the fact that the set of the stable states is totally ordered and execution of the routine *Receive_failure_ann* has either no effect or causes one of the stable states to be restored. So repeated execution of a number of failure announcements is same as executing them once in an arbitrary order. Therefore, any failure is equivalent to a failure just before the routine *Log_state* is called. ∎

The following theorem proves the correctness of our protocol.

Theorem 4 *The protocol rolls back all orphan states and orphan states only.*

Proof. As per Lemma 7, any failure is equivalent to a failure before executing the routine *Log_state*. Therefore, the only effect of a failure is that unlogged states are lost.

We show below that all orphan states are rolled back when failure announcement arrives. Also all orphan messages are discarded whether they arrive before failure announcement or after.

Therefore, no state becomes orphan with respect to a failure after the arrival of the corresponding announcement.

In routine *Restart*, an announcement is made about the entry of the last stable interval of the last incarnation. On receiving this announcement, all processes roll back all states that satisfy the predicate *knows_orphan*. By Lemma 6, all orphan states satisfy this predicate. Further, by Lemma 5, only orphan states satisfy this predicate. This ensures that the resulting system state is consistent. This point requires some elaboration. Let interval s be lost in a failure. One can imagine a protocol which rolls back a state u that is not dependent on s. A protocol that rolls back all states dependent on a lost state s can be wrong in the following way. It may not roll back a state w that is dependent on u. As it rolled back u, resulting system state will not be consistent.

As proved below, all messages that are orphan with respect to this failure are discarded. A message is orphan if its sending interval is orphan. A sending interval's dependency vector is attached with the message. As lost states are not stable, their entry in message's dependency vector is not set to NULL in the routine *Check_send_buffer*. Hence by using the Invariant 1, all orphan messages that have been received before the corresponding failure announcement are discarded when that failure announcement is received.

We assume that reliable delivery of failure announcement includes the atomic execution of the routine *Receive_failure_ann*. This means that received failure announcements are not lost afterwards. So all messages that are orphan w.r.t. this failure and that arrive after the failure announcement will be discarded upon their arrival in the routine *Receive_message*.

This completes the proof obligations stated earlier. ■

The following theorem shows that our protocol does not indefinitely postpone the sending of a message.

Theorem 5 *Each message in Send_buffer is either lost in a failure, or discarded as an orphan, or eventually sent.*

Proof. Consider a message m that is not lost in a failure and is not discarded as an orphan and is present in the *Send_buffer* of P_i. Let w be the maximum state of a process P_j on which m is dependent. If w is lost in failure then by Lemma 6, P_i will detect that m is orphan and will discard it. Else w will eventually become stable and by Lemma 4, P_i will eventually set $m.c[j]$ to NULL. As our choice of j was arbitrary, so the number of non-NULL entries in the dependency vector of m will become at most K and m will be sent. ■

The following theorem shows the meaning of K.

Theorem 6 *Given any message m released by its sender, the number of processes that can make the message orphan on their failure is at most K.*

Proof. In *Check_send_buffer*, the j'th entry of the dependency vector of a message m is set to NULL when the corresponding interval in P_j becomes stable. As per proof of Theorem 3, a failure of P_j cannot cause m to become an orphan. Since m is released when the number of non-NULL entries become at most K, the result follows. ■