Distributed Recovery with K-Optimistic Logging

Om P. Damani, Yi-Min Wang and Vijay K. Garg

TR-PDS-1998-10 **October 1998**

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

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 [9] is an effective technique for providing low-cost fault tolerance to distributed applications [4, 11, 18, 28]. For fault-resilience, a process periodically records its state on a stable storage [20]. 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 [5]. 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 [9]. It can be used either to reduce the amount of lost work due to failures in long-running scientific applications [9], or to enable fast and localized recovery in continuously-running service-providing applications [16].

Depending on when received messages are logged, log-based rollback-recovery techniques can be divided into two categories[1]: pessimistic logging [4, 16] and optimistic logging [18, 28]. 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.

Although pessimistic logging and optimistic logging provide a trade-off between failure-free overhead and recovery efficiency, it has traditionally been 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 reasonably fast upon a failure. Furthermore, since system load and failure rate may fluctuate during the execution, it is desirable to allow the parameters to be dynamically changed in response to the fluctuation. For example, when a telecommunications system is experiencing an unusual high-load condition due to special events such as holidays and disasters, it may need to reduce fault-tolerance overhead in order to handle the traffic.

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 results generalizes several previously known protocols.

The outline of the rest of the paper is as follows. Section 2 presents the related work and describes several scenarios where optimistic protocols are desirable. Section 3 describes the

[1]Recently a new class of protocols, called *causal logging* protocols have been proposed. We discuss them in Section 2.

system model. Section 4 presents the properties of dependency vector that we use to track transitive dependencies. Section 5 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 presents the experimental results comparing the failure-free overhead and recovery time for different values of K. Section 7 concludes the paper. Detailed proofs of the theorems are presented in the appendix.

2 Related Work

Strom and Yemini [28] initiated the area of optimistic message logging. Their scheme, however, suffers from the *exponential rollback* problem, where a single failure of a process can roll back another process exponential number of times. Johnson and Zwaenepoel [18] presented a centralized optimistic protocol to recover the maximum recoverable state. Peterson and Kearns [23] give a synchronous protocol based on vector clocks. Their protocol cannot handle multiple failures. Smith, Johnson and Tygar [25] presented the first completely asynchronous, optimistic protocol. The main drawback of their algorithm is the size of its dependency vectors, resulting in potentially high overhead during failure-free operations.

For K equal to n, our protocol reduces to optimistic protocol presented in [6], while for K equal to 0, it reduces to the pessimistic protocol presented in [17]. Table 1 shows a comparison of our work with some other optimistic recovery schemes. Note that, using our ideas presented in [6], Smith and Johnson reduced the size of the dependency vectors in their algorithm [26].

	Message ordering required	Asynchronous recovery	Maximum rollbacks per failure	Number of timestamps in dependency vector	Number of concurrent failures allowed	Number of broadcasts per failure
Strom and Yemini [28]	FIFO	Yes	$\Theta(2^n)$	$O(n)$	n	$O(n)$
Johnson and Zwaenepoel [18]	None	No	1	$O(1)$	n	1
Sistla and Welch [27]	FIFO	No	1	$O(n)$	1	$O(n)$
Peterson and Kearns [23]	FIFO	No	1	$O(n)$	1	1
Smith, Johnson and Tygar [25]	None	Yes	1	$O(n^2 f)$	n	-
This Paper	None	Yes	1	$O(n)$	n	1
Smith and Johnson [26]	None	Yes	1	$O(nf)$	n	-

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

Recently a new class of protocols, called *causal logging* protocols [1, 10], have been proposed. These protocols provide orphan-free recovery without incurring the overhead of synchronous logging. They limit the rollback to the most recent checkpoint on stable storage. They avoid synchronous access to the stable storage except during output commit. These advantages come at the expense of more complex recovery protocols. They log messages in processes other than the receivers. So synchronization is required during recovery.

Note that both pessimistic and causal logging protocols do not create any orphans and so do not roll back any non-failed processes. However, we believe that the ability to roll back is desirable in many scenarios:

1. **Non-failed-stop processes:** Traditional logging protocols are based on the assumption that processes fail by simply crashing and they do not do any other harm, like sending incorrect messages. In practice, there may be some latency between fault-occurrence and fault-detection. This will result in some incorrect computation. This incorrect computation should be rolled back and incorrectly sent messages should be discarded.

2. **Software bugs:** Traditional logging protocols assume that successive failures of a process are independent. So restarting a failed process should not lead to a crash due to the reason it failed earlier. However, when a software bug crashes a program, deterministically recreating the pre-failure computation will likely trigger 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 bypassing the state that caused the original crash [28, 31].

3. **Optimistic computations:** Many applications are optimistic and require rollback capability, for example, distributed optimistic simulation [8]. In such applications fault-tolerance can be easily integrated using optimistic recovery [7]. Thus fault-tolerance overhead becomes negligible.

4. **Distributed debugging:** Techniques used for optimistic recovery can also be used for distributed debugging where a user wants to test a program under different message orderings. After looking at the results of a particular run, a failure can be simulated and a different message ordering can be tried.

In Section 5.3.5 we show how our K-optimistic protocol works well in above scenarios for all values of K(including $K = 0$, i.e. pessimistic case). Similar techniques may be applicable to causal protocols.

3 Model of Computation

A distributed computation is a set of process executions. A process execution is divided into a sequence of *state intervals* each of which is started by a nondeterministic event such as message receipt[2]. The execution within each interval is completely deterministic. This is called the *piecewise deterministic (PWD) execution model* [28]. We use the term 'state' and 'state interval' interchangeably. For presentation and proof purposes, we distinguish between the state restored after a failure or rollback and the state resulting when the corresponding process has taken the actions needed for recovery. A rollback-recovery layer is implemented underneath the application layer to perform checkpointing, message logging, dependency tracking, output commit, etc. During failure-free execution, each process takes independent or coordinated checkpoints [9], and employs additional message logging.

[2]In this paper, we assume that message-delivering events are the only source of nondeterminism.

The receiver of a message depends on the content of the message and therefore on the sender of the message. This dependency relation is transitive. The receiver becomes dependent only after the received message is delivered to the application.

A process periodically takes its checkpoint. It also asynchronously logs all received messages to the stable storage in the order they are received. At the time of checkpointing, all unlogged messages are also logged.

A failed process is *restarted* by restoring its last checkpoint and replaying the logged messages which were received after the restored state. Since some of the messages might not have been logged at the time of the failure, some of the pre-failure states, called *lost* states, cannot be recreated. Now, consider the states in other processes, which depend on the lost states. These states, called *orphan* states, must be rolled back. Since other processes have not failed, they can log all unlogged messages and save their states before rolling back. Thus no information is lost in a rollback. Note the distinction between a restart and a rollback. A failed process restarts whereas a rollback is done by non-failed processes. As a result, some information may be lost in a restart but not in a rollback. Each restart or rollback starts a new *incarnation* of the process. A message sent by a lost or an orphan state is also called an *orphan* message. A process receiving an orphan message must discard it. Otherwise the receiver becomes an orphan. If the current state of a process is orphan then process itself is called orphan.

An example of a distributed computation is shown in Figure 1. The numbers shown in rectangular boxes will be explained in Section 4 and can be ignored till then. Solid lines show the useful computation, while dashed lines show the computation that is either lost in a failure or rolled back explicitly by the recovery protocol. The letter s in the name of a state denotes an application state, while the letter r denotes a recovery state, i.e. a state in which actions needed for recovery are being taken. The application states are shown as empty circles, while the recovery states are shown as filled circles. Other 3 digits in the name of a state can be ignored till Section 4. In the figure, ci, the checkpoint of process Pi, is shown as a square.

In Figure 1(a), process $P1$ takes a checkpoint $c1$, processes some messages not shown in the figure and reaches the state $s117$. It logs all the messages that have been received so far. Then it acts on the message $m0$ and sends the message $m1$ to $P2$. It then fails (shown by a cross) without logging the message $m0$ or receiving the message $m2$. It loses its volatile memory, which includes the message $m0$. During this time, $P2$ acts on the message $m1$.

Now Figure 1(b) shows the post-failure computation. On restarting, $P1$ restores its last checkpoint $c1$, replays all the logged messages and reaches the state $s117$. Then it takes some actions needed for recovery, resulting in state $r127$. It then broadcasts a failure-announcement (shown by dotted lines) and continues its execution. $P2$ receives this announcement in state $s216$ and realizes that it is dependent on a lost state. It rolls back, restores its last checkpoint $c2$, and replays the logged messages till it reaches state $s215$. Next message to be processed is $m1$, the message that made it orphan. It discards the message m1 and continues from state $r225$. On receiving the announcement, $P0$ and $P2$ also resend the messages $m0$ and $m2$ as they might have been lost in the failure. $P1$ now processes $m0$ and $m2$ in an order different from the pre-failure order. As a result, message $m1$ is not regenerated.

In optimistic recovery schemes, each process asynchronously saves messages in the volatile buffer to stable storage. Periodically, it broadcasts a logging progress notification to let other processes know which of its state intervals have become stable. Such information is accumulated locally at each process to allow output commit and garbage collection [28].

Henceforth, we use the following notations:

- i, j, k refer to process numbers.
- t refer to the incarnation number of a process.
- s, u, v, w, z refer to a state(or a state interval).
- P_i refers to process i.
- $P_{i,k}$ refers to incarnation k of P_i.
- $s.p$ denotes the process number to which s belongs, that is, $s.p = i \Rightarrow s \in P_i$.
- x, y refer to state interval indices.
- $(t, x)_i$ refers to the x^{th} state interval of the t^{th} incarnation of process P_i.
- m refers to a message.

We extend Lamport's *happen before*($\rightarrow$) relation [19] as follows. For the state intervals s and u, $s \rightarrow u$ is the transitive closure of the relation defined by the following three conditions:

- $s.p = u.p$ and s was executed immediately before u without any intermediate failure or rollback, (for example, $s117 \rightarrow s118$ in Figure 1(a)), or
- $s.p = u.p$ and s is the state restored after a failure or a rollback and u is the state after $P_{u.p}$ has taken the actions needed for recovery, (for example, $s117 \rightarrow r127$ in Figure 1(b)), or
- s is the sender of a message m and u is the interval which results from the processing of m by $P_{u.p}$ (for example, $s011 \rightarrow s118$ in Figure 1(a)).

In Figure 1(a), $s011 \rightarrow s216$, but in Figure 1(b) $s118 \not\rightarrow s128$ (*not happen before*). 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, $s = u$.

The protocol for recovery might cause some recovery messages to be sent among processes. These recovery messages do not contribute to *happen before*: if s sends a recovery message to u then, because of this message, u does not become dependent on s.

4 Dependency Vector

To enable decentralized recovery and efficient output commit, transitive dependency tracking is commonly used to record the highest-index state interval of each process, on which a local process depends. A dependency vector (DV) is used for the above purpose [28]. Although DV has been used before, its properties have not been discussed before. We discuss its properties for two reasons. First, DV can be used in applications other than the recovery, for example, in

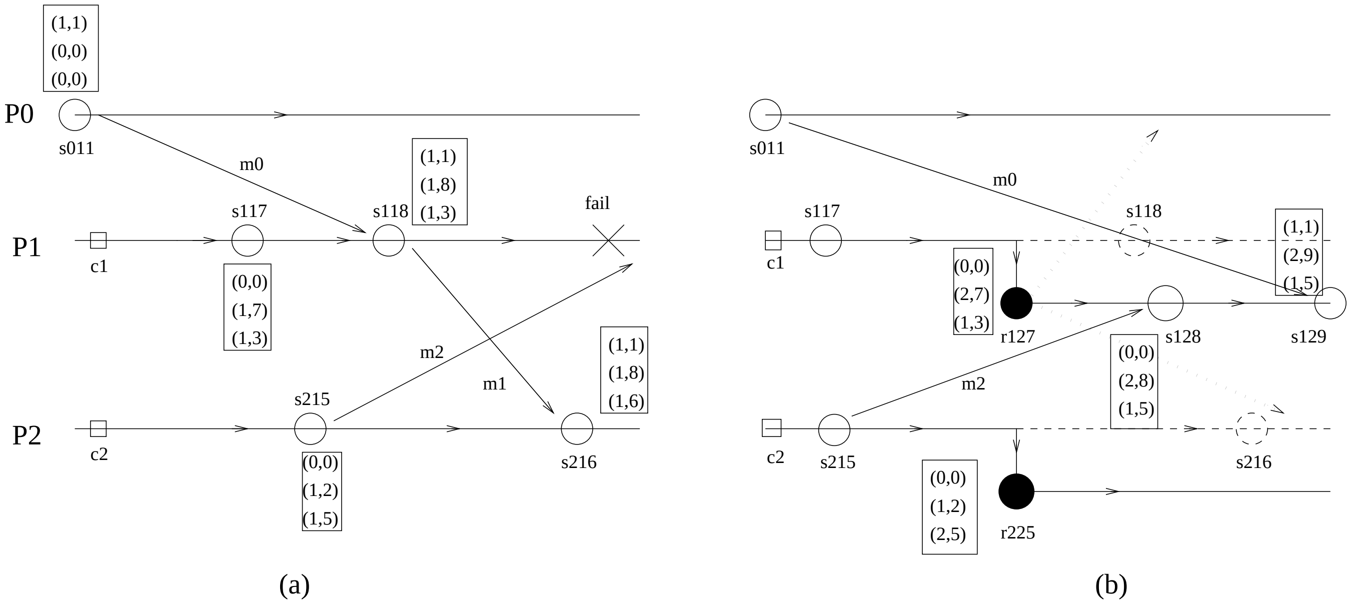

Figure 1: A Distributed Computation: (a) Pre-failure computation. (b) Post-failure computation

predicate detection [14]. Second, we modify DV for use in our K-optimistic algorithm and hence a basic appreciation of its properties will help in understanding our algorithm.

Let us consider the DV of a process P_i. It has n entries, where n is the number of processes in the system. Each entry contains an *incarnation number* and a *state interval index* (or simply *index*). The incarnation number in the i'th entry of its DV (its own incarnation number) is equal to the number of times it has restarted 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, index sii). Then, $e_1 < e_2 \equiv (t_1 < t_2) \vee [(t_1 = t_2) \wedge (sii_1 < sii_2)]$.

A process P_i sends its DV along with every outgoing message. Before processing a message, the process updates its DV with the message's DV by taking the componentwise maximum of entries and incrementing its own index.

To start a new incarnation, a process increments its incarnation number and leaves the index unchanged. A new incarnation is always started after a rollback or a restart after a failure. The formal description of the DV algorithm is given in Figure 2. An example of DV is shown in Figure 1. DV of each state is shown in a rectangular box near it. The row i of the DV corresponds to Pi. The three digits in the name of a state refer to the process id, the incarnation number and the state interval index, respectively.

4.1 Properties of DV

DV has properties similar to Mattern's vector clock [21]. It can be used to detect transitive dependencies between *useful* states, that is, the states which are neither lost nor orphan.

We define ordering between two DV $c1$ 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 sii)       // incarnation, index
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 (data, m.c) :
    // P_i receives dependency vector 'm.c' with incoming message
    ∀ j: c[j] := max(c[j],m.c[j]) ;
    c[i].sii := c[i].sii + 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 DV of $P_{s.p}$ in state s. The following lemma gives a necessary condition for '$\not\rightarrow$' 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 DV's and then increments the index 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, local DV 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 DV of $P_{s.p}$. This argument can be formalized using the induction technique on '$\not\rightarrow$', given by Garg and Tomlinson [13]. Hence $u.c[s.p] < s.c[s.p]$. ■

As shown in the next theorem, the above condition is also sufficient for '$\not\rightarrow$' relation. The next theorem shows that, despite failures, DV keeps track of causality for the *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 $s \rightarrow u$. 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 the DV along each link in the path, $\forall j : s.c[j] \leq u.c[j]$. Since $u \not\rightarrow s$, from Lemma 1, $s.c[u.p] < u.c[u.p]$. The converse follows from Lemma 1. ■

Note that the DV does not detect the causality for either lost or orphan states. In Figure 1, $s118.c < s129.c$, even though $s118 \not\rightarrow s129$. To detect causality for lost or orphan states, we use *incarnation end table*, as explained in Section 5.3.

5 K-optimistic Logging

5.1 Motivation

Traditional pessimistic logging and optimistic logging provide only a coarse-grain trade-off between failure-free overhead and recovery efficiency. For long-running scientific applications, the primary performance measure is typically the total execution time. Since hardware failures are rare events in most systems, minimizing failure-free overhead is more important than improving recovery efficiency. Therefore, optimistic logging is usually a better choice. In contrast, for continuously-running service-providing applications, the primary performance measure is typically 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, most commercial service-providing applications have chosen pessimistic logging [16].

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 logging 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 outgoing message. Specifically, 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 7).

This provides a spectrum of trade-off between recovery time and logging overhead, with traditional optimistic and pessimistic logging being the two end-points of the spectrum. 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 tuned to adjust to a changing environment.

5.2 Theoretical Basis

In Section 3 we presented the distinction between restart and rollback. However, traditional optimistic recovery protocols [25, 28] have treated restart due to a process's own failure and rollback due to some other process's failure in a similar fashion and have referred to lost states as rolled-back states. In the same vein, they have also referred to lost states as orphan states. In order to relate our results to those in the literature, from here on, we also use the same terminology unless explicitly noted otherwise. A state satisfies predicate $rolled\_back$ if it has been either lost in a failure or explicitly rolled back by the recovery protocol. Any state dependent on a rolled back state is called an *orphan*. The following predicate formally defines an orphan state.

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

Traditional asynchronous recovery protocols usually require every non-failed rolled-back process to behave as if it itself has failed [25, 28] by starting a new incarnation and broadcasting a rollback announcement. We observe that, under piecewise deterministic execution model, announcing only 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 due to rollback of another interval u. Now interval u rolled back either because $P_{u.p}$ failed or because it became orphan due to the rollback of another interval z. By repeatedly applying the previous observation, we find an interval w whose rollback due to $P_{w.p}$'s failure caused v to become orphan. By definition of transitive dependency tracking, $P_{v.p}$ can detect that v transitively depends on w. Therefore, $P_{v.p}$ will detect that v is orphan when it receives the failure announcement from $P_{w.p}$. ∎

Above observation was first used in [6] and later used in [26]. We carry this observation even further by proving, in Theorem 3, 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 the 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 says 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 can never be lost in a failure. It can always be reconstructed by restarting from its previous checkpoint and replaying the logged messages in the original order. Now following the proof in Theorem 2, the orphan interval v transitively depends on interval w which was lost due to $P_{w.p}$'s failure. That must mean that w had not become stable when the failure occurred. By definition of commit dependency tracking, $P_{v.p}$ can detect that v transitively depends on w, and so it will detect that v is orphan when it receives the failure announcement from $P_{w.p}$. ∎

Logging progress notification is an explicit way to inform other processes of new stable state intervals. Such information can also be obtained in a less obvious way from another source. A failure announcement containing index $(t, x')_i$ indicates that all states of incarnation t of P_i with index greater than x' have been lost in a failure. Since the state with index 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 $x \leq x'$.*

Completely asynchronous recovery protocols in general need to keep track of dependencies on all incarnations of all processes. Strom and Yemini [28] introduced the following blocking in their protocol 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 $(t-1)^{th}$ incarnation, 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 $(t-4, x)_i$ 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 $(t-4, x)_i$ has become stable before it can acquire the dependency on $(t, x+10)_i$ to overwrite $(t-4, x)_i$. Process 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.

5.3 The Protocol

The data structures used by our protocol are shown in Figure 3. The integer K is the degree of optimism. Logging progress information is stored in *log_prog*. It contains an entry for the highest known stable interval of each known incarnation of each process. All the received failure announcements are stored in an incarnation end table (*iet*). Variable *cur_inc* stores the current incarnation number in stable storage. This is to avoid the loss of incarnation number information in a failure. A simplification that we have used to clarify the correctness proof is that of replacing the checkpointing and message logging with the saving of entire states. Our implementation indeed uses checkpointing and message logging. We discuss this point in detail in Section 5.3.2.

Figure 4 shows formal definitions of several predicates that are used in our protocol. If a state s knows that a state u is orphan then the predicate $knows\_orphan(s, u)$ is said to be true. This is the case, when the *iet* of s shows u to be dependent on a lost state. If s belongs to the stable state list of $P_{s.p}$, then $stable(s)$ is said to be true. Function *index* takes a set of entries and an incarnation number and returns the state index associated with the incarnation number in the set. A state u is said to correspond to entry e if $u.c[u.p]$ is equal to e. If s knows that state of P_j that corresponds to entry e is stable then predicate $knows\_stable(s, e, j)$ is said to be true. Message m can be processed in state s if no dependency on an unstable interval will be overwritten in taking maximum of the transitive dependency vectors and then the predicate

Process P_i :

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

Figure 3: Data structures used in our protocol

$admissible(m, s)$ is said to be true. Function *get_state* takes a process id and an entry and returns the state interval of the given process that corresponds to that entry. We later show that state of a process corresponding to a given entry is unique. In other words, the function *get_state* is well defined. A NULL entry is lexicographically smaller than any non-NULL entry.

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

All routines modifying volatile state are shown in Figure 5 and those modifying stable storage are shown in Figure 6. We assume that routines *Check_orphan, Check_send_buffer, Broadcast_log_prog, Log_state* are executed periodically. We also assume that reliable broadcast of a failure includes the execution of the routine *Receive_failure_ann*. Next, we describe the actions taken by a process P_i upon the occurrence of different events. We first describe the routines in Figure 5.

Initialize: Upon starting the execution, a process has no dependency on any other process. Therefore, P_i sets all except its own dependency vector entries to NULL. Since each process execution can be considered as starting with an initial checkpoint, the first state interval is always stable. Therefore, it 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.

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

Process_message: When the application wants to process a message, any of the *admissible* messages among the received ones is selected. As explained earlier, 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

$$\text{knows\_orphan(s,u)} \equiv \exists \text{j}: \exists \text{(t,x)} \in \text{s.iet[j]}: (\text{t} = \text{u.c[j].inc}) \wedge (\text{x} < \text{u.c[j].sii})$$

$$\text{stable(s)} \equiv \text{s} \in \text{Stable\_state\_list}$$

$$\text{index(se, t)}: \textbf{return } \text{x } \textbf{where } \text{(t,x)} \in \text{se}$$

$$\text{knows\_stable(s,e,j)} \equiv \text{index(s.log\_prog[j], e.inc)} \geq \text{e.sii}$$

$$\text{admissible(m,s)} \equiv \forall \text{j}: [\text{s.c[j].inc} \neq \text{m.c[j].inc} \Rightarrow \text{knows\_stable(s, min(s.c[j], m.c[j]), j)}]$$

$$\text{get\_state(j,e)}: \textbf{return } \text{s } \textbf{where } \text{s.p} = \text{j} \wedge \text{s.c[j]} = \text{e}$$

$$\text{e} = \text{NULL} \equiv (\text{e.inc} = \text{NULL} \wedge \text{e.sii} = \text{NULL})$$

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

Figure 4: Formal definitions

incarnations of any process, P_i waits for the interval with the smaller incarnation number to become stable. Such 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, so such blocking will rarely occur. After application acts on the message, current state is included in the volatile log.

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

Send_message: To send a message m, the current DV is attached to the message and it is added to the *Send_buffer*. The message is held in a Send_buffer if the number of non-NULL entries in its dependency vector $m.c$ is greater than K.

Check_send_buffer: This routine updates the dependency vectors of messages in Send_buffer. It is invoked by 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 released.

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

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 the 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.

Insert: This function inserts an entry (t, x) in a set se. If an entry (t, y) for incarnation t

```
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 := {} ;

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].sii := c[i].sii + 1 ;
            // Application acts on the message
        State_list := State_list ∪ {s} ;

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

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

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) ;

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 ;

Insert(se, (t,x)) :        // Insert entry (t,x) in the set se
    if ∃ y: (t,y) ∈ se then                  // an entry for incarnation t exists in se
        se := (se - {(t,y)}) ∪ {(t,max(x,y))}
    else se := se ∪ {(t,x)} ;
```

Figure 5: K-optimistic logging protocol: Routines modifying volatile states only.

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

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(j,t,x) ;

Rollback(j,t,x) :
      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 6: K-optimistic logging protocol: Routines modifying stable states.

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.

We next describe the routines in Figure 6.

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

Restart: On restarting after a failure, P_i restores its last stable state and broadcasts the index of this state. It then starts a new incarnation by incrementing its incarnation number in routine **Start_incarnation**.

Receive_failure_ann: On receiving a failure announcement, P_i updates its incarnation end table. As explained in Section 5.2, 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 5). If the current state of P_i has become orphan due to this failure, then it rolls back by calling **Rollback**.

Rollback: Before rolling back, P_i logs its volatile states in the stable storage. Clearly, an implementation will log only the non-orphan states. The highest non-orphan stable state is restored and the following orphan states are discarded from the 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 the stable storage as the variable *cur_inc*. This ensures that current incarnation number is not lost in a failure. It also updates the DV.

Note that there is nothing in the protocol to prevent a change in the value of K. So 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 file server becomes busy, a process may choose to increases its K value.

5.3.1 Output Commit

If a process needs to commit output to external world during its execution, it maintains an Output_buffer like the Send_buffer. This buffer is also updated whenever the Send_buffer is updated. An output is released when all of its dependency entries 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 6, we do not use the Output_buffer. Instead, we attach a K value with each message with 0 being assigned to the output messages.

In practice, the concept of K-output commit may also be useful. Although strict output commit may be necessary for military applications or medical applications, most service-providing applications can revoke 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 commit latency and the degree of commitment.

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

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

Restart :        // after failure
    s := head(Stable_state_list) ;
    replay the logged messages that follow;
    Insert(iet[i], c[i]) ;
    Broadcast_failure(c[i]) ;
    // An implementation will compute the index of maximum recoverable
    // state and broadcast it before replaying the messages.
    Start_incarnation ;

Rollback(j,t,x) :
    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 ;

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

Figure 7: Modified K-optimistic logging protocol routines.

5.3.2 Implementation Notes

There are many implementation issues that we have not discussed so far.

1. A simplification that we have used to clarify the correctness proof is that of replacing the checkpointing and message logging with the saving of entire states. In our presentation, we save all the states on the volatile and the 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 will save the received message instead of the states in the volatile memory. Periodically, 'current state' will be saved on the stable storage as a checkpoint. Any state can be reconstructed by restoring the previous checkpoint and replaying all the messages that have been received since then. In the implementation described in Section 6, we use this strategy. Figure 7 shows the routines that are affected due to this implementation. Instead of volatile *State_list*, a volatile *Message_list* is used. A *Stable_message_list* is also used. Checkpoints are stored in *Stable_state_list*.

2. In Figure 7, routine **Log_message** is called in the routine **Rollback** to log all the unlogged messages. An implementation will log only the 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.

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

5. In general, logging progress need not be broadcast reliably. This is because for a given incarnation, logging progress is monotonic. So 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.

6. In case of a delayed failure announcement, a message from a new incarnation can also help identify the orphan states. Suppose the j^{th} entry of P_i's dependency vector is (t, y), and P_i has not received failure announcement for P_j's t^{th} incarnation. If P_i receives a message m, such that the j^{th} entry of m's DV is (t', y') and $t' > t$ and $y' \leq y$, then P_i knows that P_j's interval (s, y) must have been rolled back and so P_i's current state is an orphan. However, it does not know how far to roll back. So it stops computing and waits for the missing failure announcements to arrive. Similarly, messages in *Send_buffer* can be detected to be orphan.

7. Logging information can also help identify the orphan states in a way similar to that explained in the previous paragraph. Suppose the j^{th} entry of P_i's DV is (t, y). If P_i receives a logging progress notification from P_j, informing that P_j's interval $(t + 1, y - 3)$ has become stable, P_i detects that its current state is orphan.

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

9. As discussed in Section 5.3.6, recovery layer needs to maintain the message send history.

5.3.3 An Example

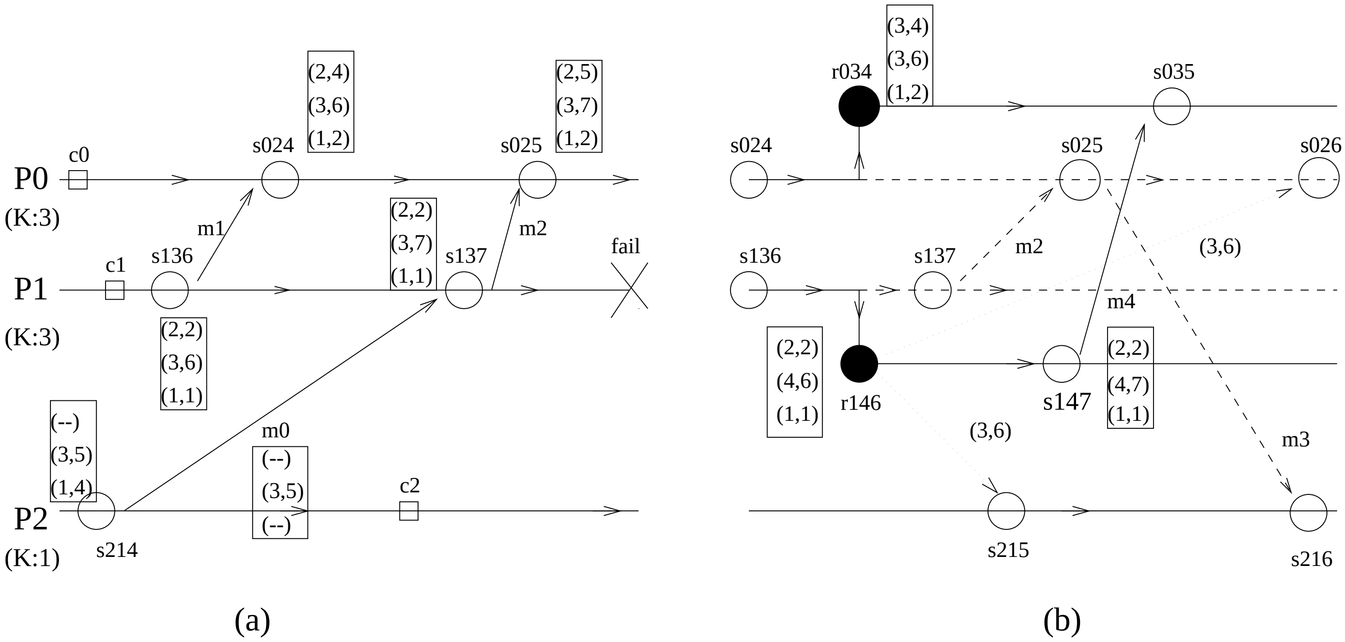

Figure 8: K-optimistic recovery: (a) Pre-failure computation (b) Post-failure computation

Figure 8 shows an example of our protocol in action. The value of the DV is shown for some of the states and messages. Some of the state transitions are not shown to avoid cluttering of the figure. The K value for $P0$ and $P1$ is 3 and that for $P2$ is 1.

In Figure 8(a), $P1$ sends the message $m1$ to $P0$ in state $s136$. In state $s214$, $P2$ sends the message $m0$ to $P1$. However, 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 state $s136$ and has not logged the message $m0$. During this time, $P0$ acts on the messages $m1$ and $m2$.

The post-failure computation is shown in the Figure 8(b). On restarting, $P1$ restores the last checkpoint $c1$, replays the logged messages and recreates the state $s136$. It increments its incarnation number, resulting in state $r146$ and sends the failure-announcement (3,6) to other processes (shown by dotted arrow). $P1$ now continues execution and sends message $m4$ to $P0$. During this time, $P0$ sends the message $m3$ to $P2$. Then the recovery layer of $P0$ receives the message $m4$ in state $s026$. Note that the message $m4$ is received by the recovery layer and 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 DV of $m4$ is $(4, 7)$, while the second entry in $P0$'s DV in state $s026$ is $(3, 7)$. Therefore $P0$ decides itself to be orphan. It stops computing any further and waits for the rollback announcement from $P1$. On receiving this announcement it rolls back by restoring the checkpoint $c0$ and replaying the logged messages until, in state $s024$, it is about to process the message $m2$ that made it an orphan. It discards $m2$, increments its incarnation number, resulting in recovery state $r034$. It does not send out any rollback announcement. Now message $m4$ is processed. On receiving message $m3$, $P2$ detects that $m3$ is orphan and discards it.

5.3.4 Other Applications

K-optimistic protocol 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 become 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 a generalization of the result presented in [2], where checkpoints are used to avoid propagating dependencies between different applications that use a common file server.

5.3.5 Simulating a Failure

As discussed in Section 2, there are many scenarios like non-crash failures and software error recovery, where recreating the pre-failure state is undesirable. This poses a problem for our protocol as we are setting stable entries to NULL under the assumption that they can never be lost in a failure. But, now we want to simulate the loss of even the 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. Only incarnation numbers and state indices are compared to determine the maximum of two entries. Everything else remains the same except that for purpose of orphan detection, all entries including the stable ones must 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 DV.

There are clear limitations 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. This limitation adds little to other known limitations that a failure can be simulated only till last output commit or garbage collection point [28].

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

5.3.6 Remarks

The following issues are relevant to all the message logging protocols including ours. We just mention them and do not discuss them any further. These and many other issues are discussed in detail in [9].

- **Failure Detection:** In theory, it is impossible to distinguish a failed process from a very slow process [12]. In practice, many failure detectors have been built that work well for most practical situations [15]. Most of these detectors use a timeout mechanism.

- **Lost Messages:** On a failure, a process P_i loses information about the messages that it received but did not log before the failure. These messages are lost forever, unless P_i also broadcasts its DV along with the failure announcement and other processes resend all the messages that they sent to P_i (only those messages need to be retransmitted whose send states are concurrent with the restored state). This means that processes have to keep send-history. Observe that no retransmission of messages is required during a rollback of a process that has not failed, but has become orphan due to a failure of some other process. Before rolling back, it can log all the messages and so no message is lost.

- **Garbage Collection:** Some form of garbage collection is also required for reclaiming space used for checkpointing and message logging [28].

- **Stable Storage:** Stable storage must be available across failures. In a multi-processor environment this is easy, as other processors can access the disk even if one of the processors fails. In a networking environment, the local disk may be inaccessible when the corresponding processor fails. So a network server can be used to make checkpoints stable.

- **Process Identity:** When a failed process is restarted, it may have a different port number or IP address. So location independent identifiers should be used for the purpose of inter-process communication.

- **Environment Variables:** If a process is restarted on a different processor then some inconsistency may arise due to mismatch of the values of environment variables in pre- and post-failure computation. Logging and resetting of environment variables is required.

5.4 Properties of the Protocol

The following invariants, lemmas and theorems are used to prove our protocol correct. Their proofs can be found in the appendix.

The following lemma shows that the function *get_state* is well defined.

Lemma 2 *Different states of the same process have different entries.*

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

Lemma 3 $(s \underline{\rightarrow} w \wedge s.p = w.p \wedge stable(w)) \Rightarrow stable(s)$

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

Lemma 4 $knows\_stable(s, e, j) \Rightarrow stable(get\_state(j, e))$.

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

Lemma 5 $stable(get\_state(j, e)) \Rightarrow \forall i : \exists s \in P_i : knows\_stable(s, e, j)$

The following invariant is at the heart of our protocol. It ensures that dependencies on unstable state are never set to NULL. So when an unstable state fails, it helps in the detection of all the orphan states.

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

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

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

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

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

Lemma 6 $knows\_orphan(s, u) \Rightarrow orphan(u).$

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

Lemma 7 $orphan(u) \Rightarrow \forall i : \exists w \in P_i : knows\_orphan(w, u)$

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

Lemma 8 *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.*

The following theorem proves the correctness of our protocol.

Theorem 4 *The protocol correctly implements the recovery.*

The following theorem shows that our protocol restores the maximum recoverable state.

Theorem 5 *The protocol restores the maximum recoverable state.*

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

Theorem 6 *Each message in Send_buffer is either lost in a failure or discarded as an orphan or eventually sent.*

The following theorem shows the meaning of K.

Theorem 7 *Given any message m released by its sender, K is the maximum number of processes whose failures can revoke m.*

6 Implementation

6.1 Objective

The objective of our experiments is to present a methodology to select a value of K for a given system. There is no general answer to the question, "What value of K shall 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, file 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.

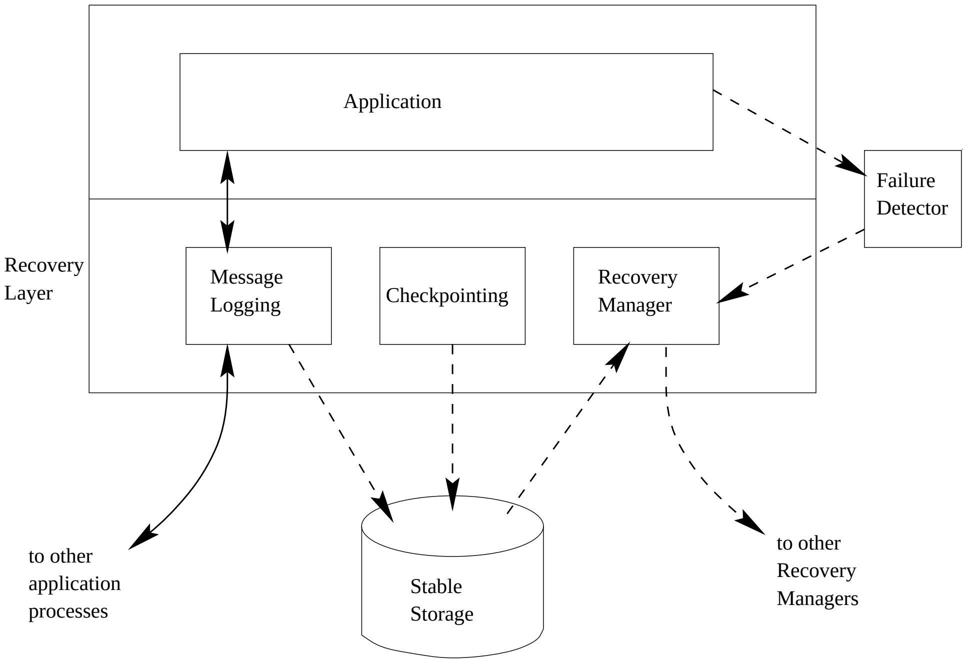

Figure 9: Recovery layer architecture.

6.2 Architecture

We have implemented a prototype of the K-optimistic protocol. Our architecture is shown in the Figure 9. 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, failure detector will start a new recovery manager who 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.

6.3 Message Logging Policy

In traditional optimistic schemes, message log is periodically flushed to the 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.

Note that an 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 may not work in presence of changing message sizes, message arrival rates and the 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. Regardless of the system load, this policy works like a traditional pessimistic logging policy for K equal to 0.

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.

In traditional optimistic schemes, logging notification period is determined by the maximum acceptable output commit latency, as the latter varies directly with the former. In this paper we have not focused on the issue of output commit and hence we do not discuss it any further. We have chosen a period of 500 millisecond for the logging notification, except when K equals n, for which notification period is 1 second. This is because when K equals 1, logging notification is needed only for the output commit and not for the progress of the computation.

6.4 Application Parameters

In our applications, each process receives a message, sleeps for a while, sends a message and then blocks for the next message receive. In the application *Random*, receiver of a message is chosen randomly by the sender, whereas in the application *Neighbor*, processes are arranged in a ring and they alternately send messages to their left and right neighbors only. For a given run, the size of a message is fixed but the compute time (time to sleep) is chosen uniformly from a range.

The applications *Random* and *Neighbor* were chosen because they are representatives of the many different applications studied in the literature [11, 24] and because of their contrasting characteristics. 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 processes 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.

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. So 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 applications. For message size of 10K, compute time of 80 to 100 millisecond is less than the average logging time for both the applications. 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*. We chose above values, because they represent a spectrum of application characteristics and result in logging overheads of 5-70%.

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 of our application 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 [24].

We measure the failure-free overhead by running the application without and with 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 may be different. This is because processes may roll back concurrently. Also, when one process rolls back, other processes may block. So the recovery time may change little if the blocked processes were to actually roll back a little.

6.5 Results

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 as the stable storage.

All measurements are made with applications distributed across 4 machines with 2 processes per machines. All results presented in this paper are averages over a number of trials. The duration of a trial for the case of no message logging ranged from 24 to 40 seconds for different combinations of application parameters.. Standard deviations for most measurements are under 1% of the average.

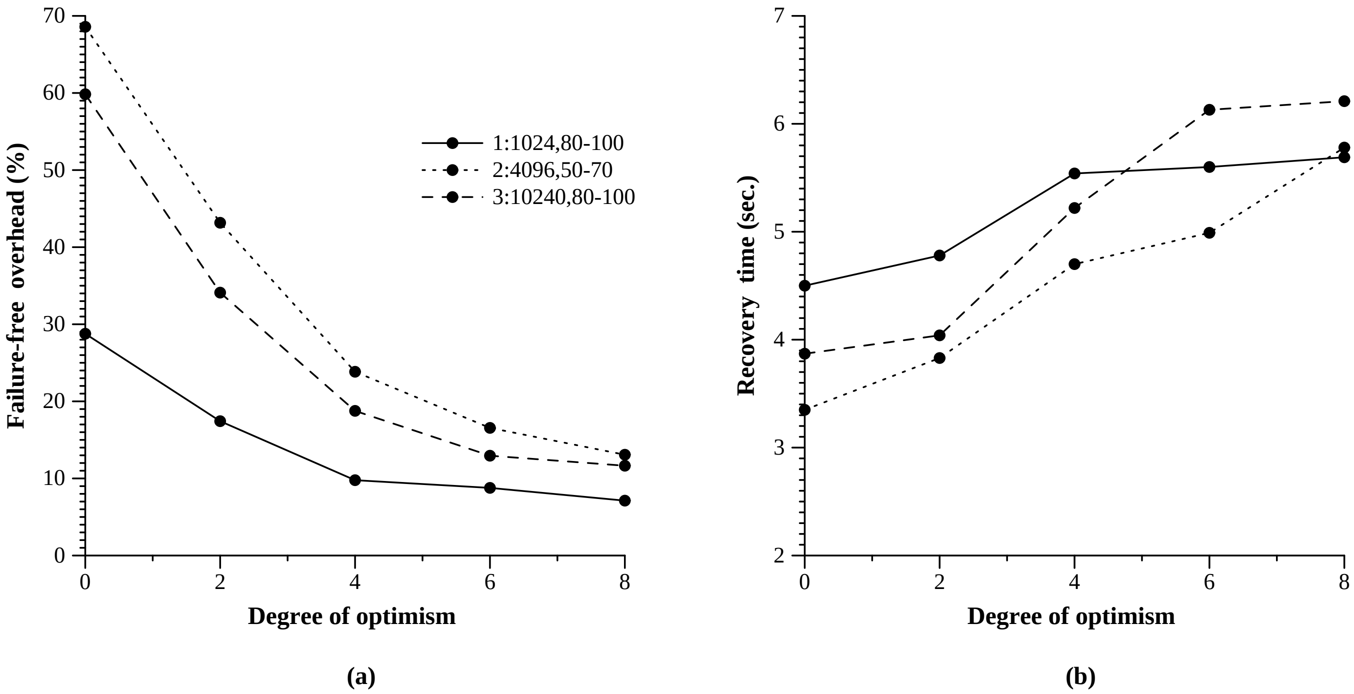

Figure 10: Performance results for application *Neighbor*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.

The results of our experiments are shown in Figures 10 and 11. 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

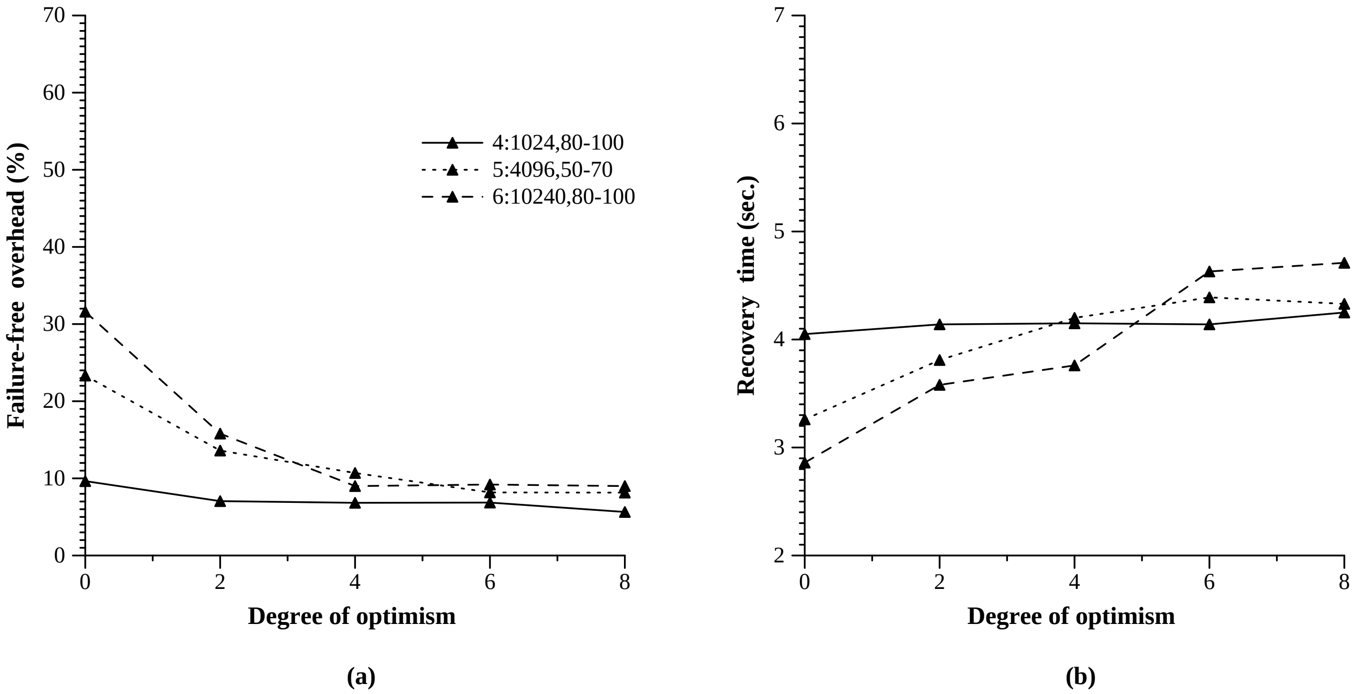

Figure 11: Performance results for application *Random*. Legend consists of the experiment number, the message size in bytes and the compute time range in milliseconds.

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.

Our experiments confirm that the trade-off between failure-free overhead and recovery time indeed exists and it is a continuous spectrum with the traditional optimistic and pessimistic logging being its endpoints. The actual trade-off involved depends on many parameters, three of which are discussed in this paper: communication pattern, message size and compute time.

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. So 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 failure-free runs and the single-failure run. This is because while a process is recovering, other processes are making progress. Also, other processes are sending messages to the recovering process. So 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. We explain this with reference to the experiments 1 and 3. In general, rate of progress of experiment 1 is more than that of experiment 2 as messages of size 1K are logged faster than messages of size 10K. However, the failed process takes same time to restore the checkpoint and replay the messages in both the experiments. So relative to failure-free computation, single failure computation runs slower for experiment 1 than for experiment 3.

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.

Note the contrast between lower and higher values of K. For lower values of K, recovery time is mainly determined by the replay time of a single process as very few other processes roll back and the extent of their roll back is also smaller. For higher values of K, many processes rollback and extent of their rollback is also higher. So the recovery time is determined by the multiple rollbacks. This effect is so pronounced that every pair of graphs in Figures 10(b) and 11(b) cross each other.

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.

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.

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

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 [31], 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.

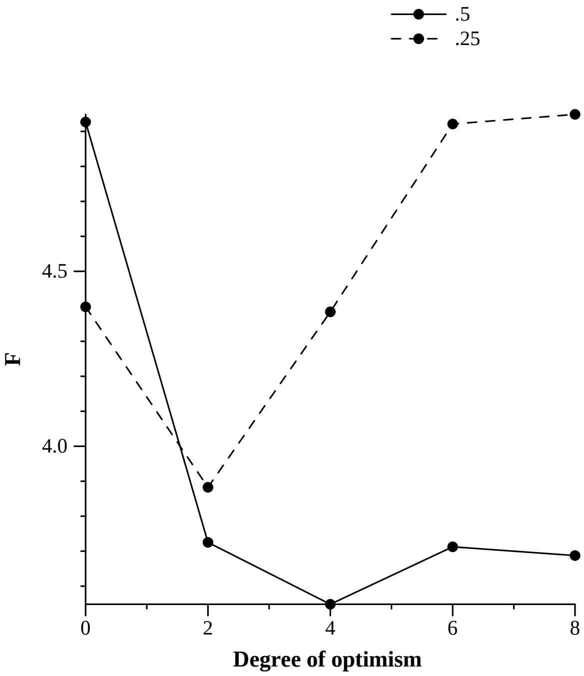

Figure 12: Selecting a K value for a given α. Numbers in the legend represent the α values.

6.6 Selecting K

In Section 6.1, 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 the 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 10. If the system designer wants to minimize 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 may be 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 8. In Figure 12, 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 and argument R is taken as the recovery time. As both the curves have a unique minima, we select the K value of 4 for α equal to .5 and K value of 2 for α equal to .25.

7 Conclusion

We have extended the happen before relation to the case when processes can fail and restart. We use dependency vector to track this extended happen before relation. This may be useful in applications other than the recovery, for example in predicate detection. In this model, 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 the results, we introduced the concept of K-optimistic logging to allow a system to explicitly fine-tune the trade-off between failure-free overhead and recovery efficiency. In such system, given any message, the number of processes whose failures can revoke the message is bounded by K, and so K basically 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 entire spectrum spanned by K-optimistic logging. We also presented a methodology to determine the value of K suitable for a particular application. Our experiments have confirmed that such a trade-off indeed exists in practice.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] O. P. Damani and V. K. Garg. Fault-Tolerant Distributed Simulation. To appear in *Proc. 12th Workshop on Parallel and Distributed Simulation*, 1998.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

[20] B. W. Lampson and H. E. Sturgis. Crash recovery in a Distributed Data Storage System. Tech. Rep., Xerox Palo Alto Research Center. April 1979.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] Y. M. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating Checkpointing with Transaction Processing. *Proc. 27th Intl. Symp. on Fault-Tolerant Computing*, 304-308, 1997.

[30] 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.

[31] 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*, Vol. 46, No. 10, pp. 1137-1141, Oct. 1997.

Appendix

In the following proofs, a set or an entry satisfies the predicate *broadcast* if it has been broadcast by some state. The following lemma shows that the function *get_state* is well defined.

Lemma 2 *Different states of the same process have different entries.*

Proof. Whenever a new state is started in the routine *Process_message*, state index 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. So reuse of the incarnation number does not matter. ■

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

Lemma 3 $(s \underline{\rightarrow} w \wedge s.p = w.p \wedge stable(w)) \Rightarrow stable(s)$

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: Only first two conditions in the definition of happen before are applicable. let s immediately precede w without any intermediate failure or rollback. If s is not made stable in the call to *Log_state* that made w stable, then s must have been made stable in the previous call to *Log_state* as no failure or rollback took place between s and w. If a failure or rollback took place then s is the state that is restored from the stable storage. Hence it is stable.

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 4 $knows\_stable(s, e, j) \Rightarrow stable(get\_state(j, e))$.

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. So 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* }
$index(s.log\_prog[j], e.inc) \geq e.sii$
$\Rightarrow$ { Definition of *index* }
$\exists x : (e.inc, x) \in s.log\_prog[j] \ \wedge \ x \geq e.sii$
$\Rightarrow$ { 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* }
$\exists w : stable(w) \ \wedge \ w.p = j \ \wedge \ get\_state(j, e).c[j].inc = w.c[j].inc \ \wedge \ get\_state(j, e).c[j].sii \leq$

$w.c[j].sii$
$\Rightarrow$ { Lemma 3 }
$stable(get\_state(j, e))$ ■
The following lemma proves a similar result in the other direction.

Lemma 5 $stable(get\_state(j, e)) \Rightarrow \forall i : \exists s \in P_i : knows\_stable(s, e, j)$

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*. So u and w have same incarnation number. Now,
$stable(w)$
$\Rightarrow$ { Above argument, *Log_state* is atomic }
$u.c[j].inc = e.inc \ \wedge \ u.c[j].sii \geq e.sii \wedge u.c[j] \in u.log\_prog[j]$
$\Rightarrow$ { Future inserts maintain the following property }
$\forall v : v.p = j \ \wedge \ u \underline{\rightarrow} v : [\exists x : (e.inc, x) \in v.log\_prog[j] \wedge x \geq e.sii \ ]$
$\Rightarrow$ { *Broadcast_log_prog* is called eventually. }
$\exists log\_prog : broadcast(log\_prog) : [\exists x : (e.inc, x) \in log\_prog[j] \wedge x \geq e.sii]$
$\Rightarrow$ {*Receive_log_prog* is called eventually }
$\forall i : \exists s \in P_i : \exists x : (e.inc, x) \in s.log\_prog[j] \wedge x \geq e.sii$
$\Rightarrow$ { Definition of *index* }
$\forall i : \exists s \in P_i : index(s.log\_prog[j], e.inc) \geq e.sii$
$\Rightarrow$ { 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 fails, it helps in detection of all the orphan states.

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

Proof. Invariant holds trivially if s is same as u. So 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 DV.

Initialize: Invariant is trivially true as 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 state index in the second operation in the routine leaves the invariant unaffected. So it is sufficient to show that the invariant holds after the *max* operation.

In general, DV of w and m can be different as some entries in the DV of m might be set to NULL in the routine *Check_send_buffer*. Happen before relation is defined between w and u, whereas DV of u is updated by taking maximum of DVs of v and m. In order to reason about happen before relation, sometimes we would like to use the DV of w instead of DV of m.

The following claim shows that DVs 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 }
$(w.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (w.c[s.p].sii \geq s.c[s.p].sii)$
$\Rightarrow$ { Definitions of get_state and happen before }
$s \underline{\rightarrow} get\_state(s.p, w.c[s.p])$
$\Rightarrow$ { $\neg stable(s)$, Lemma 3 }
$\neg stable(get\_state(s.p, w.c[s.p]))$
$\Rightarrow$ { Lemma 4 }
$\forall r : \neg knows\_stable(r, s.p, w.c[s.p])$
$\Rightarrow$ { 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 happen 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. }
$(v.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (v.c[s.p].sii \geq s.c[s.p].sii)$
$\Rightarrow$ { $u.c[s.p] = max(v.c[s.p], m.c[s.p])$, case 1 }
$(u.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (u.c[s.p].sii \geq s.c[s.p].sii)$

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 }
$(w.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (w.c[s.p].sii \geq s.c[s.p].sii)$
$\Rightarrow$ { Claim 1: $m.c[s.p] = w.c[s.p]$ }
$(m.c[s.p].inc = s.c[s.p].inc) \ \wedge \ (m.c[s.p].sii \geq s.c[s.p].sii)$

$\Rightarrow$ { $admissible(m, v)$, case 2 }
$(m.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (m.c[s.p].sii \geq s.c[s.p].sii) \;\wedge\; knows\_stable(v, m.c[s.p], s.p)$
$\Rightarrow$ { Definition of *knows_stable* }
$knows\_stable(v, s.c[s.p], s.p)$
$\Rightarrow$ { Lemma 4 }
$stable(s)$

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 w be the state restored and u be the state after taking the actions needed for the recovery. we consider the following two cases: 1) $s.p = u.p$, 2) $s.p \neq u.p$.
Case 1: $s.p = u.p$
$\Rightarrow$ { Definition of happen before }
$s \to u \Rightarrow s \underline{\to} w$
$\Rightarrow$ { w is stable, stability is monotonic }
$stable(s)$

case 2: $s.p \neq u.p$
Now, $s \to u$
$\Rightarrow$ { definition of happen before }
$s \to w$
$\Rightarrow$ { Antecedent: $\neg stable(s)$ }
$s \to w \;\wedge\; \neg stable(s)$
$\Rightarrow$ { Invariant holds before this routine is called }
$(w.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (w.c[s.p].sii \geq s.c[s.p].sii)$
$\Rightarrow$ { Case 2: $s.p \neq u.p$, $w.c$ and $u.c$ differ only in $w.p$'th entry }
$(u.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (u.c[s.p].sii \geq s.c[s.p].sii)$

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) $s \to u$, 2) $u \to s$. 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 }
$(u.c[s.p].inc = s.c[s.p].inc) \;\wedge\; (u.c[s.p].sii \geq s.c[s.p].sii) \;\wedge\; knows\_stable(u, u.c[s.p], s.p)$
$\Rightarrow$ { Definition of *knows_stable* }
$knows\_stable(u, s.c[s.p], s.p)$
$\Rightarrow$ { Lemma 4 }
$stable(s)$ ■

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 $s \not\rightarrow u \Rightarrow ((s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].sii > u.c[s.p].sii))$

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, state index is only increased. Hence the invariant holds in this case. So 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 DV.

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 state index 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 happen before }

$s \not\rightarrow v$

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

$(s.c[s.p].inc \neq v.c[s.p].inc) \vee (s.c[s.p].sii > v.c[s.p].sii)$

$\Rightarrow$ { Case 1 }

$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].sii > u.c[s.p].sii)$

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. So case 1 applies. So 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 w be the state restored and u be the state after taking the actions needed for the recovery.

Now, $s \not\rightarrow u$
$\Rightarrow$ { Definition of happen before }
$s \not\rightarrow w$
$\Rightarrow$ { Invariant holds before this routine is called }
$(s.c[s.p].inc \neq w.c[s.p].inc) \vee (s.c[s.p].sii > w.c[s.p].sii)$
$\Rightarrow$ { Assumption $s.p \neq u.p$, $u.c$ and $w.c$ differ only in $u.p$'th entry }
$(s.c[s.p].inc \neq u.c[s.p].inc) \vee (s.c[s.p].sii > u.c[s.p].sii)$

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 6 $knows\_orphan(s, u) \Rightarrow orphan(u)$.

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

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

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

Lemma 7 $orphan(u) \Rightarrow \forall i : \exists w \in P_i : knows\_orphan(w, u)$

Proof. Our protocol explicitly rolls back a state u only if it satisfies the predicate *knows_orphan*, which means u is dependent on a lost state. So, for our protocol,
$orphan(u) \equiv \exists v : lost(v) \wedge v \underline{\rightarrow} u$.

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. If s and v have different incarnation number then routine *Rollback* is 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. So s and v have the same incarnation number.

Now, $orphan(u)$

$\Rightarrow$ { Definition of orphan for our protocol }

$lost(v) \wedge v \underline{\rightarrow} u$

$\Rightarrow$ { Above argument, *Restart* is atomic, idempotent. }

$lost(v) \wedge v \underline{\rightarrow} u \wedge broadcast(s.c[v.p]) \wedge s.c[v.p].inc = v.c[v.p].inc \wedge v.c[v.p].sii > s.c[v.p].sii$

$\Rightarrow$ { Reliable broadcast includes the execution of *Receive_failure_ann.* }

$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = v.c[v.p].inc \wedge v.c[v.p].sii > e.sii$

$\Rightarrow$ { Invariant 1, not stable v }

$\forall i : \exists w \in P_i : \exists e \in w.iet[v.p] \wedge e.inc = u.c[v.p].inc \wedge u.c[v.p].sii > e.sii$

$\Rightarrow$ { 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 the other processes are the same.

Lemma 8 *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. All routines in the Figure 5 modify volatile memory only. They do not modify the stable storage. Volatile memory is lost in a failure. So a failure during execution of any of these routines is same as the failure before the execution of that routine.

Using the techniques in [20], all routines in Figure 6 are made atomic. They are also idempotent. A repeated execution of any routine in Figure 6 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. So 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 correctly implements the recovery.*

Proof. As per Lemma 8, any failure is equivalent to a failure before executing the routine *Log_state*. So 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. So no state becomes orphan with respect to a failure after the arrival of the corresponding announcement. So the protocol correctly implements the recovery.

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 7, all orphan states satisfy this predicate. Further, by Lemma 6, 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. In fact, any protocol that does not restore the maximum recoverable state is an example of such protocol. Now 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, so 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. At the time of send, sending interval's DV is attached with the message. As lost states are not stable, so their entry in message's DV 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 restores the maximum recoverable state.

Theorem 5 *The protocol restores the maximum recoverable state.*

Proof. Follows from the proof of the Theorem 4.

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

Theorem 6 *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 7, P_i will detect that m is orphan and will discard it. Else w will eventually become stable and by Lemma 5, 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 DV of m will become at most K and m will be sent. ■

The following theorem shows the meaning of K.

Theorem 7 *Given any message m released by its sender, K is the maximum number of processes whose failures can revoke m.*

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. ■