Distributed Recovery with K-Optimistic Logging

Yi-Min Wang Om P. Damani Vijay K. Garg*

Abstract

Fault-tolerance techniques based on checkpointing and message logging have been increasingly used in real-world applications to reduce service downtime. Most industrial applications have chosen pessimistic logging because it allows fast and localized recovery. The price that they must pay, however, is the higher 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 tradeoff 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 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 piggybacked on a message then indicates the number of processes whose failures may revoke the message, and K corresponds to the system-imposed upper bound on the vector size.

1 Introduction

Log-based rollback-recovery [3] is an effective technique for providing low-cost fault tolerance to distributed applications [1, 4, 7, 12]. It is based on the following *piecewise deterministic (PWD) execution model* [12]: process execution is divided into a sequence of *state intervals* each of which is started by a nondeterministic event such as message receipt[1]. The execution within an interval is completely deterministic. During normal execution, each process periodically saves its state on stable storage as a checkpoint. The contents and processing orders of the received messages are also saved on 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 [4]. It can be used either to reduce the amount of lost work due to failures in long-running scientific applications [4], or to enable fast and localized recovery in continuously-running service-providing applications [5].

Depending on when received messages are logged, log-based rollback-recovery techniques can be divided into two categories: pessimistic logging [1, 5] and optimistic logging [12]. Pessimistic logging either synchronously logs each message upon receiving it, or logs all delivered messages before sending a message. It guarantees that any process state from which a message is sent is always recreatable, and therefore no process failure will ever revoke any message to force its receiver to also roll back. This advantage of localized recovery comes at the expense of a higher failure-free overhead. In contrast, optimistic logging first saves messages in a volatile buffer and later writes several messages to stable storage in a single operation. It incurs a lower failure-free overhead due to the reduced number of stable storage operations and the asynchronous logging. The main disadvantage is that messages saved in the volatile buffer may be lost upon a failure, and the corresponding lost states may revoke messages and force other non-failed processes to roll back as well.

Although pessimistic logging and optimistic logging provide a tradeoff between failure-free overhead and recovery efficiency, it has traditionally been only a coarse-grain tradeoff: 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 tradeoff 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

*Yi-Min Wang is with AT&T Labs, Murray Hill, NJ 07974. Om P. Damani is with Dept. of Computer Sciences, University of Texas, Austin, TX 78712. Vijay K. Garg is with Dept. of Electrical and Computer Engineering, University of Texas, Austin, TX 78712. Damani and Garg's research was supported in part by the NSF Grants ECS-9414780, CCR-9520540, and a General Motors Fellowship.

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

scope so that it can recover reasonably fast upon a failure.

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 tradeoff between failure-free overhead and recovery efficiency.

Our approach to deriving the K-optimistic logging protocol is to first prove a fundamental property on minimum transitive dependency tracking. To enable decentralized recovery and efficient output commit, transitive dependency tracking [10] is commonly used to record the highest-index state interval of each process, on which a local process depends. Since we need at least one entry for each process[2], the size of a transitive dependency vector is at least N. In general, transitive dependency tracking does not scale well because a size-N vector needs to be piggybacked on every application message. We introduce the concept of *commit dependency tracking* by proving that *any dependencies on stable state intervals, i.e., intervals that can be reconstructed from information saved in stable storage, can be omitted.* In other words, if process P_j transitively depends on the xth state interval of P_i, and P_i notifies P_j that the interval has become stable, then P_j can remove that entry from its dependency vector. By removing such redundant dependency information, we effectively reduce the size of the dependency vector. We then show that the integer K in the K-optimistic logging protocol is exactly the upper bound on the size of the dependency vector.

The outline of this paper is as follows. Section 2 describes the system model and gives a numerical example for an optimistic logging scheme with asynchronous recovery. Section 3 proves that tracking only dependencies on non-stable states is sufficient for the correct operation of any optimistic logging protocol employing transitive dependency tracking. The same example is then used to demonstrate how the theorem can be applied to reducing the size of dependency vectors. Section 4 motivates and defines the concept of K-optimistic logging, and gives a description of the protocol. Section 5 describes related work, and Section 6 summarizes the paper.

[2] A process may have multiple incarnations, as discussed later.

2 Asynchronous Recovery

We consider distributed applications consisting of N processes communicating only through messages. The execution of each process satisfies the piecewise deterministic (PWD) model. 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 [4], and employs additional optimistic logging. When a checkpoint is taken, all messages in the volatile buffer are also written to stable storage at the same time so that stable state intervals are always continuous. Upon a failure, non-stable state intervals are lost and cannot be reconstructed. Messages sent from those lost intervals become orphan messages. Any process states and messages which causally depend on any such message also become orphan states and orphan messages, respectively. Correct recovery then involves rolling back orphan states and rejecting orphan messages to bring the system back to a globally consistent state[3].

In this section, we first describe a completely asynchronous recovery protocol that piggybacks only dependency information. This protocol has the feature of *completely decoupling dependency propagation from failure information propagation*, and is useful for illustrating the basic concept of asynchronous recovery. A major disadvantage is that it allows potential orphan states to send messages, which may create more orphans and hence more rollbacks. To avoid this disadvantage, existing protocols couple dependency propagation with failure information propagation: the protocol by Strom and Yemini [12] delays the delivery of certain messages until receiving necessary failure information (to be described later); the protocol by Smith et al. [11] piggybacks failure information along with dependency information. We focus on Strom and Yemini's approach in this paper. In the next section, we prove a theorem on omitting redundant dependency tracking and describe how the result can be used to improve their protocol. The improved version then serves as the basis for K-optimistic logging.

We use the example in Figure 1 to illustrate the major components of an asynchronous recovery protocol. Each rectangular box represents a state interval started by a message-delivering event. A shaded box

[3] We do not address the issue of lost in-transit messages [3] in this paper. They either do not cause inconsistency, or they can be retrieved from the senders' volatile logs [12].

indicates that the state interval is *stable*, i.e., can always be recreated from a checkpoint and message logs saved on stable storage. When a process rolls back, it starts a new incarnation [12] (or version [2]), as illustrated by P_1's execution. Each $(t,x)_i$ identifies the interval as the xth state interval of the tth incarnation of process P_i. We use m_i to denote application messages (solid lines), and r_i for rollback announcements that propagate failure information (dotted lines). "Rolling back to state interval u" means rolling back to a checkpoint and reconstructing the process state up to the end of u, while "rolling back state interval v" means the execution within v is undone.

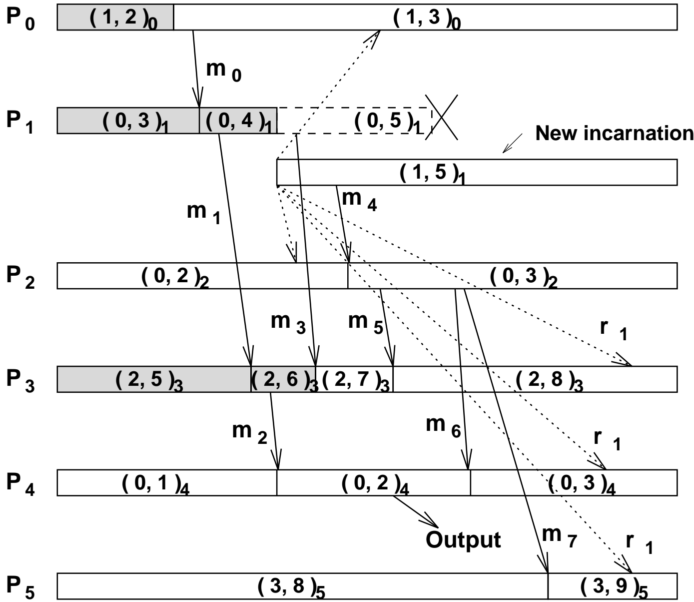

Figure 1: Optimistic logging with asynchronous recovery. (Shaded intervals represent stable state intervals; dotted lines represent rollback announcements.)

We next describe four major components of the protocol.

Dependency tracking: With asynchronous recovery, message chains originating from multiple incarnations of the same process may coexist in the system (with or without FIFO assumption). Therefore, a process needs to track the highest-index interval of every incarnation, that its current state depends on. This can be maintained in the following way: a message sender always piggybacks its dependency information on each outgoing message. Upon delivering a message, the receiver adds the piggybacked dependency to its local dependency. If there are two entries for the same incarnation, only the one with the larger state interval index is retained. For example, when P_4 receives m_2, it records dependency associated with $(0,2)_4$ as $\{(1,3)_0,(0,4)_1,(2,6)_3,(0,2)_4\}$. When it receives m_6, it updates the dependency to $\{(1,3)_0,(0,4)_1,(1,5)_1,(0,3)_2,(2,6)_3,(0,3)_4\}$.

Rollback announcements: When a process P_j fails, it restores the most recent checkpoint and replays the logged messages that were processed after that checkpoint. Then P_j increments its incarnation number and broadcast a rollback announcement (or recovery message [12]) containing *the ending index number of the failed incarnation*. Upon receiving a rollback announcement, a process P_i compares its dependency with that index. If the dependency shows that P_i's state depends on a higher-index interval of any failed incarnation of P_j, P_i rolls back to undo the orphan states, and starts a new incarnation as if it itself has failed [12]. For example, suppose process P_1 in Figure 1 fails at the point marked "X". It rolls back to $(0,4)_1$, increments the incarnation number to 1, and broadcast announcement r_1 containing $(0,4)_1$. When P_3 receives r_1, it detects that the interval $(0,5)_1$ that its state depends on has been rolled back. Process P_3 then needs to roll back to $(2,6)_3$, and broadcast its own rollback announcement. In contrast, when P_4 receives r_1, it detects that its state does not depend on any rolled-back intervals of P_1. In either case, r_1 is saved in an *incarnation end table* so that the process can reject messages from those rolled-back intervals, which may arrive later. Note that, after receiving r_1, P_4 may still need to remember its dependency on $(0,4)_1$ because a future failure of P_0 that rolls back $(1,3)_0$ may force P_1 to announce a new incarnation that invalidates $(0,4)_1$.

Logging progress notification: Each process asynchronously saves messages in the volatile buffer to stable storage. Periodically, it broadcast a logging progress notification to let other processes know which of its state intervals has become stable. Such information is accumulated locally at each process to allow output commit and garbage collection [12]. For example, after P_3 makes the state intervals $(2,5)_3$ and $(2,6)_3$ stable, it can broadcast a notification to let others know that.

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

intervals that it depends on have all become stable [12]. For example, P_4 in Figure 1 can commit the output sent from $(0,2)_4$ after it makes $(0,2)_4$ stable and also receives logging progress notifications from P_0, P_1 and P_3, indicating that $(1,3)_0$, $(0,4)_1$ and $(2,6)_3$ have all become stable. An alternative is to perform output-driven logging by sending additional messages to force the logging progress at P_0, P_1 and P_3 [6].

3 Commit Dependency Tracking

In this paper, we use i, j, k for process numbers, t and s for incarnation numbers, and x and y for state interval indices. In this section, u, v, w, z refer to state intervals, and $P_{v.p}$ refers to the process to which v belongs.

Lamport [8] defined the *happen before* relation for states. Similarly, Johnson and Zwaenepoel [7] defined the happen before relation (or *transitive dependency* relation [10]) for state intervals. Let $u \prec v$ if u and v are intervals of same process and u immediately precedes v. Let $u \rightsquigarrow v$ if a message sent from interval u is delivered to start interval v. Transitive dependency ($\rightarrow$) is defined as the transitive closure of the union of relations $\prec$ and $\rightsquigarrow$. Given any two intervals u and v, if it is possible to determine whether v transitively depends on u ($u \rightarrow v$) then the underlying system is said to be employing *transitive dependency tracking*. Now we can formally define *orphan* as follows.

DEFINITION 1 *A state interval v is orphan if, $\exists u$: rolled_back(u) $\wedge$ $(u \rightarrow v)$.*

Messages sent by orphan states are also called orphans. If the current state of a process is orphan then process itself might be called orphan when there is no confusion.

Traditional asynchronous recovery protocols usually require every non-failed rolled-back process to behave as if it itself has failed [11, 12] by starting a new incarnation and broadcasting a rollback announcement. It was recently observed that, under piecewise deterministic execution model, announcing only failures is sufficient for orphan detection [2]. We give a proof of this observation in Theorem 1, and carry the observation even further by proving, in Theorem 2, that any dependencies on stable intervals can be omitted without affecting the correctness of a recovery protocol which tracks dependencies transitively.

THEOREM 1 *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 $P_{w.p}$. ■

We define that v is *commit dependent on* w if $w \rightarrow v$ and ¬stable(w). That is 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 2 *Commit dependency tracking and failure announcements[4] 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 1, 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 two other sources. First, a rollback announcement containing ending index $(t, x')_i$ can also serve as a logging progress notification that interval $(t, x')_i$ has become stable; Second, when process P_i takes a checkpoint at state interval $(t, x)_i$, it can be viewed as P_i receiving a logging progress notification from itself that interval $(t, x)_i$ has become stable. Since each process execution can be considered as starting with an initial checkpoint, the first state interval is always stable. Corollaries 1, 2 and 3 summarize these results.

[4] Failure announcements are rollback announcements sent by failed processes.

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

COROLLARY 2 *Upon taking a checkpoint and saving all the messages in the volatile buffer to stable storage, a process can omit the dependency entry for its own current incarnation*[5]*.*

COROLLARY 3 *Upon starting the execution, a process has no dependency entry.*

As pointed out earlier, completely asynchronous recovery protocols that decouple dependency propagation from failure information propagation in general need to keep track of dependencies on all incarnations of all processes. Strom and Yemini [12] introduced the following coupling 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.* For example, in Figure 1, P_4 should delay the delivery of m_6 until it receives r_1. After P_4 determines that its state has not become orphan, a lexicographical maximum operation [12] is applied to the two pairs $(0, 4)$ and $(1, 5)$ to update the entry to $(1, 5)$. This update in fact implicitly applies Corollary 1: r_1 notifies P_4 that $(0, 4)_1$ has become stable, and so the dependency on $(0, 4)_1$ can be omitted. The entry can then be used to hold $(1, 5)_1$.

We next describe three modifications to Strom and Yemini's protocol, based on Theorem 1, Theorem 2 and Corollary 1, respectively. The modified protocol then serves as the basis for K-optimistic logging.

Applying Theorem 1: Damani and Garg improved Strom and Yemini's protocol by applying Theorem 1 [2]. Since only failures are announced, the number of rollback announcements and the size of incarnation end tables are reduced. They did not increment the incarnation number on occurrence of non-failure rollback. In this paper, we also apply Theorem 1 but we still require each non-failed rolled-back process to increment its incarnation number. This is necessary for applying Theorem 2 because logging progress notification is on a per-incarnation basis.

Applying Theorem 2: Theorem 2 can be used to omit redundant dependency entries, thereby reducing the size of dependency vector to below N. For example, in Figure 1, when P_4 receives P_3's logging progress notification indicating that $(2, 6)_3$ has become stable, it can remove $(2, 6)_3$ from its dependency vector. If $(2, 6)_3$ is later rolled back due to P_0's failure, P_4's orphan status can still be detected by comparing the entry $(1, 3)_0$ against the failure announcement from P_0.

Applying Corollary 1: Strom and Yemini's protocol waits for the rollback announcement for P_i's $(t-1)^{th}$ incarnation before acquiring a dependency on P_i's t^{th} incarnation. Corollary 1 can be used 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 2, 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 progressive 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 eliminated. For example, when P_5 in Figure 1 receives m_7 which carries a dependency on $(1, 5)_1$, it can deliver m_7 without waiting for r_1 because it has no existing dependency entry for P_1 to be overwritten.

4 K-optimistic Logging

4.1 Motivation

Traditional pessimistic logging and optimistic logging provide only a coarse-grain tradeoff 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 [5].

The above coarse-grain tradeoff, 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

[5] Assume that a separate counter is used to maintain the current state interval index.

triggered boundary conditions, temporary resource unavailability, and bypassable deadlocks. If an application may suffer 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 tradeoff. 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 4).

4.2 The Protocol

Figures 2 and 3 give a complete description of a K-optimistic logging protocol with asynchronous recovery. The protocol is based on Strom and Yemini's protocol with the three improvements described in the previous section. Also, unlike Strom and Yemini's protocol, this protocol does not require FIFO ordering of messages. To simplify presentation by using vector notation and operations, the description always maintains a size-N dependency vector with entries of the form (t, x). When an entry can be omitted, it is represented as setting the entry to NULL which is lexicographically smaller than any non-NULL entry. In an implementation of the protocol, NULL entries can be omitted and any non-NULL entry (t, x) for P_i can be converted to the $(t, x)_i$ form, as used in the previous sections.

The protocol describes the actions taken by a process P_i upon the occurrence of different events. We explain in detail only those parts that are unique to our algorithm. A complete explanation of the generic parts for optimistic logging and asynchronous recovery can be found in previous papers [2, 12]. All routines modifying volatile state are described in Figure 2 and those modifying stable storage are described in Figure 3.

In the variable definition section, the integer K is the degree of optimism known to all processes at **Initialize**. According to Corollary 3, process P_i sets all its dependency vector entries to NULL at **Initialize**, including the ith entry. At **Check_deliverability**, the scheme described at the end of Section 3 is employed: if delivering a message to the application would cause P_i to depend on two incarnations of any process, P_i *waits for the interval with the smaller incarnation number to become stable.* Such information may arrive in the form of a logging progress notification or a failure announcement. When P_i calls **Send_message** for message m, the message is held in a Send_buffer if the number of non-NULL entries in its dependency vector $m.tdv$ is greater than K. The dependency vectors of messages in Send_buffer are updated in **Check_send_buffer** which is invoked by events that can announce new stable state intervals, including (1) **Receive_log** for receiving logging progress notification; (2) **Receive_failure_ann** (according to Corollary 1); and (3) **Checkpoint** (Corollary 2). When a message's dependency vector contains K or less non-NULL entries, it is released by **Check_send_buffer**. Process P_i also needs to check and discard orphan messages in Send_buffer and Receive_buffer upon receiving a failure announcement, as shown in **Receive_failure_ann**.

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.

4.2.1 Properties of the Protocol

We next prove two properties that are unique to the K-optimistic logging protocol.

THEOREM 3 *The protocol implements commit dependency tracking.*

Proof. Given any two state intervals w and v such that $w \rightarrow v$ and $\neg$stable(w), we want to show that the dependency tracking scheme described in the protocol preserves the $w \rightarrow v$ information. Such information can be lost only when the dependency on w is set to NULL, which can happen only when a process receives a notification that w has become stable. Since w is not stable, the $w \rightarrow v$ information could not have been lost, and so the protocol implements commit dependency tracking. ■

THEOREM 4 *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 jth entry of the dependency vector of a message m is set to NULL when

```
type entry : (inc int, ssi int)
var tdv : array[1..N] of entry; /* dep. vector */
    /* log: logging progress notification */
    log : array[1..N] of set of entry;
    /* iet: incarnation end table */
    iet : array[1..N] of set of entry;
    fa : entry; /* failure announcement */
    current: entry; /* current index */
    K: int; /* degree of optimism */

Process P_i :

Initialize :
  ∀ j : tdv[j] = NULL;
  ∀ j : iet[j] = log[j] = {} ;        /* empty set */
  current = (0,1);
  Initialize K;
Receive_message (m) :
  Check_orphan({m}) ;
  if not discarded then
    Check_deliverability({m}) ;
    Receive_buffer = Receive_buffer ∪ {m} ;
Deliver_message (m) :
  /* m is delivered only if m.deliver is true. */
  ∀ j: tdv[j] = max(tdv[j],m.tdv[j]) ;
  current.sii++ ; tdv[i] = current ;
Check_deliverability(buffer) :
  ∀m ∈ buffer : if ∀j : tdv[j].inc ≠ m.tdv[j].inc :
          min(tdv[j],m.tdv[j]) = (t, x)
          ∧ (t, x') ∈ log[j] ∧ x ≤ x'
    then m.deliver = true ;
    else m.deliver = false ;
Check_orphan(buffer) :
  ∀m ∈ buffer : ∀ j :
    if ∃t : (t, x') ∈ iet[j] ∧ t ≥ m.tdv[j].inc
      ∧ x' < m.tdv[j].sii then discard m ;
Send_message(data) :
  put (data,tdv) in Send_buffer ;
  Check_send_buffer ;
Check_send_buffer :
  ∀m ∈ Send_buffer: ∀ j:
    if m.tdv[j] = (t, x) ∧ (t, x') ∈ log[j] ∧ x ≤ x'
      then m.tdv[j] = NULL ;
  ∀m ∈ Send_buffer:
    if Number of non-NULL entries in m.tdv
          is at most K
      then send m ;
```

Figure 2: K-optimistic logging protocol: Part 1.

```
Restart /*after failure */ :
  Restore last checkpoint ;
  Replay the logged messages that follow ;
  fa = current ; current.inc++ ;
  current.sii++ ; tdv[i] = current ;
  Insert(iet[i],fa) ; Insert(log[i], fa) ;
  Synchronously log fa ; Broadcast fa ;
Receive_failure_ann (j, t, x') /* from P_j */:
  Synchronously log the received announcement;
  Insert(iet[j],(t, x')) ; Insert(log[j],(t, x')) ;
  Check_orphan(Send_buffer);
  Check_orphan(Receive_buffer);
  Check_send_buffer ;
  Check_deliverability(Receive_buffer);
  if tdv[j].inc ≤ t ∧ tdv[j].sii > x' then
    Rollback(j, t, x') ;
Rollback(j, t, x') :
  Log all the unlogged messages to the stable storage ;
  Restore the latest checkpoint with tdv such that
    ¬(tdv[j].inc ≤ t ∧ tdv[j].sii > x') ..(I)
  Discard the checkpoints that follow ;
  Replay the messages logged after restored checkpoint
    till condition (I) is not satisfied ;
  Among remaining logged messages, discard orphans
    and add non-orphans to Receive_buffer ;
  /* These messages will be delivered again */
  current.inc++ ; current.sii++ ; tdv[i] = current ;
Checkpoint :
  Log all the unlogged messages ;
  Take checkpoint ;
  Insert(log[i],current) ;
  tdv[i] = NULL ;
  Check_send_buffer ;
Receive_log(mlog) :
  ∀ j,t : (t, x') ∈ mlog[j] : Insert(log[j],(t, x')) ;
  ∀ j: if tdv[j] = (t, x) ∧ (t, x') ∈ log[j] ∧ x ≤ x'
        then tdv[j] = NULL ;
  Check_deliverability(Receive_buffer);
  Check_send_buffer ;
Insert(se, (t, x')) :
  if (t, x'') ∈ se then  /* entry for inc. t exists in se */
    se = (se - {(t, x'')}) ∪ {(t,max(x', x''))}
  else se = se ∪ {(t, x')} ;
```

Figure 3: K-optimistic logging protocol: Part 2.

the corresponding interval in P_j becomes stable. As per proof of Theorem 2, 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. ■

5 Related Work

Rollback-recovery techniques can be classified into two primary categories: checkpoint-based rollback-recovery uses only checkpoints, and log-based rollback-recovery employs additional message logging. In the area of checkpoint-based rollback-recovery, the concept of lazy checkpoint coordination [13] has been proposed to provide a fine-grain tradeoff in-between the two extremes of uncoordinated checkpointing and coordinated checkpointing. An integer parameter Z, called the laziness, was introduced to control the degree of optimism by controlling the frequency of coordination. The concept of K-optimistic logging can be considered as the counterpart of lazy checkpoint coordination for the area of log-based rollback-recovery.

To address the scalability issue of dependency tracking for large systems, Sistla and Welch [10] divided the entire system into clusters and treated intercluster messages as output messages. Lowry et al. [9] introduced the concept of *recovery unit gateways* to compress the vector at the cost of introducing false dependencies. Direct dependency tracking techniques [6, 7, 10] piggyback only the sender's current state interval index, and so are in general more scalable. The tradeoff is that, at the time of output commit and recovery, the system needs to assemble direct dependencies to obtain transitive dependencies [10].

6 Summary

In this paper, we proved a fundamental result in distributed systems recovery: with transitive dependency tracking, dependencies on stable state intervals are redundant and can be omitted. The result naturally lead to a dependency tracking scheme with a variable-size vector carrying only minimum amount of information. By imposing a system-wide upper bound K on the vector size, two things were achieved: first, the vector size does not grow with the number of processes and so the dependency tracking scheme has better scalability; second, 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. Based on the result, we introduced the concept of K-optimistic logging to allow systems to explicitly fine-tune the tradeoff between failure-free overhead and recovery efficiency.

References

[1] 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.

[2] 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.

[3] 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.

[4] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 298–307, 1994.

[5] Y. Huang and Y. M. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 459–463, June 1995.

[6] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 86–95, October 1993.

[7] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *J. Algorithms*, 11:462–491, 1990.

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

[9] A. Lowry, J. R. Russell, and A. P. Goldberg. Optimistic failure recovery for very large networks. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 66–75, 1991.

[10] A. P. Sistla and J. L. Welch. Efficient distributed recovery using message logging. In *Proc. 8th ACM Symposium on Principles of Distributed Computing*, pages 223–238, August 1989.

[11] S. W. Smith, D. B. Johnson, and J. D. Tygar. Completely asynchronous optimistic recovery with minimal rollbacks. In *Proc. IEEE Fault-Tolerant Computing Symp.*, pages 361–370, 1995.

[12] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[13] Y. M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *Proc. IEEE Symp. Reliable Distributed Syst.*, pages 78–85, October 1993.