Message Logging: Pessimistic, Optimistic, and Causal

Lorenzo Alvisi*
Cornell University
Department of Computer Science
Ithaca, NY

Keith Marzullo†
University of California, San Diego
Department of Computer Science and Engineering
La Jolla, CA

Abstract

Message logging protocols are an integral part of a technique for implementing processes that can recover from crash failures. All message logging protocols require that, when recovery is complete, there be no orphan processes*, which are surviving processes whose states are inconsistent with the recovered state of a crashed process. We give a precise specification of the consistency property "no orphan processes". From this specification, we describe how different existing classes of message logging protocols (namely* optimistic*,* pessimistic*, and a class that we call* causal*) implement this property. We then propose a set of metrics to evaluate the performance of message logging protocols, and characterize the protocols that are* optimal *with respect to these metrics. Finally, starting from a protocol that relies on causal delivery order, we show how to derive optimal causal protocols that tolerate f overlapping failures and recoveries for a parameter $f : 1 \leq f \leq n$.*

1 Introduction

Message logging (for example, [BBG83, PP83, SY85, JZ87, SBY88, SW89, JZ90, VJ94, EZ92]) is a common technique used to build systems that can tolerate process crash failures. These protocols require that each process periodically record its local state and log the messages it received after having recorded that state. When a process crashes, a new process is created in its place: the new process is given the appropriate recorded local state, and then it is sent the logged messages in the order they were originally received. Thus, message logging protocols implement an abstraction of a resilient process in which the crash of a process is translated into intermittent unavailability of that process.

All message logging protocols require that the state of a recovered process be consistent with the states of the other processes. This consistency requirement is usually expressed in terms of *orphan processes*, which are surviving processes whose states are inconsistent with the recovered state of a crashed process. Thus, in the terminology of message logging, message logging protocols

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-9003440, DARPA/NSF Grant No. CCR-9014363, NASA/DARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

†This author was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, Contract N00140-87-C-8904 and by AFOSR grant number F49620-93-1-0242. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

must guarantee that there are no orphan processes, either through careful logging or through a somewhat complex recovery protocol.

In this paper, we give a specification of the consistency property "no orphan processes". From this specification, we describe how different existing classes of message logging protocols (namely *optimistic*, *pessimistic*, and a class that we call *causal*) implement this property. We then propose a set of metrics to evaluate the performance of message logging protocols, and characterize the protocols that are *optimal* with respect to these metrics. Finally, starting from a simple protocol that relies on causal multicast, we show how to derive optimal causal protocols that tolerate f overlapping failures and recoveries for a parameter $f : 1 \leq f \leq n$. A thorough discussion of such optimal causal protocols and of the tradeoffs involved in their implementation is presented in [AM94].

The paper proceeds as follows. Section 2 describes the system model commonly assumed for message logging protocols. Section 3 discusses the notion of consistency in message logging protocols. Section 4 presents the derivation of the "no-orphan processes" condition, and explains how pessimistic, optimistic and causal protocols implement it. Section 5 defines optimal message logging protocols. Section 6 shows how a simple message logging protocol that uses causal delivery can be refined to obtain optimal causal protocols. Section 7 concludes the paper.

2 System Model

We assume a system $\mathcal{N}$ of n processes that can communicate only by exchanging messages. The system is asynchronous: there exists no bound on the relative speeds of processes, no bound on message transmission delays, and no global time source. Hence, the *order* in which a process receives messages is nondeterministic. For example, if processes p and q concurrently send messages to process r, then r may receive these messages in either order.

The execution of the system is represented by a *run*, which is an irreflexive partial ordering of the send events, receive events and local events ordered by potential causality [Lam78]. Delivery events are local events that represent the delivery of a received message to the application or applications running in that process. For any message m from process p to process q, q delivers m only if it has received m, and q delivers m no more than once.

We use the *happens-before* relationship of Lamport [Lam78]. For two events e_i of process i and e'_j of process j, $e_i \rightarrow e'_j$ (read "e_i happens before e'_j" or "e'_j causally follows e_i") is the transitive closure of the relation defined by the following two conditions:

- $i = j$ and e_i was executed before e'_j;
- e_i is a send event and e'_j is the corresponding receive event.

At any point in time, the *state* of a process is a mapping of program variables and implicit variables, such as program counters, to their current values. We assume that the state of the process does not include the state of the underlying communication system, such as the queue of messages that have been received but not yet delivered to the process. Consider the states s_p and s_q of two processes p and q, $p \neq q$. We say that s_p and s_q (or, more simply, p and q) are *mutually consistent* if all of the messages from q that p has delivered during its execution up to s_p were sent by q during its execution up to s_q, and vice versa. A collection of states, one from each process, is a *consistent global state* if all pairs of states are mutually consistent [CL85];[1] otherwise it is

[1]This definition is different from that of [CL85] in that it is defined in terms of *deliver* events rather than *receive* events. Our usage corresponds to the literature on message-logging protocols.

inconsistent.

We assume that processes are *piecewise deterministic* [SBY88] in that the only nondeterminism in a process arises from the nondeterministic order in which messages are received. It is therefore natural to think of the execution of a process as being partitioned into intervals, with the beginning of each interval being defined by the initial state of the process or the delivery of a message. Such an interval is called a *state interval.* Thus, given the first state of a state interval and the message whose delivery defines the beginning of the interval, the rest of the states in the interval are uniquely determined by the process.

For any message m delivered by process p, the *receive sequence number* of m, denoted $m.rsn$, represents the order in which m was delivered: $m.rsn = \ell$ if m is the ℓ^{th} message delivered by p [SY85]. The state interval that initiates with the delivery of m is denoted $p[\ell]$ where ℓ, the *index* of $p[\ell]$, is equal to $m.rsn$. The state interval $p[0]$ is defined to be the interval of states of p from its initial state to the state immediately before the delivery of the first message.

We further assume that:

- Processes fail independently according to the fail-stop model [Sch84];
- There exists common knowledge on the identity of the fixed set of processes that belong to the system;
- Channels are point-to-point, FIFO, and fail by intermittently losing messages.

3 Consistency in Message Logging Protocols

In message logging protocols, each process typically records both the content and receive sequence number of all the messages it has delivered in a location (called a *message log*) that will survive the failure of the process. This action is called *logging.* The process may also periodically create a *checkpoint* of its local state, thereby allowing message logs to be trimmed. For example, in some message logging protocols once process p knows that state $p[\ell]$ is checkpointed, all messages m such that $m.rsn \leq \ell$ can be removed from p's message log. Note that the periodic checkpointing of a process' state is only needed to bound the length of its message log (and hence the recovery time). For simplicity we ignore checkpointing in this paper.

Logging a message may take time: therefore, there is a natural design decision of whether or not a process should wait for the logging to complete before delivering the message to the application. For example, suppose that having delivered message m, process p sends message m' to process q, and q delivers m'. If message m were not logged by the time p sent m', then the crash of p may cause information about m to be lost. Then, when a new process p is initialized and replayed logged messages, p may follow an execution in which it does not send m' to q. Hence, process q would no longer be consistent with p, because q would have delivered a message that was not sent by the current process p. Such a process q is called an *orphan.* Protocols that can create orphans are called *optimistic* because they are willing to take the (hopefully) small risk of creating orphans in exchange of better performance during failure-free runs. If a failure occurs, though, an optimistic protocol must determine whether there are any orphans, and roll them back in order to make the states of the processes consistent again.

A *pessimistic* protocol is one in which each process p never sends a message m' until it knows that all messages delivered before sending m' are logged. Pessimistic protocols will never create orphans, and so the reconstruction of the state of a crashed process is very straightforward as compared to optimistic protocols. On the other hand, pessimistic protocols potentially block a process for each message it receives. This can slow down the throughput of the processes even when no process ever crashes.

There are message logging protocols that do not exhibit the above tradeoff: they neither create orphans when there are failures nor do they ever block a process when there are no failures. We call such protocols *causal* message logging protocols, for reasons that will become clear later. Examples of this class are presented in [EZ92] and [AHM93].

All message logging protocols must address the problem of recovering a process that communicates with the environment, such as sensors, terminals and actuators. It is natural to model such communication in terms of the sending and delivery of messages. Because one end of the communication is not a process, recovery of a process with respect to these kinds of messages is usually done differently than with respect to messages between processes. For input, the data from the environment must be stored in a location that is accessible for the purpose of replay. For output, a process must be in a recoverable state before sending any message to the environment. This means that, in general, even optimistic message logging protocols may block before sending a message to the environment. Such issues are outside of the scope of this paper.

4 Specification of Message Logging

Let ρ be a run of $\mathcal{N}$, and assume that no process makes any nondeterministic choices in ρ. If a process were to fail, then recovering the system to a consistent state after it fails would be straightforward: the system, restarted from its initial state, would re-execute ρ. If processes did make nondeterministic choices in ρ, however, then the same choices that were made originally must be made again during the re-execution of ρ. A message logging protocol can be thought of as a technique for recording such non-deterministic choices. The logged information can then be used during recovery to force re-execution to make the same nondeterministic choices, and hence restore the values of any lost states.

With the assumption that processes are piecewise deterministic, the only non-deterministic choices made during ρ concern the order in which each message is delivered to each process. Hence, we need to represent the nondeterministic choice made by each delivery event. We assume that a message has a single destination, but the following can be easily extended to include unreliable multicast as well.

For each message m delivered during ρ, let $m.source$ and $m.ssn$ denote, respectively, the identity of the sender process and a unique identifier assigned to m by the sender. The latter may, for example, be a sequence number. Let $deliver_{m.dest}(m)$ denote the event that corresponds to the delivery of message m by process $m.dest$. The tuple $\langle m.source, m.ssn, m.rsn\rangle$ unequivocally determines m and the order in which m was delivered by $m.dest$. We refer to this tuple as the *determinant* of the event $deliver_{m.dest}(m)$ and we denote it as $\#m$. The ability of any message logging protocol to correctly recover from process failures experienced during a run ρ therefore crucially depends on the availability upon recovery of the determinants of the delivery events that occurred during ρ.

In contrast, the immediate availability of the *content* $m.data$ of a message m exchanged during ρ is not critical for the correctness of a message logging protocol. In practice, an effective technique for logging the content of messages can result in dramatic performance gains, but, at least in principle, the content of a message can be deterministically regenerated from the initial states of the processes and the determinants of each delivery event in ρ. In other words, message logging protocols are not really about logging messages, but rather about logging message determinants. Hence, we will ignore in our specification the specifics of logging the contents of messages, just as we ignore the details related to taking periodic checkpoints— both are caching that can be added to speed up recovery.

Let $Depend(m)$ denote the set of processes whose state reflects the delivery of message m. Formally,

$$Depend(m) \stackrel{\text{def}}{=} \{j \in \mathcal{N} \mid (j = m.dest) \vee$$
$$(\exists \text{ an event } e_j \text{ of process } j\text{: } (deliver_{m.dest}(m) \rightarrow e_j))\}$$

Let $Log(m)$ denote the smallest set of non-crashed processes such that, if all the members of $Log(m)$ crash, then the determinant of $deliver_{m.dest}(m)$ is lost. If processes fail independently and logs are kept in volatile memory, then $Log(m)$ denotes the set of processes that maintain a copy of $\#m$ in their address space:[2] in particular, process $m.dest$ is a member of $Log(m)$ once it delivers m. Some message logging protocols take advantage of *stable storage*, a system utility used to atomically and reliably log data for later retrieval [Gra77]. If the determinant of $deliver_{m.dest}(m)$ is logged in stable storage, then any subset of the processes in $\mathcal{N}$ can crash without $\#m$ being lost. In this case, we will say that the size of $Log(m)$, written $|Log(m)|$, is greater than n. Our specification is written such that if $|Log(m)| > n$ then the members of $Log(m)$ need not be defined. Hence, our specification applies both to message logging protocols that use stable storage and those that do not use stable storage.

Suppose that a set of processes $\mathcal{C} \subseteq \mathcal{N}$ fails. The determinant of a delivery event $deliver_{m.dest}(m)$ is lost if $Log(m) \subseteq \mathcal{C}$. Because $m.dest \in Log(m)$, the process $m.dest$ that originally delivered m is among the set of crashed processes and will therefore execute a recovery procedure. Since the determinant of m was lost, though, $m.dest$ may not know in what order m was originally delivered and so may not re-execute the same set of instructions that it did originally. A process p is an *orphan process* if its state depends on $m.dest$ being able to exactly re-execute the same set of instructions:

$$p \textit{ orphan process of } \mathcal{C} \stackrel{\text{def}}{=} (p \in \mathcal{N} - \mathcal{C})$$
$$\wedge \exists m : ((p \in Depend(m)) \wedge (Log(m) \subseteq \mathcal{C})) \tag{1}$$

Negating Equation 1 and quantifying over all p gives the following sufficient condition for there being no orphans created by the crash of a set of processes $\mathcal{C}$:

$$\forall m : ((Log(m) \subseteq \mathcal{C}) \Rightarrow (Depend(m) \subseteq \mathcal{C})) \tag{2}$$

We require that no set of crashed processes results in the creation of orphans. Quantifying 2 over all $\mathcal{C}$, we obtain:

$$\forall m : (Depend(m) \subseteq Log(m)) \tag{3}$$

Property 3 is not necessary for orphan-freedom, however, when the size of $\mathcal{C}$ can be *a priori* bounded by a constant f. The determinant of a delivery event $deliver_{m.dest}(m)$ is *stable* when $\#m$ cannot become lost due to process crashes. Hence, Property 3 need hold only when $\#m$ is not stable:

$$\forall m : ((|Log(m)| \leq f) \Rightarrow ((Depend(m) \subseteq Log(m)) \tag{4}$$

[2]One can imagine protocols where no single process knows the value of $\#m$ but a set of processes collectively do. An example of such a protocol is given in [JV87] where the value of $\#m$ for some message m may be inferred by "holes" in the sequence of logged receive sequence numbers. Since this is the only such case that we know of and its utility is limited to only $f = 2$, we do not consider this more general method of logging in this paper.

Different classes of message logging protocols can be characterized with respect to Property 4. Optimistic protocols assume that the interval of time during which ($|Log(m)| \leq f$) holds is brief, and so Property 4 holds nearly all the time. In the unlikely event that failures occur when Property 4 does not hold, the recovery protocol can diminish $Depend(m.dest)$ by rolling back processes with orphan states until $Depend(m.dest) \subseteq \mathcal{C}$ and Property 2 holds. Thus, optimistic message logging protocols implement the following weaker property:

$$\forall m : ((|Log(m)| \leq f) \Rightarrow \\ ((Log(m) \subseteq \mathcal{C}) \Rightarrow \Diamond(Depend(m) \subseteq \mathcal{C}))) \tag{5}$$

where $\Diamond$ is the temporal "eventually" operator [Pnu77].

Pessimistic message logging protocols, however, do not admit orphans and so must implement Property 4. In particular, they implement the following property:

$$\forall m : ((|Log(m)| \leq f) \Rightarrow (|Depend(m)| \leq 1)) \tag{6}$$

The reasoning behind pessimistic protocols starts from the same observation used by optimistic protocols: the time during which $|Log(m)| \leq f$ holds is brief. During this interval, the process that delivered m is not allowed to send any application messages, since their delivery would make $|Depend(m)| > 1$. [3] However, it is argued that the process that delivered m is unlikely to wish send an application message during this brief interval, and so any performance loss will most likely be small.

Property 4 allows for $Log(m)$ to grow arbitrarily larger than $Depend(m)$ and allows for protocols that disseminate a large number of unnecessary copies of $\#m$. As the number of delivery events performed during a run increases, these extra copies may end up wasting a significant portion of the address spaces of the processes in the system. In order to address this problem, one may consider protocols that implement the following stronger property:

$$\forall m : ((|Log(m)| \leq f) \Rightarrow (Depend(m) = Log(m))) \tag{7}$$

Protocols implementing Property 7 successfully limit the size of $Log(m)$ with respect to $Depend(m)$, but in such a protocol a process must simultaneously join $Log(m)$ and $Depend(m)$. Enforcing this requirement would be unnecessarily complicated. Thus, a third class of message logging protocols implements a weaker strengthening of Property 4 that still bounds $Log(m)$:

$$\forall m, p : ((|Log(m)| \leq f) \Rightarrow ((Depend(m) \subseteq Log(m)) \wedge \\ (p \in Log(m) \Rightarrow \Diamond(p \in Depend(m) \vee p \notin Log(m)))) \tag{8}$$

Property 8 strongly couples logging with causality. It says that all processes whose state causally depends on $deliver_{m.dest}(m)$ have logged $\#m$, and that if process p has logged a copy of $\#m$ and does not leave $Log(m)$ (for instance, by crashing), then p's state will eventually causally depend on $deliver_{m.dest}(m)$. Hence, we call the protocols in this class *causal message logging protocols.* Family-Based-Logging [AHM93] and Manetho [EZ92] are two examples of causal message logging protocols for the special cases $f = 1$ and $f = n$ respectively.

[3] In pessimistic sender based logging [JZ87] process $m.dest$ increases $|Log(m)|$ by sending the value of $m.rsn$ to process $m.source$, piggybacked on the acknowledgment of message m. Notice however that $m.dest$ is still not allowed to send any *application* messages until it is certain that $m.source$ has become a member of $Log(m)$.

5 Optimal Message Logging Protocols

Let Π be a non-fault-tolerant protocol run by a set of processes $\mathcal{N}$, and let Π_μ denote Π combined with a message logging protocol μ that tolerates process crashes and transient link failures. We consider the following four metrics for comparing message logging protocols:

1. *number of forced roll-backs*: Consider a run ρ of Π, and suppose that a set $\mathcal{C} \subseteq \mathcal{N}$ of processes crash. After the failure, all correct process have their state intact, but the system will not in general be in a consistent global state. Consider now the same run ρ with Π_μ instead of Π. We say that μ forces r roll-backs if, for all ρ, μ requires at most r correct processes to roll back their state in order to build a consistent global state upon recovery. Clearly, a lower bound for r is 0; furthermore, the existence of pessimistic protocols proves that the bound is tight.

2. *k–blocking*: Consider a message m delivered by process $m.dest$, and let e be the first send event of process $m.dest$ that causally follows $deliver_{m.dest}(m)$. We say that a message logging protocol is k–blocking if, in all failure-free runs and for all messages m, process $m.dest$ delivers no less than k messages between $deliver_{m.dest}(m)$ and e. For example, pessimistic sender-based logging [JZ87] is 1–blocking because process $m.dest$ must receive a message acknowledging the logging of $m.rsn$ before sending a message subsequent to the delivery of m. Optimistic protocols are, by design, 0–blocking. For purposes of completeness, we define pessimistic protocols that log the determinant of m in local stable storage to be 1–blocking, since there is one event (the acknowledgment of $\#m$ being written to stable storage) that must occur between $deliver_{m.dest}(m)$ and a subsequent send [BBG83]. Clearly, a lower bound for k is 0; furthermore, the existence of optimistic protocols proves that the bound is tight.

3. *number of messages*: Protocol Π can be transformed into a protocol Π_ℓ that tolerates transient link failures by a simple acknowledgment scheme: whenever process $m.dest$ receives a message m generated by Π, it sends an acknowledgment to process $m.source$. Process $m.source$ continues to send m until it receives such an acknowledgment. For any given failure-free run ρ, protocol Π_ℓ will send twice the number of messages that Π sends.

   Suppose now that ρ is run using Π_μ instead of Π_ℓ. We say that μ sends additional messages in ρ if Π_μ sends more messages than Π_ℓ. For example, in order to tolerate single crash failures pessimistic sender based logging [JZ87] potentially requires that one extra acknowledgment be sent for each message exchanged in the system. Clearly, a lower bound is to send no additional messages, and the existence of optimistic protocols proves that the bound is tight.

4. *size of messages*: Consider any message m sent in a run of Π_ℓ, and let m_μ be the equivalent message sent in the equivalent failure-free run of Π_μ. Let $|m|$ and $|m_\mu|$ be the amount of data carried in m and m_μ, respectively. We will say that μ sends a additional data if the maximum value of $|m_\mu|$ - $|m|$ is a for all such m and m_μ taken from all pairs of equivalent runs. For example, for optimistic protocols tracking direct dependencies a is a constant [SW89, VJ94], while for optimistic protocols tracking transitive dependencies a is proportional to the number of processes in the system [SY85, SW89]. Clearly, a lower bound for a is 0, and pessimistic receiver-based logging proves that the bound is tight.

Note that there is a tradeoff between the number of additional messages sent in Π_μ and the size of the messages—a message logging protocol μ can include less information on a message by sending additional messages containing the extra data. Since there is, to a point, a performance benefit in sending few large messages instead of several small messages, we will consider keeping the number of additional messages small at the expense of the size of the messages. Hence, we define an *optimal message logging protocol* as a message logging protocol that is 0–blocking, introduces 0 forced roll-backs, and sends no additional messages. Notice that protocols that are optimal according to our definition do not necessarily outperform in practice non-optimal protocols. For instance, a message may become large enough that the underlying network will need to break it into separate packets, thereby significantly increasing the overhead of the protocol. In addition, other issues that are difficult to quantify, such as the cost of output commit, must be taken into consideration in assessing the performance of a message logging protocol [EZ94]. Nevertheless, the protocols that we call optimal are unique in optimally addressing the theoretical *desiderata* of the message logging approach.

6 Deriving an Optimal Causal Protocol

Family-Based-Logging [AHM93] and Manetho [EZ92] are two examples of optimal causal protocols for the special cases $f = 1$ and $f = n$ respectively. We now show how optimal causal protocols for arbitrary values of f can be derived by giving the outline of a refinement starting from a message logging protocol that uses *causal delivery order* [BSS91].

Let $send_p(m)$ to q denote the event whereby process p sends message m to process q. FIFO order ensures that the following property holds:

$$\begin{aligned}&\forall p, r, m_1, m_2 : send_p(m_1) \text{ to } r \rightarrow send_p(m_2) \text{ to } r \Rightarrow \\ &\qquad deliver_r(m_1) \rightarrow deliver_r(m_2)\end{aligned} \tag{9}$$

Causal delivery order strengthens FIFO delivery order:

$$\begin{aligned}&\forall p, q, r, m_1, m_2 : send_p(m_1) \text{ to } r \rightarrow send_q(m_2) \text{ to } r \Rightarrow \\ &\qquad deliver_r(m_1) \rightarrow deliver_r(m_2)\end{aligned} \tag{10}$$

Consider the following protocol:

- All messages are delivered according to causal delivery order.
- Whenever a process p receives an application message m and assigns it a receive sequence number $m.rsn$, p logs $\#m$ in its local volatile storage before delivering m. p then sends a message containing $\#m$ to all other processes before sending any further application messages.
- When a process receives a determinant $\#m$, it logs the determinant in its local volatile storage.

This protocol satisfies Property 3: process p becomes a member of $Depend(m)$ only after having added $\#m$ to its volatile log, and causal delivery order ensures that no process delivers an application message multicast causally after $deliver_p(m)$ unless it has received and logged $\#m$.

The protocol has several limitations. First, it is not optimal, since for each delivery event $deliver_p(m)$ it sends the additional messages necessary to disseminate $\#m$. Second, it does not satisfy Property 8, since a process may enter $Log(m)$ by receiving the multicast of $\#m$, without ever becoming a member of $Depend(m)$. Finally, the protocol forces all application messages to be delivered using causal delivery order, even though causal delivery is only used to regulate the dissemination of determinants with respect to application messages.

However, this protocol can be used as the starting point of a refinement to a more efficient protocol by choosing an instance of a causal delivery order protocol. For example, the class of optimal causal message logging protocols described in [AM94] can be derived by choosing the original ISIS causal multicast protocol [BJ87].

Consider Property 10. If q does not know that m_1 has already reached r, then q can include a copy of m_1 piggybacked on m_2. In general, q will piggyback to r any message m' that was sent in the causal past such that q does not know that r has a copy of m'. Then, Property 10 is implemented by having r deliver such piggybacked messages that are destined to r but have not yet been delivered before r delivers m_2.

As a first step in the refinement process, consider the effects of using this piggybacking scheme to implement causal delivery in our message logging protocol. In particular, implement the sending of $\#m$ to all other processes not by sending additional messages, but rather by always piggybacking $\#m$ on any further application messages.

The resulting protocol successfully addresses the first two limitations of the original protocol. By piggybacking $\#m$ on application messages, it ensures that only processes that will enter $\mathit{Depend}(m)$ will join $\mathit{Log}(m)$. Furthermore, the protocol is optimal in the number of messages it sends because the determinants are piggybacked on existing application messages.

Note that once the piggybacking scheme is in place, there is no need to require that application messages be delivered using causal delivery order. The resulting protocol is both optimal and causal, and can tolerate n overlapping failures without violating Property 8.

The last step of the refinement process produces a protocol that scales to tolerate $f : 1 \leq f \leq n$ overlapping failures: a process piggybacks $\#m$ on an application message only when it does not know that $|\mathit{Log}(m)| > f$.

The resulting protocol ensures the availability of the information necessary to recover from up to f overlapping failures, but leaves many open questions. Two important questions are how to represent $|\mathit{Log}(m)|$ and evaluate the condition $|\mathit{Log}(m)| \leq f$ efficiently, and how to use the logged information to perform recovery. We discuss these issues in [AM94], in which we present a class of optimal causal protocols that tolerate f overlapping failures and recoveries for a parameter $f : 1 \leq f \leq n$. These protocols are based on the same piggybacking scheme given above.

7 Conclusions

We have presented the first specification of the message logging approach to fault-tolerance, by precisely defining the "no-orphan processes" property, which is at the foundation of the consistency requirements of all the protocols in this class. The specification is general enough to describe all existing classes of message logging protocols, yet simple enough to be effectively used as a tool to derive more efficient protocols. We have provided a set of metrics to evaluate the performance of a message logging protocol, and given a set requirements that a protocol must meet in order to be optimal. Finally, we have argued the existence of optimal protocols that tolerate f overlapping failures and recoveries for a parameter $f : 1 \leq f \leq n$ by showing how such a protocol can be derived starting from a simple protocol that relies on causal delivery order. A thorough discussion of optimal protocols and of the tradeoffs involved in their implementation is presented in [AM94].

Acknowledgments We would like to thank Bruce Hoppe, Fred Schneider and Sam Toueg for many discussions on this research and on early versions of this paper. We would also like to thank Matthew Clegg and Jeremy Sussman for their comments on drafts of this paper.

References

[AHM93] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, June 1993.

[AM94] Lorenzo Alvisi and Keith Marzullo. Optimal message logging protocols. Technical Report TR94-1457, Cornell University Department of Computer Science, September 1994.

[BBG83] Anita Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles*, pages 90–99. ACM SIGOPS, October 1983.

[BJ87] Kenneth Birman and Tommy Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(2):47–76, February 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer System*, 9(3):272–314, August 1991.

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

[EZ92] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[EZ94] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers: 24 Annual International Symposium on Fault-Tolerant Computing*, pages 298–307. IEEE Computer Society, June 1994.

[Gra77] James N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmueller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1977. Lecture Notes on Computer Science 60.

[JV87] T.Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461. IEEE Computer Society, June 1987.

[JZ87] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14–19. IEEE Computer Society, June 1987.

[JZ90] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

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

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 46–57, November 1977.

[PP83] M.L. Powell and D.L. Presotto. Publishing: A reliable broadcast communication mechanism. In *Proceedings of the Ninth Symposium on Operating System Principles*, pages 100–109. ACM SIGOPS, October 1983.

[SBY88] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[SW89] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proceedings of the Eighth Symposium on Principles of Distributed Computing*, pages 223–238. ACM SIGACT/SIGOPS, August 1989.

[SY85] R. B. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.

[VJ94] S. Venkatesan and T.Y. Juang. Efficient algorithms for optimistic crash recovery. *Distributed Computing*, 8(2):105–114, June 1994.