Message Logging: Pessimistic, Optimistic, Causal, and Optimal

Lorenzo Alvisi and Keith Marzullo

Abstract—Message-logging protocols are an integral part of a popular 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 $f: 1 \leq f \leq n$.

Index Terms—Message logging, optimistic protocols, pessimistic protocols, checkpoint-restart protocols, resilient processes, specification of fault-tolerance techniques.

1 Introduction

MESSAGE-LOGGING protocols (for example, [4], [15], [21], [11], [17], [20], [12], [22], [7] are popular for building 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 once a crashed process recovers, its state is 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, message-logging protocols guarantee that upon recovery, no process is an orphan. This requirement can be enforced either by avoiding the creation of orphans during an execution, as *pessimistic* protocols do, or by taking appropriate actions during recovery to eliminate all orphans as *optimistic* protocols do.

This paper examines more carefully the different approaches to message-logging protocols. To do so, we characterize more precisely the property of having no orphans. This characterization is surprisingly simple yet it is general enough to describe both pessimistic and optimistic message-logging protocols. Both classes of protocols, however, have drawbacks, which are apparent from their characterizations. Thus, we use the same characterization to define a third class of protocols that we call *causal* message-logging protocols. These protocols are interesting because they do not suffer from the drawbacks of the other two classes and can be made optimal with respect to a set of reasonable performance metrics.

Finally, starting from a simple but inefficient message-logging protocol that uses causal multicast, we derive an optimal causal protocol that maintains enough information to tolerate *f* overlapping failures, where *f* is a parameter of the protocol. We only derive this protocol far enough to illustrate how piggybacking can be used to implement causal message-logging. A thorough discussion of such optimal causal protocols and of the tradeoffs involved in their implementation is presented in [3].

The characterization that we give is concerned with how information can be lost in the system due to crashes, and how message-logging protocols cope with this loss of information. There are other issues of message-logging protocols that are out of the scope of this paper, such as communication with the environment, checkpointing, and recovery. A discussion of these issues in the context of causal message logging is presented in [2].

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 *always-no-orphans condition*, and explains how it relates with the consistency conditions implemented by pessimistic, optimistic and causal protocols. Section 5 defines optimal message-logging protocols. In Section 6, we present a simple nonoptimal protocol that uses causal delivery to implement the always-no-orphans consistency condition. In Section 6.3 we refine this protocol to obtain an optimal causal message-logging protocol. Section 7 concludes the paper.

• *L. Alvisi is with the University of Texas at Austin, Department of Computer Sciences, Austin, TX 78712. E-mail: lorenzo@cs.utexas.edu.*
• *K. Marzullo is with the Department of Computer Science and Engineering, University of California at San Diego, La Jolla, CA 92093. E-mail: marzullo@cs.ucsd.edu.*

Manuscript received 7 June 1996; revised 19 Nov. 1997.
Recommended for acceptance by A. Shaw.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 101222.

2 System Model

Consider a distributed system $\mathcal{N}$ consisting of n processes. Processes 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.

Processes execute events, including *send* events, *receive* events, and local events. These events are ordered by the irreflexive partial order *happens before* $\rightarrow$ that represents potential causality [13]. We assume that a send event specifies only a single destination, but the following could be easily extended to include multiple destination messages. We also assume that the channels between processes do not reorder messages and can fail only by transiently dropping messages.

Processes can fail independently according to the fail-stop failure model [18]. In this model, processes fail by halting, and the fact that a process has halted is eventually detectable by all surviving processes.

Outside of satisfying the happens-before relation, the exact order a process executes receive events depends on many factors, including process scheduling, routing, and flow control. Thus, a recovering process may not produce the same run upon recovery even if the same set of messages are sent to it since they may not be redelivered in the same sequence as before. We represent this low-level nondeterminism by creating a nondeterministic local event *deliver* that corresponds to the delivery of a received message to the application. When an deliver event is executed, any received message can be chosen to be delivered to the application. The only constraints are that a message can be delivered only once, channel order is preserved, and as long as a process does not fail, it eventually delivers all messages that it receives.

A deliver event assigns to a message m a *receive sequence number* that encodes the order in which m was delivered.[1] We denote the receive sequence number of a message m as $m.rsn$. Thus, if process p delivers m and $m.rsn = \ell$ then m is the ℓth message that p delivered [21].

An execution of the system is represented by a *run*, which is the sequence of states the system passes through during that execution. Each state consists of the individual process states, each of which is a mapping from program variables and implicit variables (such as program counters) to values. The state of a process does not include the variables defined in the underlying communication system, such as the queues of messages that have been received but not yet delivered to processes. We assume that only one process changes state between any two adjacent states in the run. Thus, each pair of adjacent states defines an event that was executed by a process. The resulting sequence of events is consistent with the happens before relation.

A *property* is a logical expression evaluated over runs. We use the two temporal operators $\Box$ and $\Diamond$ [14] to express properties. The property $\Box\Phi$ (read "always Φ") at a state in a run means that the property Φ holds in the current state and all subsequent states of the run. The property $\Diamond\Phi$ (read "eventually Φ") at a state in a run means that the property Φ holds in the current state or in a subsequent state of the run. If no state and run are specified, then a property holds if it holds in the initial state of all runs of the system. For example, the property $\Box((a = 1) \Rightarrow \Diamond (b = 1))$ means that in all runs, for every state in a run, if the state satisfies $a = 1$, then the state or a subsequent state in the run satisfies $b = 1$.

1. A more logical name would be the *deliver sequence number*. We use *receive sequence number* in order to be consistent with message logging literature.

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 into a location (called a *message log*) that will survive the failure of the process [9]. This action is called *logging*. To trim message logs, a process may also periodically create a *checkpoint* of its local state. For example, in some message-logging protocols once a process p checkpoints its state, all messages delivered before this state can be removed from p's message log. Note that the periodic checkpointing is only needed to bound the length of message logs (and hence the recovery time of processes). 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 sending another message. For example, suppose that having delivered message m, process p sends message m' to process q, and q delivers m'. If message m is not logged by the time p sends m', then the crash of p may cause information about m to be lost. When a new process p is initialized and replayed logged messages, p may follow a different run in which it does not send m' to q. In this case, 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 for better performance during failure-free runs. If a failure occurs, though, an optimistic protocol determines whether there are any orphans. If so, it rolls them back in order to make the states of the processes consistent again.

A *pessimistic* protocol is one in which no process p ever sends a message m' until it knows that all messages delivered before sending m' are logged. Pessimistic protocols never create orphans, and so reconstructing the state of a crashed process is 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 [7], [1].

All message-logging protocols must address the issue of processes that communicate with the environment. It is natural to model such communications in terms of the sending and delivery of messages. Because one end of the communications 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 $\mathcal{N}$. If no process executes nondeterministic events in ρ and a process fails, then recovering the system to a consistent state is straightforward: The system can stop and restart from its initial state. Since ρ contained only deterministic events, ρ would be re-executed. If processes can execute nondeterministic events in ρ, then some mechanism is needed to guarantee that the same events executed in ρ are again executed during recovery. Message logging protocol implement such a mechanism by recording the information necessary to deterministically re-execute events of ρ.

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*, and let *m.data* be the application data carried by the message. The tuple $\langle m.source, m.ssn, m.dest, m.rsn\rangle$ determines m and the order in which m was delivered by *m.dest* relative to the other messages delivered by *m.dest*. We call this tuple the *determinant* of event $deliver_{m.dest}(m)$, and we denote the tuple as $\#m$. For simplicity, we also refer to $\#m$ the determinant of m.

Therefore, the ability of any message logging protocol to correctly recover from process failures experienced during ρ depends crucially on the availability upon recovery of the determinants of the delivery events that occurred during ρ.

However, the immediate availability of the content *m.data* of a message m exchanged during ρ is not an essential part of $\#m$. 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 deliver event in ρ. In other words, logging messages is not an essential task of message logging protocols, while logging message determinants is. 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.[2]

Our goal is to derive a condition, which we call *always-no-orphans*, that guarantees that no orphans are generated during any run of $\mathcal{N}$. To do so, we define two subsets of $\mathcal{N}$ for each message m delivered in a run. The first set, *Depend*(m), contains all processes whose state reflects delivery of m: *Depend*(m) contains the destination of message m, plus any process that delivered a message sent causally after the delivery of m. Formally:

$$Depend(m) \overset{\text{def}}{=} \left\{ j \in \mathcal{N} \,\middle|\, \begin{array}{l} \vee\ \ ((j = m.dest) \wedge j \text{ has delivered } m) \\ \vee\ \ (\exists m': (deliver_{m.dest}(m) \rightarrow deliver_j(m'))) \end{array} \right\}$$

The second set, *Log*(m), is used to characterize the set of processes that have a copy of $\#m$ in their volatile memory.[3] Process *m.dest* becomes a member of *Log*(m) when it delivers m.

Suppose that a set of processes $C \subseteq \mathcal{N}$ fail. The determinant of a deliver event $deliver_{m.dest}(m)$ is lost if $Log(m) \subseteq C$. Assume that $\#m$ is lost. By definition, process *m.dest* was in *Log*(m) before the failure. Therefore, *m.dest* is in C and so must be recovered. Since $\#m$ is lost, *m.dest* may not be able to deliver m in the correct order during recovery. Consequently, *m.dest* may not be able to regenerate some of the messages it sent after executing $deliver_{m.dest}(m)$ in the original execution. All processes whose state depends on process *m.dest* delivering message m in the correct order will become orphan processes.

We say that a process p becomes an orphan of C when p itself does not fail and p's state depends on the delivery of a message m whose determinant has been lost. Formally:

$$p \text{ orphan of } C \overset{\text{def}}{=} \left(\begin{array}{ll} \wedge & (p \in \mathcal{N} - C) \\ \wedge & \exists m: ((p \in Depend(m)) \wedge (Log(m) \subseteq C)) \end{array} \right) \quad (1)$$

Negating (1) and quantifying over p gives the following necessary and sufficient condition for there being no orphans created by the failure of a set of processes C:

$$\forall m: ((Log(m) \subseteq C) \Rightarrow (Depend(m) \subseteq C)) \quad (2)$$

Our goal is to derive a property that guarantees that no set C of faulty processes results in the creation of orphans. Quantifying (2) over all C, we obtain:

$$\forall m: (Depend(m) \subseteq Log(m)) \quad (3)$$

Since we want (3) to hold for every state, we require the following property:

$$\forall m: \Box(Depend(m) \subseteq Log(m)) \quad (4)$$

Property (4) is a safety property: it must hold in every state of every execution to ensure that no orphans are created. It states that, to avoid orphans, it is sufficient to guarantee that if the state of a process p depends on the delivery of message m, then p keeps a copy of m's determinant in its volatile memory.

We say that $\#m$ is *stable* (denoted *stable*(m)) when $\#m$ cannot be lost. Condition 3 must hold only for messages with a determinant that is not stable. Hence, 3 need only hold when $\#m$ *is* not stable:

$$\forall m: \Box(\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m))) \quad (5)$$

We call this property the *always no orphans condition*. If determinants are kept in stable storage [9], then *stable*(m)

2. A similar argument can be made about *m.dest* not being essential in the determinant of m. We keep *m.dest* in $\#m$ for ease of exposition.

3. 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 [10], 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 it can tolerate at most two process failures at a time, we do not consider further this more general method of logging.

holds when the write of #m to stable memory completes. If determinants are kept in volatile memory, and we assume that no more than f processes—where $|C| \leq f$—can fail concurrently, then $stable(m)$ holds as long as $f + 1$ processes have a copy of #m in their volatile memory. In the latter case, the *always no orphans condition* can be written:

$$\forall m: \Box((|Log(m)| \leq f) \Rightarrow (Depend(m) \subseteq Log(m))) \quad (6)$$

4.1 Pessimistic and Optimistic Protocols Revisited

As we saw in Section 3, pessimistic message-logging protocols do not create orphans. Therefore, they must implement Property (5). Indeed, they implement the following stronger property:

$$\forall m: \Box(\neg stable(m) \Rightarrow (|Depend(m)| \leq 1)) \quad (7)$$

In practice, (7) does not allow the process that delivers message m to send any messages until the determinant of m is stable.[4] To see this, suppose process p_1 has just delivered m. $Depend(m)$ contains only p_1, so $|Depend(m)| = 1$. Suppose now that p_1 sends a message m' to process p_2. Since $deliver_{p1}(m) \rightarrow deliver_{p2}(m')$, when process p_2 delivers m' it becomes a member of $Depend(m)$, and $|Depend(m)| = 2$. Thus, as soon as m' is sent, the consequence in (7) can become false. To preserve (7), p_1 ensures that $stable(m)$ holds before sending m'. If the time during which $\neg stable(m)$ is brief, then it is unlikely that p will attempt to send a message while #m is not stable, so any performance loss by inhibiting message sends is (hopefully) small.

The reasoning behind optimistic protocols starts from the same assumption used by pessimistic protocols: The time during which $\neg stable(m)$ holds is brief. Hence, (5) holds trivially nearly all the time. Therefore, it is not practical to incur the performance cost that pessimistic protocols suffer. Instead, optimistic protocols take appropriate actions during recovery to re-establish (5) in the unlikely event that it is violated as a result of the failure of a set of processes C.

Optimistic protocols implement the following property:

$$\forall m: \Box(\neg stable(m) \Rightarrow ((Log(m) \subseteq C) \Rightarrow \Diamond(Depend(m) \subseteq C))) \quad (8)$$

Property (8) is weaker than Property (2), and therefore weaker than Property (5). Property (5) permits the temporary creation of orphan processes, but guarantees that, by the time recovery is complete, no surviving process will be an orphan and Property (2) will hold. This is achieved during recovery by rolling back orphan processes until their states do not depend on any message whose determinant has been lost. In other words, $Depend(m)$ is made smaller (or, equivalently, C is made larger) until $Depend(m) \subseteq C$ and Property (2) is restored.

4.2 Causal Message-Logging Protocols

Pessimistic protocols implement a property stronger than Property (4) and, therefore, never create orphans. However, implementing this stronger property may introduce blocking in failure-free runs. Optimistic protocols avoid this blocking by satisfying a property weaker than Property (4). It is natural to ask whether protocol can be designed that implements Property (4)—and therefore creates no orphans—yet does not implement a property as strong as Property (7)—and thus does not introduce any blocking. We find such a property by strengthening Property (6).

Consider the following property:

$$\forall m: \Box((|Log(m)| \leq f)\ (Depend(m) = Log(m))) \quad (9)$$

Property (9) strengthens (6), and so protocols that implements (9) prevents orphans. Furthermore, such protocols disseminate the least number of copies of #m needed in order to satisfy (6), thereby conserving storage and network bandwidth.

Unfortunately, satisfying Property (9) requires processes to be added to $Log(m)$ and $Depend(m)$ simultaneously. Satisfying this requirement would result in complicated protocols. Thus, we consider a different strengthening of Property (6) that is weaker than Property (9) but still bounds $Log(m)$:

$$\forall m: \Box((|Log(m)| \leq f) \Rightarrow ((Depend(m) \subseteq Log(m)) \land \Diamond(Depend(m) = Log(m)))) \quad (10)$$

This characterization strongly couples logging with causal dependency on deliver events. It requires that:

- As long as #m is at risk of being lost, all processes that have delivered a message sent causally after the delivery of m have also stored a copy of #m.
- Eventually, all the processes that have stored a copy of #m deliver a message sent causally after the delivery of m.

We call the protocols that implement Property (10) *causal* message-logging protocols. Family-Based-Logging [1] and Manetho [7] are two examples of causal message-logging protocols for the special cases $f = 1$ and $f = n$, respectively.

5 Optimal Message-Logging Protocols

We now consider four metrics with which one can compare different message-logging protocols. These metrics are not the only ones that one might wish to consider when making such a comparison, but they do give some measure of the running time of the protocol, both in the failure-free case and during recovery.

To express these four metrics, let Π be a protocol executed by a set $\mathcal{N}$ of processes, and assume that Π is written to tolerate no process failures and no transient channel failures. Let Π_μ denote Π combined with a message-logging protocol μ that tolerates process failures and transient channel failures.

1) *Number of forced roll-backs*. This metric gives some measure of the amount of work that might need to be discarded as a result of a process crash. Consider a run ρ of Π_μ, and suppose that a set $C \subseteq \mathcal{N}$ of processes fails. We say that μ forces r roll-backs if there is a run in which μ requires r correct processes to roll back their state, and for all ρ, μ requires no more than r correct processes to roll back their state. For example, optimistic sender-based logging [11] forces $|\mathcal{N}|$ rollbacks.

4. In pessimistic sender-based logging [11], 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 $Log(m)$.

A lower bound for r is 0. The existence of pessimistic protocols, where no correct process ever rolls back, establishes that this bound is tight.

2) *k-blocking*. This metric gives some measure of the amount of idle time that can be added to the execution of a process during a failure-free run. Consider a message m delivered by process *m.dest*, and let e be the first send event of process *m.des* 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 [11] 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 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 [4].

A lower bound for k is 0. The existence of optimistic protocols establishes that this bound is tight.

3) *Number of messages*. This metric gives one measure of the load on the network generated by the message logging protocol. Protocol Π can be transformed into a protocol Π_ℓ that tolerates transient channel failures using an acknowledgment scheme: Whenever process *m.dest* receives a sequence of messages generated by Π, it sends an acknowledgment to process *m.source*. Process *m.source* resends a message until it receives such an acknowledgment. In all runs, including failure-free ones, protocol Π_ℓ will send more messages than Π.

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 [11] potentially requires that one extra acknowledgment be sent for each application message sent.

A lower bound is to send no additional messages. The existence of optimistic protocols establishes that this bound is tight.

4) *Size of messages*. This metric gives another measure of the load on the network generated by the message logging protocol. Consider any message m sent in a run of Π_ℓ, and let m_μ be the corresponding message sent in the equivalent failure-free run of Π_μ. Let $|m|$ and $|m_\mu|$ be the size of 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 corresponding runs. For example, for optimistic protocols that track direct dependencies, a is a constant [20], [22]; while for optimistic protocols that track transitive dependencies a is proportional to the number of processes in the system [21], [20].

A lower bound for a is 0. The existence of pessimistic receiver-based logging proves that this bound is tight.

There is a tradeoff between the number of additional messages sent in Π_μ and the sizes of these messages: A message-logging protocol μ could include less information in a message by sending additional messages containing the extra data. In the majority of deployed computer networks there is, to a point, a performance benefit in sending a few large messages instead of several small messages. Hence, we prefer to keep the number of additional messages small at the expense of message size, and 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. It is a trivial exercise to change protocols that are optimal in this sense to protocols that minimize the size of messages by sending additional messages.

Of course, protocols that are optimal according to our definition do not necessarily outperform in practice nonoptimal protocols. 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 [8]. Nevertheless, the protocols we call optimal are unique in optimally addressing the theoretical *desiderata* of the message-logging approach.

6 Using Causal Delivery Order to Enforce the Always-No-Orphans Consistency Condition

We now derive an optimal causal message-logging protocol, i.e., an optimal message-logging protocol that implements Property (10). We do so by first presenting, in this section, a simple nonoptimal protocol that uses causal delivery order to implement the always-no-orphans consistency condition. In the following section, we refine this protocol and obtain an optimal causal message-logging protocol. We begin by defining causal delivery order.

6.1 Causal Delivery Order

Let $send_p(m)$ to q denote the event whereby process p sends message m to process q, and $receive_q(m)$ the event whereby q receives m.

FIFO delivery order guarantees that if a process sends a message m followed by a message m' to the same destination process, then the destination process does not deliver m' unless it has previously delivered m. Formally:

$$\forall p, r, m, m' : send_p(m) \text{ to } r \rightarrow send_p(m') \text{ to } r \Rightarrow deliver_r(m) \rightarrow deliver_r(m') \quad (11)$$

FIFO delivery order constrains the order in which the destination process delivers messages m and m' only when m causally precedes m' and m and m' are sent by the same process. Causal delivery order [6] strengthens FIFO delivery order by removing the requirement that ordering occurs only when the source of m and m' are the same. It guarantees that if the sending of m causally precedes the sending of m' and m and m' are directed to the same destination process, then the destination process does not deliver m' unless it has previously delivered m. Formally:

$$\forall p, q, r, m, m' : send_p(m) \text{ to } r \rightarrow send_p(m') \text{ to } r \Rightarrow deliver_r(m) \rightarrow deliver_r(m') \quad (12)$$

An example of an execution where process r delivers messages according to causal delivery order is shown in Fig. 1. Since $send_p(m_3)$ to r causally follows $send_q(m_1)$ r, r delivers m_3 only after m_1 has been delivered.

There are two fundamental approaches to implementing causal message delivery. The first is to add to each message m additional information that m's destination process uses to determine when m can be delivered. [16], [19], [6]. Using this approach, process r in Fig. 1 would realize, when it receives m_3, that it must wait to receive m_1 and would delay delivery of m_3 accordingly. The problem with this approach is that slow messages can significantly affect the performance of the system, since they prevent faster messages from being delivered.

The second approach is for each process p to piggyback, on each message m that p sends, all messages m' sent in the causal history of the event $send_p(m)$ such that p does not know if m' has already been delivered [5]. The piggybacked messages are placed in a total order that extends the partial order imposed by the happens-before relation. Before delivering m, process $m.dest$ first checks if any message m' in m's piggyback has $m.dest$ for destination. If so, p delivers m only after each such message m' has been delivered according to causal delivery order. This is the approach illustrated in Fig. 2. Message m_1 is piggybacked on m_2 and m_3, since p and q do not know whether m_1 has already been delivered. When r receives m_3, it checks the piggyback to find m_1. Process r, therefore, immediately delivers m_1 and then delivers m_3 without waiting for the slower copy of message m_1 to arrive. When the copy of m_1 is received directly from q, r discards it.

The problem with this piggybacking approach is that, even though several optimizations can be used to reduce the size of the piggyback, the overhead on message size can become very large.

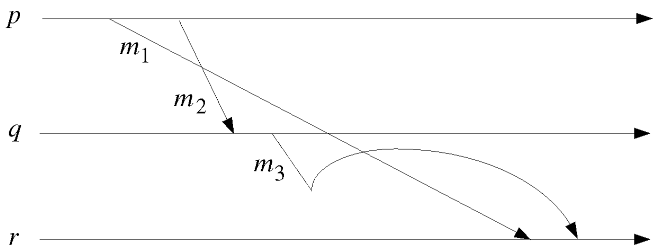

Fig. 1. An example of causal delivery order.

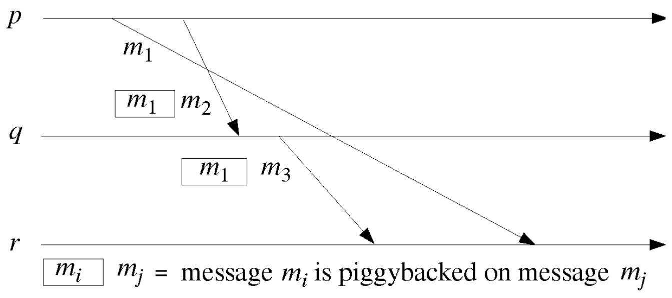
p
q
r
m_1
$\boxed{m_1}$ m_2
$\boxed{m_1}$ m_3
$\boxed{m_i}$ m_j = message m_i is piggybacked on message m_j

Fig. 2. Implementing causal delivery order through piggybacking.

6.2 A Suboptimal Causal Protocol

We now present a Protocol Π_{cd} that uses causal delivery order to satisfy Property (4). Processes in Π_{cd} behave as follows:

1) Processes exchange two kinds of messages: *application messages* and *determinant messages*. A determinant message contains the determinant of an application message.
2) Application and determinant messages are delivered according to causal delivery order.
3) Suppose process p receives an application message m. To deliver m, p creates the determinant $\#m$ of message m and logs $\#m$ in its volatile memory. Then, p supplies $m.data$ to the application.
4) Suppose process p delivers an application message m. Before sending any subsequent application messages, p sends a determinant message containing $\#m$ to all the other processes.
5) Suppose process p receives a determinant message containing $\#m$. To deliver the determinant message, p logs $\#m$ in its local volatile memory.

THEOREM 1. *Protocol Π_{cd} satisfies Property* (4).

PROOF. Property (4) requires that:

$$\forall m: ((Depend(m) \subseteq Log(m))$$

hold throughout execution. According to the definition of $Depend(m)$ given in Section 4, a process p is a member of $Depend(m)$ for an application message m if one of the following two cases holds:

Case 1. p is the destination of m.

Case 2. p is the destination of an application message m', and $deliver_{m.dest}(m)$ is in the causal history of $deliver_p(m')$.

We now show that in both cases protocol Π_{cd} guarantees that if $p \in Depend(m)$ then $p \in Log(m)$, so $Depend(m) \subseteq Log(m)$ holds.

Case 1. By point 3) of Π_{cd}, if p is the destination of m, then p will log $\#m$ in its volatile storage before delivering m. Hence, $Depend(m) \subseteq Log(m)$ holds.

Case 2. If p is not the destination of m, then some other process $q \neq p$ delivered m, and p delivered a message m' such that $deliver_q(m) \rightarrow deliver_p(m')$. Furthermore, there must exist an application message m'', not necessarily distinct from m', such that $deliver_q(m) \rightarrow send_q(m'') \rightarrow deliver_p(m')$ (see Fig. 3). By point 4) of Π_{cd}, q must have sent a determinant message containing $\#m$ to all processes—including process p—before sending m''. By Point 2) of Π_{cd}, all messages are delivered according to causal delivery: Therefore, p must have delivered the message containing $\#m$ before delivering m''. It follows from Point 5) of Π_{cd} that p logged $\#m$ in its volatile storage before delivering m''. Hence, $p \in Log(m)$ holds. □

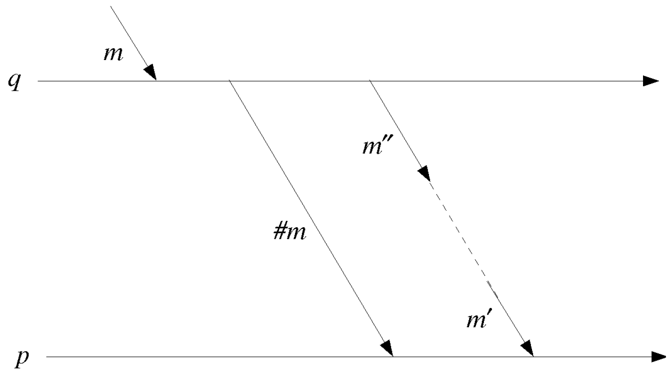

Fig. 3 . $Depend(m) \subseteq Log(m)$ in Π_{cd}.

Fig. 4 shows an execution of Protocol Π_{cd}. Process p_1 sends application message m_1 to process p_3, and then sends application message m_2 to process p_2. After delivering m_2, and before sending application message m_3, p_2 sends to all processes a determinant message containing $\#m_2$. Note that process p_3 first receives m_3, then $\#m_2$, and finally m_1. However, in order to respect causal delivery order, p_3 actually delivers first m_1, then $\#m_2$, and finally m_3.

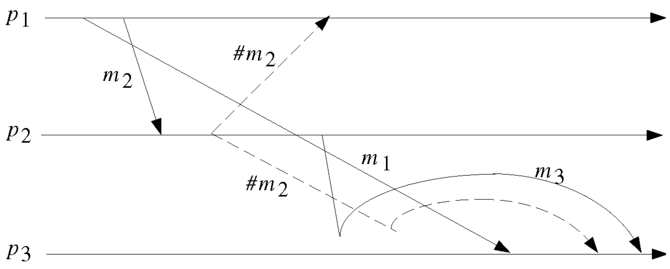

Fig. 4. An execution of Π_{cd}.

6.3 An Optimal Causal Protocol

Protocol Π_{cd} has several limitations. First, it is not optimal, since for each deliver event of an application message m, it uses additional determinant messages to send $\#m$ to all processes. Second, it does not satisfy Property (10), since a process can enter $Log(m)$ by receiving a copy of $\#m$ without ever becoming a member of $Depend(m)$. Finally, it forces both application and determinant messages to be delivered according to causal delivery order, even though causal delivery order is necessary only to regulate how determinant messages are delivered with respect to application messages.

However, by choosing an appropriate implementation of causal delivery order, protocol Π_{cd} can serve as the starting point for a more efficient protocol. Since we desire a nonblocking protocol, our first step is to choose an implementation of causal delivery based on the piggybacking scheme described in Section 6.1. Fig. 5 shows the effects of using the piggybacking scheme on the execution in Fig. 4.

Observe that, since for each application message m the determinant $\#m$ is piggybacked on any application message sent causally after event $deliver_p(m)$, there is no need to explicitly send $\#m$ in a separate determinant message. Instead, we require each process, before delivering a message m, to deliver all the determinants piggybacked on m. The resulting protocol successfully addresses the first two limitations of the original protocol. By piggybacking $\#m$ on application messages, the protocol ensures that no correct processes p will enter $Log(m)$ unless p will eventually join $Depend(m)$. Furthermore, the protocol is optimal in the number of messages it sends because the determinants are piggybacked on existing application messages. Fig. 6 shows the effect of the new protocol on the execution in Fig. 5.

Note that, once the piggybacking of the determinants is in place, we are guaranteed that no process will deliver an application message m unless it first delivers the determinants of all the messages delivered in the causal history of $send_{m.source}(m)$. This is precisely the property that we need to guarantee that $\forall m$: $Depend(m) \subseteq Log(m)$ holds. Therefore, provided that we piggyback the determinants, we can safely relax the requirement that application messages be delivered using causal delivery order. Fig. 7 shows the effects of the new protocol on the execution in Fig. 6.

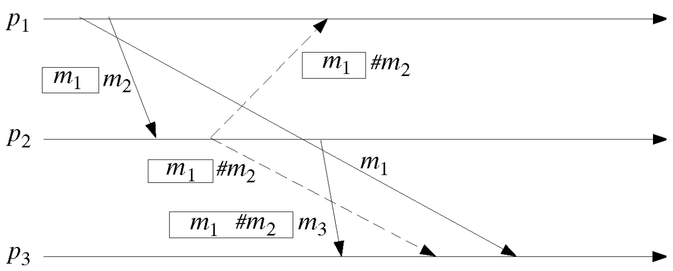

Fig. 5. Implementing causal delivery order through piggybacking.

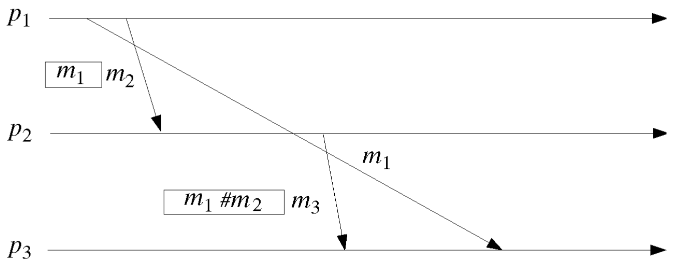

Fig. 6. Determinants are piggybacked on application messages.

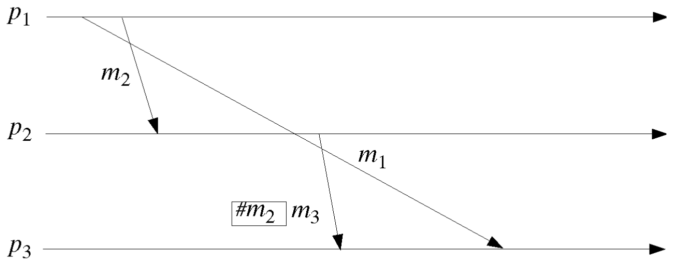

Fig. 7. An execution of Π_{oc}.

The last step of the derivation is to adapt the above protocol to the case where we assume that no more than f processes fail concurrently. In this case, a process piggybacks $\#m$ on an application message only as long as $\#m$ is not stable—i.e., as long as $|Log(m)| \leq f$. We call the resulting protocol Π_{oc}. Processes in Π_{oc} behave as follows:

1) Processes exchange only application messages. However, determinants may be piggybacked on application messages.
2) Suppose process p sends a message m to process q. p piggybacks on m all the nonstable determinants $\#m'$ such that p has $\#m'$ its volatile memory.
3) Suppose process p receives a message m. Before delivering m, process p first logs all the determinants piggybacked on m in its determinant log, which is implemented in p's volatile memory. Then, p creates the determinant for message m, and logs it in the determinant log. Finally, p delivers m by supplying $m.data$ to the application.

THEOREM 2. *Protocol* Π_{oc} *satisfies Property* (10). *Furthermore, Protocol* Π_{oc} *is optimal.*

PROOF. We first prove that Π_{oc} satisfies Property (10). In particular, we prove that:

1) $(|Log(m)| \le f) \Rightarrow (Depend(m) \subseteq Log(m))$
2) $(|Log(m)| \le f) \Rightarrow \Diamond\, (Depend(m) = Log(m))$

are satisfied by executions of Π_{oc}. We first observe that, if $|Log(m)| > f$ then 1) and 2) are trivially true. Therefore, we consider the case in which $|Log(m)| \le f$.

We prove 1) by showing that, whenever $|Log(m)| \le f$ holds, if a process p is a member of $Depend(m)$, then p is also a member of $Log(m)$. Process p is a member of $Depend(m)$ if either p is the destination of m or p is the destination of a message m', and $deliver_{m.dest}(m) \rightarrow deliver_p(m')$.

In the first case, by Point 3) of Π_{oc}, p has saved $\#m$ in its determinant log. Therefore, if p is a member of $Depend(m)$, then p is a member of $Log(m)$.

In the second case, we proceed by induction on the length ℓ of the causal chain of processes associated with a causal path that starts with event $deliver_{m.dest}(m)$ and ends with event $deliver_p(m')$.

Base Case. $\ell = 1$.

If $\ell = 1$, then process $m.dest$ was the sender of message m'. In particular, it must be the case that:

$$deliver_{m.dest}(m) \rightarrow send_{m.dest}(m') \text{ to } p \rightarrow deliver_p(m').$$

By Point 2) of Π_{oc}, process $m.dest$ has piggybacked on m' all the determinants that were not stable at the time of event $send_{m.dest}(m')$ to p. In particular, if $\#m$ was not stable, then it was piggybacked on m'. Because of Point 3) of Π_{oc}, if $\#m$ was piggybacked on m', then p has saved $\#m$ in its determinant log before delivering m'. Therefore, if p is a member of $Depend(m)$, and $|Log(m)| \le f$, then p is also a member of $Log(m)$.

Inductive Step. We assume that 1) holds for all causal paths whose associated causal chain of processes has length $\ell \le c$. We prove that 1) holds for any causal path whose associated causal chain of processes has length $\ell = c + 1$.

Consider a causal path that starts with event $deliver_{m.dest}(m)$ and ends with event $deliver_p(m')$, and assume that the associated causal chain of processes has length $\ell = c + 1$. Let q be the process that sent message m' to p. Since, by assumption, a process never sends a message to itself, it follows that $p \ne q$. Therefore, the causal chain associated with the causal path that starts with $deliver_{m.dest}(m)$ and ends with $send_q(m')$ to p is of length c. By the inductive hypothesis, if $|Log(m)| \le f$ when m' was sent, then $\#m$ was in q's determinant log. By Point 2) of Π_{oc}, q piggybacked $\#m$ on m'. By point 3)) of Π_{oc}, when p received m', p added $\#m$ to its determinant log before delivering m. Therefore, if p is a member of $Depend(m)$, and $|Log(m)| \le f$, then p is also a member of $Log(m)$.

We now prove Part 2), assuming $|Log(m)| \le f$.

If process p is a member of $Log(m)$, then p stores a copy of $\#m$ in its determinant log. For this to happen, either p received m, or it received a message m', onto which $\#m$ was piggybacked, such that $deliver_{m.dest}(m) \rightarrow Send_{m'.source}(m')$. In the first case, unless p fails, it will eventually deliver m and become a member of $Depend(m)$. The second case is similar: Unless p fails, it will eventually deliver m' and become a member of $Depend(m)$. Furthermore, since there is only a finite number of processes in $\mathcal{N}$ and $Log(m) \subseteq \mathcal{N}$, eventually all nonfaulty members of $Log(m)$ will join $Depend(m)$ and 2) will hold. Note that, if p fails, then p will no longer be a member of $Depend(m)$. However, in this case, p will lose all the data in its determinant log, which is implemented in volatile memory, and will therefore leave $Log(m)$. Hence, 2) holds whether p is a correct process or not. This concludes the proof that Π_{oc} satisfies Property (10).

We now have only to show that Π_{oc} is optimal. To do so, we observe that no additional messages are generated by protocol Π_{oc} over the ones needed by the application, because determinants are piggybacked on existing application messages. Furthermore, no correct processes are forced to rollback as a result of a process' failure, because Π_{oc} is a causal protocol. Finally, Π_{oc} is 0-blocking, because none of 1), 2), and 3) imply blocking. □

6.4 Implementation Issues

Protocol Π_{oc}'s piggybacking scheme guarantees that all the determinants needed to recover the system to a consistent global state from up to f concurrent failures will be available during recovery. However, the scheme leaves several open questions. First, Π_{oc} does not specify how to collect and use the logged determinants to perform recovery. Furthermore, Π_{oc} assumes that processes have knowledge of the current value of $|Log(m)|$ when they determine whether or not to piggyback the determinant $\#m$ on an application message. This assumption is not realistic in an asynchronous distributed system, since—as we have observed before—it requires processes to have instantaneous access to $Log(m)$, which is defined over the entire distributed system. Since the focus of this paper is not the implementation of optimal causal message-logging protocols but rather their specification, we do not address these issues here. The interested reader is referred to [3], in which we discuss the implementation of a class of optimal causal message-logging 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

In this paper, we have shown that a small amount of simple formalism can go a long way. We have presented the first specification of the message logging approach to fault-tolerance, by precisely defining the *always-no-orphans* condition, 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 [3], [2].

APPENDIX A—DERIVATIONS

In this section, we show the derivations of the equations in this paper that may not be immediately obvious.

A.1 Derivation of (2) from (1)

$\forall p : \neg(p \; \textit{orphan of} \; C)$

$= \langle\langle \textit{Definition of p orphan } \text{of } C \rangle\rangle$

$$\forall p : \neg\begin{pmatrix} (p \in \mathcal{N} - C \wedge \\ \exists m : ((p \in \textit{Depend}(m)) \wedge (\textit{Log}(m) \subseteq C)) \end{pmatrix}$$

$= \langle\langle \textit{De Morgan's Law} \rangle\rangle$

$$\forall p : \begin{pmatrix} (p \notin \mathcal{N} - C) \vee \\ (\forall m : (\neg(p \in \textit{Depend}(m)) \vee \neg(\textit{Log}(m) \subseteq C))) \end{pmatrix}$$

$= \langle\langle \text{For } x \text{ not free in Q: } (\forall x : P(x)) \vee Q = \forall x : (P(x) \vee Q) \rangle\rangle$

$\forall p, m : ((p \in C) \vee \neg(p \in \textit{Depend}(m)) \vee \neg(\textit{Log}(m) \subseteq C))$

$= \langle\langle \text{Definition of} \Rightarrow \rangle\rangle$

$\forall p, m : ((\textit{Log}(m) \subseteq C) \Rightarrow ((p \in \textit{Depend}(m)) \Rightarrow (p \in C)))$

$= \langle\langle P \subseteq Q = \forall p : (p \in \text{P} \Rightarrow p \in \text{Q}) \rangle\rangle$

$\forall m : ((\textit{Log}(m) \subseteq C) \Rightarrow (\textit{Depend}(m) \subseteq C))$

A.2 Derivation Showing that (7) Implies (5)

1) $\forall m : \square(\neg \textit{stable}(m) \Rightarrow (\,|\textit{Depend}(m)|\, \leq 1))$
   $\langle\langle \text{Definition of } \textit{Depend}(m) \rangle\rangle$
2) $\forall m : \square((\,|\textit{Depend}(m) \leq 1)\,| \leq (\textit{Depend}(m) \subseteq \{\textit{m.dest}\}))$
   $\langle\langle \text{Definition of } \textit{Log}(m) \rangle\rangle$
3) $\forall m : (\{\textit{m.dest}\} \subseteq \textit{Log}(m))$
   $\langle\langle \text{Transitivity of Set Inclusion, using 2) and 3)} \rangle\rangle$
4) $\forall m : \square(\,|\textit{Depend}(m)|\, \leq 1) \Rightarrow (\textit{Depend}(m) \subseteq \textit{Log}(m))$
   $\langle\langle \text{Transitivity of Implication, using 1)and 4)} \rangle\rangle$
5) $\forall m : \square(\neg \textit{stable}(m) \Rightarrow (\textit{Depend}(m) \subseteq \textit{Log}(m)))$

APPENDIX B—SHARING THE LOG

The specification derived in Section 4 assumes that a process logs determinants either in its own volatile memory or to stable memory. Hence, determinants may be lost only when a process fails, and since processes fail independently, different copies of the same determinant are lost independently. A more general approach is to regard determinant logs as being first-class objects. By doing so, one can model a set of processes sharing storage for logging purposes and thereby decouple the failure of processes from the loss of determinants. For example, one reasonable approach would be to implement the logging component of a message-logging protocol so that a single, shared log is maintained for all the processes that run on the same processor. This would result in a more efficient implementation of causal message logging.

Define a *logging site* to be a storage object that a process can read and write. Each process uses a logging site for its determinant log. We assume that logging sites fail and recover. When a logging site fails, the values written to the logging site are lost, and all processes using that logging site also fail. A process may fail without its associated logging site failing, however.

When determinants are kept at logging sites, $\textit{Log}(m)$ denotes the set of logging sites that contain the determinant of event $\textit{deliver}_{m.dest}(m)$. An implementation of stable storage using stable memory is represented by a single logging site that never fails.

Let $\mathcal{L}$ denote the set of logging sites. Function $L(p)$ denotes the logging site used by process p. Assume that $L(p)$ is a constant function—each process uses a single logging site that does not change. $P(\ell)$, for $\ell \in \mathcal{L}$, denotes the processes that are associated with logging site ℓ. Define functions $L^{\cup}(\mathcal{P})$ and $P^{\cup}(S)$ to be the set-valued domain versions of $L(p)$ and $P(\ell)$, respectively:

$$L^{\cup}(\mathcal{P}) \overset{\text{def}}{=} \bigcup_{p \in \mathcal{P}} L(p) \tag{13}$$

$$P^{\cup}(S) \overset{\text{def}}{=} \bigcup_{\ell \in S} P(\ell) \tag{14}$$

We now repeat, under these new assumptions, the derivation that in Section 4 led us to Property (6), which enforces the always-no-orphans consistency condition.

A process is an orphan process when its state depends on a determinant that is not logged. Let C denote a set of failed processes and $\mathcal{E}$ denote a set of failed logging sites. Then:

$$p \textit{ orphan of } C, \mathcal{E} \overset{\text{def}}{=} \begin{pmatrix} (p \in \mathcal{N} - C) \wedge \\ \exists m : ((p \in Depend(m)) \wedge (Log(m) \subseteq \mathcal{E})) \end{pmatrix} \quad (15)$$

A logging site must fail for a process to become an orphan. Negating (15) and quantifying over all p gives the following:

$$\forall m : ((Log(m) \subseteq \mathcal{E}) \Rightarrow (Depend(m) \subseteq C)) \quad (16)$$

Property (16) is necessary and sufficient to guarantee that the failure of C processes and of $\mathcal{E}$ logging sites creates no orphans. From definitions (13) and (14), it follows that:

$$\mathcal{A} \subseteq \mathcal{B} \Rightarrow P^{\cup}(\mathcal{A}) \subseteq \mathrm{P}^{\cup}(\mathcal{B}) \quad (17)$$

$$\mathcal{A} \subseteq \mathcal{B} \Rightarrow L^{\cup}(\mathcal{A}) \subseteq L^{\cup}(\mathcal{B}) \quad (18)$$

where $\mathcal{A}$ and $\mathcal{B}$ are sets of the appropriate type. Noting that $L^{\cup}(P^{\cup}(S) = S$, it is straightforward to show that $P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B}) \Rightarrow \mathcal{A} \subseteq \mathcal{B}$:

⟨⟨Hypothesis assumed⟩⟩

1) $P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B})$

⟨⟨Modus ponens, 1 and (18)⟩⟩

2) $L^{\cup}(P^{\cup}(\mathcal{A})) \subseteq L^{\cup}(P^{\cup}(\mathcal{B}))$

⟨⟨$L^{\cup}(P^{\cup}(S)) = S$ and 2⟩⟩

3) $\mathcal{A} \subseteq \mathcal{B}$ □

Hence, $\mathcal{A} \subseteq \mathcal{B} \equiv P^{\cup}(\mathcal{A}) \Rightarrow P^{\cup}(\mathcal{B})$. We can replace the antecedent of (16) with the equivalent expression $P^{\cup}(Log(m)) \subseteq P^{\cup}(\mathcal{E})$:

$$\forall m : ((P^{\cup}(Log(m)) \subseteq P^{\cup}(\mathcal{E}) \Rightarrow (Depend(m) \subseteq C)) \quad (19)$$

By assumption, $\mathcal{E}$ and C are constrained: $P^{\cup}(\mathcal{E}) \subseteq C$. Hence, the following strengthens (19):

$$\forall m : ((P^{\cup}(Log(m)) \subseteq C) \Rightarrow (Depend(m) \subseteq C)) \quad (20)$$

Universally quantifying (20) over C, we obtain:

$$\forall m : (Depend(m) \subseteq P^{\cup}((Log(m))) \quad (21)$$

Since we want (21) to hold in every state, we obtain the following property:

$$\forall m : \Box(Depend(m) \subseteq P^{\cup}(Log(m))) \quad (22)$$

Property (22), like Property (4) of Section 4, is a safety property. If satisfied, (4) guarantees that no orphans will be created during a run.

If we assume that no more than $f_{\mathcal{E}}$ logging sites can fail at any time, then #m is stable once it has been logged at more than $f_{\mathcal{E}}$ logging sites. Hence, Property (22) need hold only as long as failures cannot cause #m to be lost:

$$\forall m : \Box((\,|Log(m)|\, \leq f_{\mathcal{E}}) \Rightarrow Depend(m) \subseteq P^{\cup}(Log(m))) \quad (23)$$

Finally, we can strengthen Property (23) as in Section 4.2, to obtain the following property, which characterizes causal message logging when there are shared logging sites:

$$\forall m : \Box((\,|Log(m)|\, \leq f_{\mathcal{E}}) \Rightarrow \wedge\, Depend(m) \subseteq P^{\cup}(Log(m))) \wedge \Diamond(Depend(m) = P^{\cup}((Log(m)))) \quad (24)$$

Using a derivation similar to the one given in Section 6.3, it is straightforward to develop an optimal protocol Π_{ol} that satisfies Property 24. It may be surprising that at this level of refinement, protocol Π_{ol} is essentially identical to protocol Π_{oc} of Section 6.3. The main differences in the protoco$\mathcal{L}$ is found through further refinement. The most significant difference is in the size of the data structures they to compute $Log(m)$: for Π_{oc}, the data structure is size $O(n^2)$ while for Π_{ol}, the data structure is size $O(\,|\mathcal{L}|^2)$ where $|\mathcal{L}|$ is the number of logging sites. Details can be found in [2].

ACKNOWLEDGMENTS

We thank Bruce Hoppe, Fred Schneider, and Sam Toueg for many discussions on this research and on early versions of this paper. We also thank the anonymous referees for their helpful suggestions.

Keith Marzullo's work was supported by the DoD Defense Advanced Research Projects Agency (DARPA) under NASA Ames Grant No. NAG 2–593, Contract N00140-87-C-8904. Marzullo was supported by the Air Force Office of Scientific Research (AFOSR) Grant No. 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.

Lorenzo Alvisi's research is based on work supported, in part, by the Office of Naval Research (ONR) under Contract No. N00014-91-J-1219; the National Science Foundation (NSF) under Grant No. CCR-9003440; DARPA/NSF under Grant No. CCR-9014363; NASA/DARPA under Grant No. NAG-2-893; and the AFOSR under Contract No. 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.

REFERENCES

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

[2] L. Alvisi, “Understanding the Message Logging Paradigm for Masking Process Crashes,” PhD thesis, Cornell Univ., Dept. of Computer Science, Jan. 1996. Available as Technical Report TR-96-1577.

[3] L. Alvisi and K. Marzullo, “Tradeoffs in Implementing Optimal Message Logging Protocol,” *Proc. 15th Symp. Principles of Distributed Computing*, pp. 58-67, ACM, June 1996.

[4] A. Borg, J. Baumbach, and S. Glazer, “A Message Systems Supporting Fault Tolerance,” *Proc. Symp. Operating Systems Principles*, pp. 90-99, ACM SIG OPS, Oct. 1983.

[5] K. Birman and T. Joseph, “Reliable Communication in the Presence of Failures,” *ACM Trans. Computer Systems*, vol. 5, no. 2, pp. 47-76, Feb. 1987.

[6] K. Birman, A Schiper, and P. Stephenson, “Lightweight Causal and Atomic Group Multicast,” *ACM Trans. Computer System*, vol. 9, no. 3, pp. 272-314, Aug. 1991.

[7] E.N. Elnozahy and W. Zwaenepoel, “Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit,” *IEEE Trans. Computers*, vol. 41, no. 5, pp. 526-531, May 1992.

[8] E.N. Elnozahy and W. Zwaenepoel, "On the Use and Implementaiton of Message Logging," *Digest of Papers: 24th Ann. Int'l Symp. Fault-Tolerant Computing*, pp. 298-307, IEEE Computer Society, June 1994.

[9] J.N. Gray, "Notes on Data Base Operating Systems," R. Bayer, R.M. Graham, and G. Seegmueller, eds., *Operating Systems: An Advanced Course*, pp. 393-481, *Lecture Notes in Computer Science 60*. Springer-Verlag, 1977.

[10] T.Y. Juang and S. Venkatesan, "Crash Recovery with Little Overhead," *Proc. 11th Int'l Conf. Distributed Computing Systems*, pp. 454-461, IEEE Computer Society, June 1987.

[11] D.B. Johnson and W. Zwaenepoel, "Sender-Based Message Logging," *Digest of Papers: 17th Ann. Int'l Symp. Fault-Tolerant Computing*, pp. 14-19, IEEE Computer Society, June 1987.

[12] D.B. Johnson and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," *J. Algorithms*, vol. 11, pp. 462-491, 1990.

[13] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[14] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th Ann. Symp. Foundations of Computer Science*, pp. 46-57, Nov. 1977.

[15] M.L. Powell and D.L. Presotto, "Publishing: A Reliable Broadcase Communication Mechanism," *Proc. Ninth Symp. Operating System Principles*, pp. 100-109. ACM SIGOPS, Oct. 1983.

[16] M. Raynal, A. Schiper, and S. Toueg, "The Causal Ordering Abstraction and a Simple Way to Implement It," *Information Processing Letters*, vol. 39, no. 6, pp. 343-350, 1991.

[17] R.E. Strom, D.F. Bacon, and S.A. Yemini, "Volatile Logging in n-Fault-Tolerant Distributed Systems," *Proc. 18th Ann. Int'l Symp. Fault-Tolerant Computing*, pp. 44-49, 1988.

[18] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Trans. Computer Systems*, vol. 2, no. 2, pp. 145-154, May 1984.

[19] A. Sandoz and A. Schiper, "A Characterization of Consisting Distributed Snapshops Using Causal Order," Technical Report TR92-14, Departement d'Informatique, Ecole Politechnique Fédérale de Lausanne, 1992.

[20] A.P. Sistla and J.L. Welch, "Efficient Distributed Recovery Using Message Logging," *Proc. 18th Symp. Principles of Distributed Computing*, pp. 223-238, ACM SIGACT/SIGOPS, Aug. 1989.

[21] R.B. Strom and S. Yemeni, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, vol. 3, no. 3, pp. 204-226, Apr. 1985.

[22] S. Venkatesan and T.Y. Juang, "Efficient Algorithms for Optimistic Crash Recovery," *Distributed Computing*," vol. 8, no. 2, pp. 105-114, June 1994.

Lorenzo Alvisi received his PhD degree from the Computer Science Department at Cornell University in 1996. He is currently an assistant professor in the Department of Computer Sciences at the University of Texas at Austin, where he cofounded the Laboratory for Experimental Software Systems (LESS). Dr. Alvisi's research interests include concurrency, distributed systems, and fault tolerance.

Keith Marzullo received his PhD degree from the Electrical Engineering Department at Stanford University in 1984. He has worked at Xerox, Palo Alto; has been on the faculty of Cornell University; and is currently an associate professor in the Department of Computer Science and Engineering at The University of California, San Diego. His research interests are in distributed fault-tolerant systems, both asynchronous and hard-real-time. Dr. Marzullo is an associate editor for *IEEE Transactions on Software Engineering*.