Trade-Offs in Implementing Causal Message Logging Protocols

Lorenzo Alvisi* Keith Marzullo†

Abstract

Casual message logging protocols [3] have several attractive properties: they introduce no blocking, send no additional messages over those sent by the application, and can never cause orphans to be created by crashes. Causal message logging, however, does require additional data to be piggybacked on application messages. The amount of such piggybacked data can become large.

In this paper, we present five different implementations of casual message logging. All of the corresponding protocols are parameterized by f, the maximum number of processes that can fail concurrently. We also explore how the application's communication structure can be exploited to limit the amount of piggybacked data.

1 Introduction

Message logging 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 state is inconsistent with the recovered state of a crashed process. Thus, in the terminology of message logging, message logging protocols must guarantee that there are no orphan processes, either through careful logging or through a somewhat complex recovery protocol.

The two main approaches to message logging are *optimistic* (for example, [18, 17, 11, 20]) and *pessimistic* (for example, [7, 15, 10, 19]). We have recently defined a third approach that we call *causal* [3]. There are two published causal message logging protocols: Family Based Logging (FBL) [4] and Manetho [9].

In the same paper we defined a message logging protocol to be *optimal* if it is causal and does not send any additional messages over those needed to mask transient link failures. Optimal message protocols do exact a price, however: they piggyback additional information on the application's messages.

One parameter of message logging protocols is the number of crash failures f that can occur before one of the processes successfully recovers. The two existing optimal message logging protocols are at opposite ends of the spectrum: FBL can tolerate only one crash at a time while Manetho can tolerate all processes crashing. On the other hand, FBL is a much simpler protocol than Manetho and on average it piggybacks significantly less information than Manetho does. If one can safely assume (by examining the architecture of the system) that the probability is extremely small that a second process will crash before a previously crashed process completes recovery, then the FBL strategy of piggybacking information would be a better choice than the Manetho strategy.

In this paper, we show how FBL can be extended to derive a continuum of optimal message logging protocols for values of f ranging from 1 up to the number

*Department of Computer Sciences, University of Texas at Austin. 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.

†Department of Computer Science and Engineering, University of California at San Diego. 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.

of processes n. Protocols with a lower value of f piggyback information to fewer processes than protocols with larger values of f, and so both the average size of messages and the amount of information logged in volatile memory will be less for smaller values of f.

We also present five different techniques for determining whether information should be piggybacked or not. These technique differ in their accuracy in determining whether or not a piece of information needs to be piggybacked or not. The more accurate techniques increase the size of some messages, but the additional size may be offset by piggybacking less information on average. On the other hand, we show that for some (very reasonable) applications, the least accurate technique is the most efficient even when $f = n$.

Due to lack of space, we do not present the protocol that is run when a crashed process recovers. All five protocols that we develop in this paper can use the same recovery protocol. A discussion on recovery as well as the actual recovery protocol can be found in [1].

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.

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 [12]. 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.

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. Given the states s_p and s_q of two processes p and q, $p \neq q$ respectively, 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 [8]; otherwise it is *inconsistent*.

We assume that processes are *piecewise deterministic* [19] in that the only nondeterminism in a process arises from the nondeterministic order in which messages are delivered. 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 [18]. 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 [16];
- 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 Specification

With the assumption that processes are piecewise deterministic, the only non-deterministic choices made during an execution 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.

For each message m delivered during a given run, 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 $\#m$.

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

$$
\begin{aligned}
Depend(m) \overset{\text{def}}{=} \{ j \in \mathcal{N} \mid \; & ((j = m.dest) \wedge j \text{ has delivered } m) \vee \\
& (\exists \text{ an event } e_j \text{ of non-crashed process } j: \\
& \quad (deliver_i(m) \rightarrow e_j))\}
\end{aligned}
$$

where $\rightarrow$ denotes the *happens-before* relationship [12]. Let $Log(m)$ denote the set of processes that maintain a copy of $\#m$ in their address space: in particular, process $m.dest$ is a member of $Log(m)$ once it delivers m. In [3], we showed that the following property ensures that no set of f or less crashed processes can lead to the creation of orphans:

$$\forall m : \Box(|Log(m)| \leq f \Rightarrow Depend(m) \subseteq Log(m)) \qquad (1)$$

($\Box$ is the temporal "always" operator) [14].

We say that the determinant of a delivery event $deliver_m(m.dest)$ is *stable* when $\#m$ cannot become lost due to process crashes, i.e. when $|Log(m)| > f$. Property 1 allows $Log(m)$ 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, we consider protocols that implement the following strengthening of Property 1:

$$\forall m : \Box \left(\begin{array}{c} (|Log(m)| \leq f) \Rightarrow \\ \left(\begin{array}{l} \wedge \quad (Depend(m) \subseteq Log(m)) \\ \wedge \quad \Diamond(Depend(m) = Log(m)) \end{array} \right) \end{array} \right) \qquad (2)$$

($\Diamond$ is the temporal "eventually" operator) [14]. This characterization strongly couples logging with causal dependency on deliver events. It requires that:

- All processes that delivered an application message sent causally after the delivery of m must have stored a copy of m's determinant.
- Eventually, the states of all the processes that have stored a copy of m's determinant will deliver an application message sent causally after the delivery of m.

We call the protocols that implement Property (2) *causal* message-logging protocols. In [3] we define *optimal* message logging protocols to be those protocols that (1) do not create orphans, (2) introduce no blocking, and (3) do not send any additional messages over those needed to mask transient link failures. Notice that the first two conditions require the protocol to be causal.

4 Family Based Logging

Family Based Logging (FBL) is a logging technique used by a class of optimal protocols that implement Property 2 as follows:

1. Before delivering m, process p logs $\#m$ in its volatile memory. By causality, when p sends a subsequent message m' to some process q, q will become a member of $Depend(m)$ when it delivers m'. Hence, if p has not determined that $|Log(m)| > f$ when it sends m', it piggybacks $\#m$ on m'. When it receives m', q logs $\#m$ and $\#m'$ in its volatile memory before delivering m'.
2. Suppose a process p receives a set of determinants piggybacked on a message m. Process p writes these determinants and the determinant $\#m$ to its log in volatile memory atomically with respect to another process q requesting those determinants (Process q would make such a request while executing the recovery procedure).
3. When p sends a subsequent message m' to another process r, p examines all determinants $\#m$ it has received through piggybacking. If p does not know that $|Log(m)| > f \vee r \in Log(m)$, then p piggybacks $\#m$ on m'.
4. During normal operation, a process does not send determinants except by piggybacking. Hence, a process does not receive a determinant of m unless it is to deliver a message sent causally after the delivery of m.

Conceptually, in this protocol each process p maintains a set $\mathcal{DS}_p$ that contains the determinants of all the delivery events reflected in p's state and that p does not know to be stable. This set is a subset of all of the determinants that p has logged in its volatile storage. Whenever p sends a message m' to some process q, process p piggybacks onto m' all the determinants in $\mathcal{DS}_p$ that p does not know that process q has seen. Hence, a fundamental issue of implementing FBL is how a process p determines $Log(m)$ for any determinant $\#m$ that p has received. In general, however, p may not know the exact values of $Log(m)$ and $|Log(m)|$, and so it must estimate these values. We denote p's estimated values for $Log(m)$ and $|Log(m)|$ as $Log(m)_p$ and $|Log(m)|_p$ respectively.

4.1 Estimating $Log(m)$ and $|Log(m)|$

In order for the protocol to satisfy Property 2, p must never overestimate $Log(m)$ or $|Log(m)|$. However, if p underestimates $|Log(m)|$, it may needlessly piggyback determinants that are already stable, making the messages on average significantly larger. Process p computes its estimates through additional information that is piggybacked to it by other processes. Thus, by exchanging more information, the processes can improve the accuracy of their estimates, and thereby avoid piggybacking useless data; yet, maintaining more accurate estimates requires the processes to piggyback more information which can in turn make the messages significantly larger.

The most basic piece of information about $|Log(m)|$ is gained when a process q delivers a message m. Once q delivers m, q knows that $q \in Log(m)$. Further pieces of information about $|Log(m)|$ are piggybacked on messages. Three natural pieces of information are:

$\#m$: When q receives $\#m$ from p, process q can safely infer that $Log(m)$ contains at least process p, process $m.dest$ (the original destination of message m) and process q itself.

$|Log(m)|_p$: Process p can send to q this piece of information either in addition to $\#m$, if $\#m \in \mathcal{DS}_p(q)$ or without $\#m$ if $\#m \notin \mathcal{DS}_p(q)$. Upon receipt of $|Log(m)|_p$, q can always safely infer that $|Log(m)|$ is no smaller than $|Log(m)|_p$. When q receives $\#m$ for the first time, q can further safely infer that $|Log(m)|$ must be at least equal to $|Log(m)|_p + 1$, since q itself could not be counted in $|Log(m)|_p$. Note that this scheme allows q to safely infer a value for $|Log(m)|$ without knowing the identity of the processes in $Log(m)$.

$Log(m)_p$: Process p can send to q this piece of information either in addition to $\#m$, if $\#m \in \mathcal{DS}_p(q)$ or without $\#m$ if $\#m \notin \mathcal{DS}_p(q)$. Upon receipt of $Log(m)_p$, process q can safely infer that $Log(m)_q$ must be at least equal to the union of the current set $Log(m)_q$ and $Log(m)_p$, and update $|Log(m)|_q$ accordingly. Using this scheme, when process p sends its estimate of $Log(m)$ to process q, it is providing q with the union of all the estimates relative to $Log(m)$ computed by the processes along the causal path that connects process $m.dest$ to process p.

Let $\mathcal{DS}_p(q)$ denote the subset of $\mathcal{DS}_p$ that contains the determinants $\#m$ such that $q \notin Log(m)_p$. One can define five protocols from these different information-exchange schemes.

1. Π_{Det}: Process p piggybacks only the determinants in $\mathcal{DS}_p(q)$.

2. $\Pi_{|Log|}$: For each determinant $\#m$ in $\mathcal{DS}_p(q)$, process p piggybacks both $\#m$ and $|Log(m)|_p$.

3. $\Pi^+_{|Log|}$: Process p piggybacks the determinants in $\mathcal{DS}_p(q)$. In addition, p piggybacks the value of $|Log(m)|_p$ for all messages m in its determinant log. Note that with $\Pi^+_{|Log|}$ process p piggybacks the value of $|Log(m)|_p$ even if m is not a member of $\mathcal{DS}_p(q)$ or of $\mathcal{DS}_p$.

4. Π_{Log}: For each determinant $\#m$ in $\mathcal{DS}_p(q)$, process p piggybacks both $\#m$ and $Log(m)$.

5. Π^+_{Log}: Process p piggybacks the determinants in $\mathcal{DS}_p(q)$. In addition, p piggybacks the set $Log(m)_p$ for all messages m in its determinant log. Note that with $\Pi^+_{|Log|}$ process p piggybacks the value of $Log(m)_p$ even if m is not a member of $\mathcal{DS}_p(q)$.

4.2 Comparison of the Protocols

The five protocols piggyback different amounts of information and estimate $Log(m)$ and $|Log(m)|$ differently. We examine these differences below.

4.2.1 Accuracy of $Log(m)_p$ and $|Log(m)|_p$

The execution shown in Figure 1 illustrates the differences between Π_{Det}, $\Pi_{|Log|}$ and Π_{Log} with respect to how accurately they estimate $Log(m)$ and $|Log(m)|$. For each deliver event executed by process p_i and for each of the three protocols, we show $Log(m)_{p_i}$ and $|Log(m)|_{p_i}$.

Through the receipt of message m_3, the three protocols yield the same estimates of $Log(m)$ and $|Log(m)|$. Once p_3 receives m_4, however, different estimates of $Log(m)$ and $|Log(m)|$ are computed by the three protocols:

Π_{Det}: Upon receipt of the copy of $\#m$ piggybacked on message m_4, process p_3 concludes that, in addition to itself, $Log(m)$ must include at least process $p_1 = m_4.source$ and process $p_2 = m.dest$. Process p_3 thus sets $Log(m)_{p_3} = \{p_1, p_2, p_3\}$, and $|Log(m)|_{p_3} = 3$.

$\Pi_{|Log|}$: As in the previous case, process p_3 sets $Log(m)_{p_3}$ to $\{p_1, p_2, p_3\}$. However, since this is the first time that p_3 receives $\#m$, p_3 was not in $Log(m)$ when p_1 sent m_4. Since $|Log(m)|_{p_1} = 3$, p_3 can infer that $|Log(m)|$ must be at least 4.

Π_{Log}: Process p_3 receives $Log(m)_{p_1}$ in addition to $\#m$. It then concludes that $Log(m)$ must include at least p_1, p_2, p_3, and p_4 and that $|Log(m)| \geq 4$.

Although Π_{Log} provides a more accurate assessment of $Log(m)$, both $\Pi_{|Log|}$ and Π_{Log} allow process p_3 to conclude that $|Log(m)| \geq 4$. The benefits of the extra information exchanged by protocol Π_{Log} become evident when process p_5 receives message m_5, at which point Π_{Log} has the most accurate determination of $|Log(m)|$.

Protocols $\Pi^+_{|Log|}$ and Π^+_{Log} are similar to $\Pi_{|Log|}$ and Π_{Log}, but can provide better estimates of $Log(m)$ and $|Log(m)|$. An example illustrating the difference between Π_{Log} and Π^+_{Log} is given in Figure 2. Assume $f = 3$. Determinant $\#m$ becomes stable when p_5 receives m_3. With Protocol Π_{Log}, when p_5 subsequently sends m_4 to p_3, $\#m$ is not piggybacked, and therefore message m_4 does not carry $Log(m)_{p_5}$. With Protocol

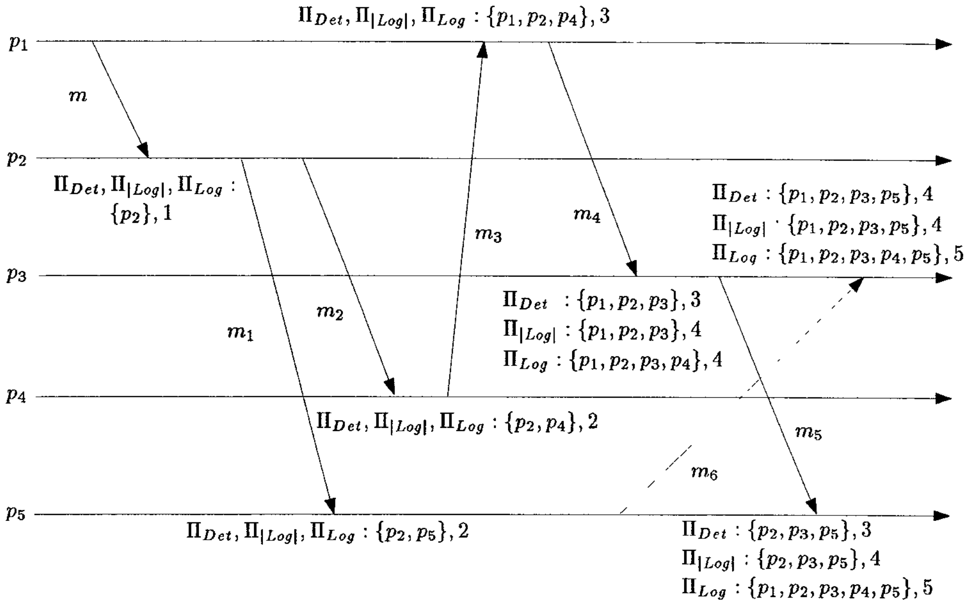
$\Pi_{Det}, \Pi_{|Log|}, \Pi_{Log} : \{p_1, p_2, p_4\}, 3$
p_1
m
p_2
$\Pi_{Det}, \Pi_{|Log|}, \Pi_{Log} : \{p_2\}, 1$
m_3
m_4
$\Pi_{Det} : \{p_1, p_2, p_3, p_5\}, 4$
$\Pi_{|Log|} : \{p_1, p_2, p_3, p_5\}, 4$
$\Pi_{Log} : \{p_1, p_2, p_3, p_4, p_5\}, 5$
p_3
m_2
m_1
$\Pi_{Det} : \{p_1, p_2, p_3\}, 3$
$\Pi_{|Log|} : \{p_1, p_2, p_3\}, 4$
$\Pi_{Log} : \{p_1, p_2, p_3, p_4\}, 4$
p_4
$\Pi_{Det}, \Pi_{|Log|}, \Pi_{Log} : \{p_2, p_4\}, 2$
m_5
m_6
p_5
$\Pi_{Det}, \Pi_{|Log|}, \Pi_{Log} : \{p_2, p_5\}, 2$
$\Pi_{Det} : \{p_2, p_3, p_5\}, 3$
$\Pi_{|Log|} : \{p_2, p_3, p_5\}, 4$
$\Pi_{Log} : \{p_1, p_2, p_3, p_4, p_5\}, 5$

Figure 1: $Log(m)_{p_i}$ and $|Log(m)|_{p_i}$ for Π_{Det}, $\Pi_{|Log|}$ and Π_{Log}.

Π^+_{Log} instead, p_5 piggybacks $Log(m)_{p_5}$ even if $\#m$ is already stable. Hence, using protocol Π_{Log} a subsequent message sent by p_3 will contain a piggybacked value of $\#m$, while using Protocol Π^+_{Log} it will not. A similar scenario can be constructed with Protocols $\Pi_{|Log|}$ and $\Pi^+_{|Log|}$. Note that $\Pi^+_{|Log|}$ and Π^+_{Log} can provide better estimates of $\Pi_{|Log|}$ and Π_{Log} even when $f = n$. This is somewhat surprising, since when $f = n$ the $\mathcal{DS}$ set of a process contains all the determinants in that process determinant log, and $\Pi_{|Log|}$ and Π_{Log} would appear to become identical to $\Pi^+_{|Log|}$ and Π^+_{Log}, respectively. To see why the differ, consider $\Pi_{Log}$and Π^+_{Log}. A process p using Π_{Log} piggybacks $Log(m)_p$ on a message to a process q only if $\#m$ is also piggybacked on the same message. Since channels are FIFO, p does not piggyback $\#m$ to q more than once, and therefore does not piggyback $Log(m)_p$ to q more than once. If p instead uses Π^+_{Log}, then, once $\#m$ is in p's determinant log, $Log(m)_p$ is piggybacked on every message p sends to q, and q can use $Log(m)_p$ repeatedly to update its own estimate of $Log(m)$.

Consider again the execution shown in Figure 1. As long as a process estimates that $|Log(m)|$ is less than three, the three protocols produce identical estimates. This is true in general: the estimates given by any FBL protocol will be identical as long as for all messages m, a sender's estimate of $|Log(m)|$ is less than three. The reason why this holds is that whenever a process q receives $\#m$ from process p, it can always conclude that $\{m.dest, p, q\} \subseteq Log(m)$. Hence, whenever $f < 3$ the most efficient FBL protocol is the one that piggybacks on each message the minimum amount of data needed to satisfy Property 2. We can therefore formulate the following general guideline:

If $f < 3$, then use Protocol Π_{Det}.

There are applications, however, for which Π_{Det} performs as well as Π_{Log} even for large values of f. For example, Figure 3 shows an application for which Π_{Det} does as well as Π^+_{Log} when $f = n$ [5]. The application is a parallel solution to the Synthetic Aperture Radar problem (SAR) in which radar echoes, collected by aircraft or spacecraft, are used to construct terrain contours. The steps necessary for producing high-quality images from SAR data consist of the following sequence of computations: two-dimensional discrete Fourier transform, binary convolution, two-dimensional inverse discrete Fourier transform, and intensity level normalization for visualization. For our purposes, however, the important property to note is that data flows in a particular manner.

To characterize a set of applications for which Π_{Det} performs as well as Π^+_{Log}, we represent an application's pattern of communication with a *channel graph*. For a given application, its associated channel graph is a directed graph. Nodes are used to represent processes as well as sources of application messages received form the environment and destinations of application messages sent to the environment, and edges are used to represent the direction application messages are sent.

Definition 1 *A channel graph is* tree-like *if, for all pairs of nodes i and j, all paths from i to j have the same length.*

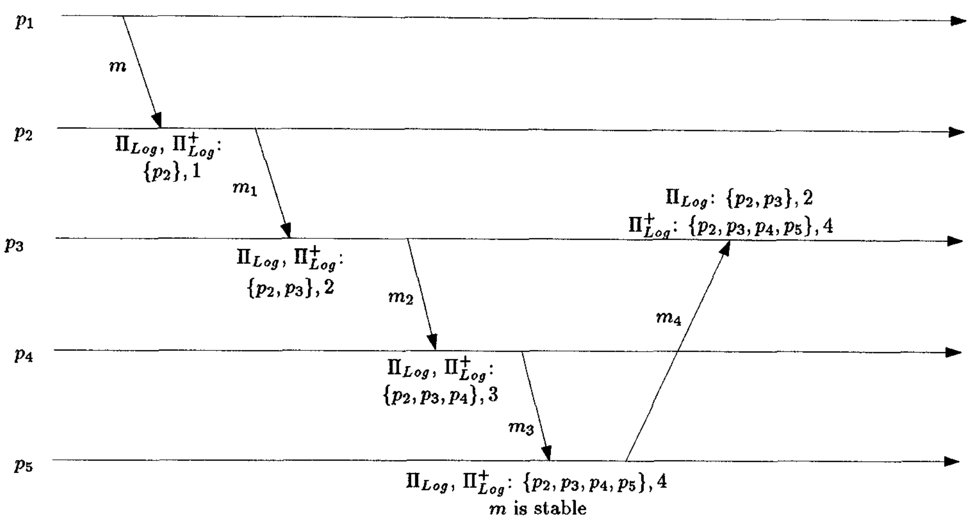
p_1
p_2
p_3
p_4
p_5
m
Π_{Log}, Π^+_{Log}: $\{p_2\}, 1$
m_1
Π_{Log}, Π^+_{Log}: $\{p_2, p_3\}, 2$
m_2
Π_{Log}, Π^+_{Log}: $\{p_2, p_3, p_4\}, 3$
m_3
Π_{Log}, Π^+_{Log}: $\{p_2, p_3, p_4, p_5\}, 4$
m is stable
m_4
Π_{Log}: $\{p_2, p_3\}, 2$
Π^+_{Log}: $\{p_2, p_3, p_4, p_5\}, 4$

Figure 2: Comparison of Π_{Log} and Π^+_{Log} for $f = 3$.

Note that the channel graph of Figure 3 is tree-like. The following theorem, proved in [1], characterizes one set of applications for which Π_{Det} performs as well as Π^+_{Log} when $f = n$.

Theorem 1 *Let $f = n$. Given a tree-like channel graph, for any run ρ, Protocol Π_{Det} piggybacks on each message the same determinants as Protocol Π^+_{Log}.*

There exist channel graphs for which Π_{Det} sends the same determinants as Π^+_{Log} even when $f < n$. The following theorem (also proved in [1]) specifies one such kind of graph.

Theorem 2 *Let $f \leq n$. Given a channel graph that is a tree (as opposed to a tree-like channel graph), for any run ρ Protocol Π_{Det} piggybacks on each message the same determinants as Protocol Π^+_{Log}.*

4.2.2 Piggyback Overheads

Even though it is only for $f < 3$ that Π_{Det} provably piggybacks the least amount of data of any FBL protocol, we expect that in practice Π_{Det} will piggyback overall fewer determinants that the other four protocols when f is small. This is important, since it indicates that applications that must tolerate only a small number of concurrent failures can use effectively the cheapest of all FBL protocols.

If f is large, however, then protocols like Π_{Det} that exchange less information may dramatically underestimate $Log(m)$ and $|Log(m)|$, and this can lead to excessive piggybacking of $\#m$. Hence, there is a trade-off between the amount of information carried in each message versus the number of unnecessary piggybacks.

As we have seen, how this trade-off works in practice for a particular application is largely a function of the application's pattern of communication and of the network's responsiveness in delivering acknowledgments. In order to understand the parameters of this trade-off, however, it is instructive to compare the amount of data that each protocol piggybacks on a message carrying a fixed number of determinants. For simplicity, in our calculations we don't consider optimizations achievable by applying compression techniques such as those described in [4, 1].

Consider a message m from process p to process q. Suppose that, when p sends m, $\mathcal{DS}_p(q)$ contains D determinants and p's determinant log contains N determinants. Let w denote the number of words needed to encode a determinant, and assume that the identity of a process and the number of processes that have logged a determinant can each be encoded in one word. A straightforward implementation of the five protocols piggybacks the following amount of words on m:

1. Π_{Det}: Dw words.
2. $\Pi_{|Log|}$: $D(w + 1)$ words, or D words more than Π_{Det}.
3. $\Pi^+_{|Log|}$: $Dw+N$ words, or N words more than Π_{Det}.
4. Π_{Log}: Up to $D(w + f)$ words, or Df words more than Π_{Det}.
5. Π^+_{Log}: Up to $Dd + Nf$ words, or Nf words more than Π_{Det}.

In the worst case, D and N can only be bound by the total number of delivery events d that causally precede the sending of m. Thus, the extra information sent by $\Pi_{|Log|}$, $\Pi^+_{|Log|}$, Π_{Log} and Π^+_{Log} does not worsen the theoretical asymptotically worst case behavior of FBL protocols. In practice, however, when D is large, adding an extra piggyback proportional to D, as $\Pi_{|Log|}$ and Π_{Log} do, can result in significant extra overhead. Furthermore, even when D is small, N is most likely large, making $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} appear even less practical. Hence, it could be advantageous to represent the extra information carried by $\Pi_{|Log|}$, $\Pi^{+}_{|Log|}$, Π_{Log} and Π^{+}_{Log} using a data structure whose size is independent of D or N.

Protocol $\Pi_{|Log|}$ can be easily modified to achieve this goal by sorting the determinants $\#m'$ piggybacked on m according to $|Log(m')|$. The resulting version of $\Pi_{|Log|}$ piggybacks no more than f additional words than Π_{Det}, an amount which is independent of D. A drawback of this approach, however, is that determinants sorted in this manner are not suitable for some of the compression techniques described in [4, 1], which can dramatically reduce the size of the piggyback. Furthermore, this approach can not be applied to $\Pi^{+}_{|Log|}$, Π_{Log} or Π^{+}_{Log}.

In the next section we introduce a data structure, called a *dependency matrix*, that will allow us to implement versions of $\Pi_{|Log|}$, $\Pi^{+}_{|Log|}$, Π_{Log} and Π^{+}_{Log} with a cost independent of D or N.

4.3 The Dependency Matrix

We exploit the relationship that exists in Property 2 between $Log(m)$ and $Depend(m)$. The following definition of $Log(m)$ satisfies Property 2:

$$Log(m) = \begin{cases} Depend(m) & \text{if } |Depend(m)| \leq f \\ \text{any set } \mathcal{S} : |\mathcal{S}| > f & \text{otherwise} \end{cases}$$

With this approach, a process can use $|Depend(m)|$ to evaluate $|Log(m)|$, and take advantage of techniques that have been developed to detect dependencies in asynchronous distributed systems. One such technique is based on *vector clocks* [13].

Strom and Yemini [18] were the first to use vector clocks with message logging when they introduced the notion of *dependency vector*. A dependency vector is a vector clock that is specialized to determine causal dependencies between delivery events occurring at different processes. Since in the piecewise deterministic model there is a one-to-one correspondence between delivery events and state intervals, dependency vectors can be used to determine dependencies among state intervals of different processes.

Let $deliver_p(m)$ denote the delivery of message m at process p, and let $DV_p(deliver_p(m))$ be the corresponding value of the dependency vector of process p:

> $DV_p(deliver_p(m))[p]$ is the index of the state interval initiated in p by event $deliver_p(m)$, as well as the receive sequence number of message m
>
> $DV_p(deliver_p(m))[q]$ is the highest index of any state interval of process q that process p depends upon, as well as the highest receive sequence number of any message delivered by process q that process p depends upon.

Furthermore, the vector clock update rules ensure that, given event $deliver_p(m)$ of process p and event $deliver_q(m')$ of process q, the following property holds:

$$deliver_p(m) \rightarrow deliver_q(m') \equiv \\ DV_p(deliver_p(m))[p] \leq DV_q(deliver_q(m'))[p] \qquad (3)$$

Dependency vectors are designed to track arbitrary dependencies between delivery events. In the context of FBL, we are interested in determining which processes depend on event $deliver_p(m)$ only when $|Depend(m)| \leq f$. We therefore introduce *weak dependency vectors* that satisfy the following weaker version of Property 3:

$$deliver_p(m) \rightarrow deliver_q(m') \wedge |Depend(m)| \leq f \Rightarrow \\ WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p] \quad (4.a)$$

$$WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p] \Rightarrow \\ deliver_p(m) \rightarrow deliver_q(m') \qquad (4.b)$$

where WDV_p and WDV_q are the weak dependency vectors of process p and q respectively.

Notice that from Properties 4.a, 4.b and from the definition of $Depend(m)$ it follows that, for any given message m for which $|Depend(m)| \leq f$, the membership of a generic process p in $Depend(m)$ can be determined at any point in time by reading process p's current weak dependency vector. In particular, the following conditions hold:

$$p \in Depend(m) \wedge |Depend(m)| \leq f \Rightarrow \\ WDV_p[m.dest] \geq m.rsn \qquad (5.a)$$

$$WDV_p[m.dest] \geq m.rsn \Rightarrow p \in Depend(m) \qquad (5.b)$$

The approach we adopt in our implementation of FBL in order to evaluate $|Depend(m)|$ derives directly from the above observation. We require each process p to maintain an $n \times n$ *dependency matrix* D_p, defined as follows:

- $D_p[p,*]$ is the weak dependency vector of process p
- $D_p[q,*]$ is process p's estimate of the weak dependency vector of process q

where q is a generic process distinct from p and $D_p[i,*]$ denotes the i-th row of matrix D_p.

Note that the estimate of the weak dependency vector of a generic process q maintained by process p in $D_p[q,*]$ will not in general be able to satisfy Condition 5.a, since the distributed and asynchronous nature of our system will not in general allow process p's estimate to be perfectly accurate.

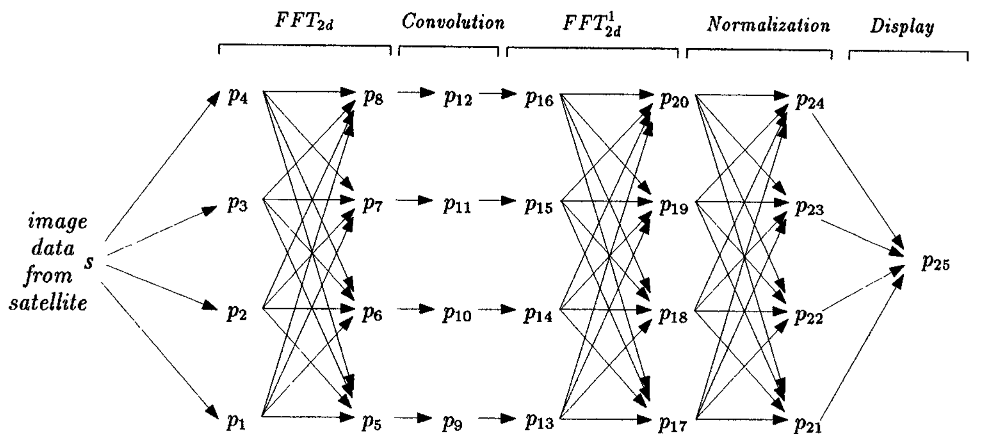

Figure 3: A parallel solution to the Synthetic Aperture Radar problem.

However, it is straightforward to design update rules that will ensure that condition 5.b holds. This provides process p with a simple method to estimate $|Depend(m)|$ and therefore $|Log(m)|$, for a particular message m: process p can just check how many entries of $D_p[*, m.\text{dest}]$, the column corresponding to process $m.dest$, are greater than, or equal to, $m.rsn$. In particular, process p will consider $\#m$ to be stable if more than f entries of $D_p[*, m.\text{dest}]$ are greater than, or equal to, $m.rsn$.

Because the order of events executed by a processor is in fact a total order, it is also straightforward to construct a dependency matrix that has size $n_P \times n_P$ where n_P is the number of processors in the system [1]. When, in the following, we discuss the cost of the protocols, we assume that this smaller representation of the dependency matrix is used.

4.4 Piggybacking the Dependency Matrix

Since the dependency matrix of process p can be used to compute $Log(m)_p$ for all the messages for which p is a member of $Depend(m)$, when p sends a message to q it can simply piggyback on it its dependency matrix. Process q can then use the piggybacked dependency matrix, the piggybacked determinants, and its own dependency matrix to compute new values of $Log(m)_q$ and $|Log(m)|_q$ for all messages m whose determinants are logged in q's address space. This protocol is another implementation of Π^{+}_{Log} that piggybacks $n_P{}^2$ additional data over Π_{Det}, independent of the number of determinants D.

Similarly, an implementation of $\Pi^{+}_{|Log|}$ that is analogous to $\Pi_{|Log|}$ and that piggybacks fn_P additional data can be derived by extracting the following data structure from the dependency matrix.

Stability Matrix: $SMat_p$ is a $(f+1) \times n$ matrix of integers. For all processes q in $\mathcal{N}$, $SMat_p[i, q]$ contains the highest receive sequence number of any message m delivered by q for which $|Log(m)|_p = i$. The entries of $SMat_p$ are initialized to 0. Notice that $SMat_p[1, :]$ is equal to WDV_p, and that $SMat_p[f+1, :]$ is equal to $Stable_p$.

Thus, $|Log(m)|_p = i$ when $\text{SMat}_p[i+1, m.dest] < m.rsn \leq \text{SMat}_p[i, m.dest]$. Protocol $\Pi^{+}_{|Log|}$ then has process p piggyback its stability matrix instead of the dependency matrix. A process q that receives this stability matrix can use it with the piggybacked determinants, its own stability matrix and its own dependency matrix to compute the new values for its stability matrix and dependency matrix.

Full descriptions of protocols $\Pi^{+}_{|Log|}$ and Π^{+}_{Log}can be found in [1].

5 Manetho and FBL Protocols

Manetho is an optimal message logging protocol designed for $f = n$. In Manetho, each process maintains an *antecedence graph* AG that records the causal relationship between all the message delivery events of an execution.[1] The nodes of the antecedence graph represent the state intervals started by each delivery event, and contain the determinants of the corresponding delivery events. In particular, if process p is executing in state interval $p[i]$, then the antecedence graph of p contains the determinant of the non-deterministic event e that started $p[i]$, plus the determinants of all the non-deterministic events that are in e's causal past. Conceptually, when process p sends a message m to process q, p piggybacks on m the current value of its antecedence graph. In practice, optimizations are used to limit the amount of piggybacked data.

[1]The papers on Manetho also mention recording determinants for nondeterministic internal events. FBL could be extended in a similar way but would need, like Manetho, to be executed on an operating system that allowed for the deterministic re-execution of nondeterministic internal event.

Since the antecedence graph records the causal relationship between the delivery events of an execution, it can be used to compute both $|Log(m)|$ and $Log(m)$ for any message m. Hence, Manetho can be thought of as providing a different representation of the same information piggybacked Π_{Log}. Since every node in an antecedent graph can have no more than two edges into it, Manetho piggybacks no more than $D(d+2)$ words, or $2D$ more words than $\Pi_0(n)$. This is better than Protocol $\Pi_{2a}(n)$, which piggybacks up to $n_P D$ more than $\Pi_0(n)$, and is better than Protocol $\Pi_{2a}(n)$ when $D > {n_P}^2/2$. However, the information carried by $\Pi_{2a}(n)$ and $\Pi_{2b}(n)$ can be compressed, while we are not aware of techniques for compressing the determinants carried by Manetho. The effects of compression can be very large [4], and so we expect in practice that Protocols $\Pi_{2a}(n)$ and $\Pi_{2b}(n)$ will often piggyback much less information than Manetho.

One major difference between Manetho and the FBL protocols is that Manetho assumes $f = n$. Applications for which a smaller value of f would suffice must nonetheless pay the full cost of ensuring resiliency from total failure. This cost is not only found in message traffic, but also in the logging of determinants in volatile storage: processes using FBL for $f < n$ fill their volatile logs more slowly, and therefore need to take checkpoints less frequently.

Another difference between Manetho and FBL is that simpler approximations of $Log(m)$ and $|Log(m)|$ can be used with FBL. Protocols $\Pi_{|Log|}$ and Π_{Log} both piggyback $O(D)$ less information than Manetho. As described in Section 4.2.1, when f is small the simpler approximations work very well. In addition, some applications are very amenable to the simpler approximations; for example, when $f = n$ and communication is tree-like, then the simplest protocol Π_{Det} is the most efficient.

A third difference has to do with recovery and garbage collection. Manetho maintains dependency relations in the antecedence graph. The antecedence graph is a powerful data structure, but is relatively difficult to reconstitute during recovery. In particular, after a failure the antecedence graph of the recovering process must be reconstituted through a non-trivial merging of the antecedence graphs of the surviving processes. In addition, garbage collection is recognized to be complex and expensive [9]. We suspect that part of the cost may arise because determinants must be logged in a way that preserves the antecedence graph structure which may not lend itself easily to garbage collection. In FBL, the dependency relation is not represented by maintaining explicit relation between determinants, but rather through the dependency matrix discussed in Section 4.3. Decoupling the representation of the dependency relation from the determinants allows FBL to easily structure the logs to allow for efficient garbage collection. Furthermore, the dependency matrix can be reconstituted easily from the matrices of the surviving processes, and recovery is a straightforward procedure.

6 Conclusions

In this paper, we presented five families of optimal message logging protocols. The simplest, Protocol Π_{Det} piggybacks the least amount of information and is the best choice for $f < 3$. It is also the best choice when communication is acyclic and $f = n$. Protocols $\Pi_{|Log|}$ and $\Pi^+_{|Log|}$ piggyback more information but are more efficient on average than Π_{Det} for certain applications and values of f. Protocols Π_{Log} and Π^+_{Log} piggyback even more information but again are the most efficient protocols in certain situations.

All optimal protocols must piggyback determinants and so, ignoring the effects of compression, a message carrying D determinants must carry Dd words where d is the size of a determinant. Protocol Π_{Det} piggybacks exactly this amount. Protocols $\Pi_{|Log|}$ and Π_{Log} carry additional data whose amount (D and up to Df respectively) scale with D. Since for some applications D can become quite large, Protocols $\Pi^+_{|Log|}$ and Π^+_{Log} may be more appropriate than $\Pi_{|Log|}$ and Π_{Log} because the amount of additional data that they carry over what Protocol Π_{Det} carries is independent of D (proportional to $(f+1)n_P$ and ${n_P}^2$ respectively).

We compared FBL with Manetho, which is the only other optimal protocol that we are aware of. Manetho provides the same information as Π^+_{Log}, and so the arguments for using a protocol simpler than Π^+_{Log} apply to using a protocol simpler than Manetho as well. Manetho uses a more compact representation than Π_{Log} (with a cost of $D(d+2)$ rather than $D(d+f)$ words). $\Pi^+_{|Log|}$ is more data efficient than Manetho if $D > {n_P}^2/2$. However, the FBL protocols are all well-suited for efficient compression of the D determinants [4]. We do not know if Manetho is equally well-suited for compression.

The only FBL protocol that has been completely implemented is Π_{Det} [4]. We are currently implementing the complete FBL family in order to understand better under what circumstances the simpler protocols are the more efficient ones. We have also extended the message logging specification to distributed shared memory architectures [2] and have designed optimal FBL protocols for the entry consistency memory coherency model [6].

Acknowledgments We would like to thank Bruce Hoppe and Fred Schneider for their help in refining our ideas and Wanda Chiu for her detailed comments on our work. We would also like to thank Mootaz Elnozahy for

his comments on an earlier draft of this paper and for helping us understanding Manetho.

References

[1] L. Alvisi. *Understanding the Message Logging Paradigm for Masking Process Crashes.* PhD thesis, Cornell University Department of Computer Science, January 1996.

[2] L. Alvisi and K. Marzullo. Deriving optimal checkpointing protocols for distributed shared memory architectures. In *Selected Papers, International Workshop in Theory and Practice in Distributed Systems*, pages 111–120. Springer–Verlag, 1995.

[3] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236. IEEE Computer Society, May 1995.

[4] 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.

[5] O. Babaoglu, L. Alvisi, et al. Paralex: An environment for parallel programming in distributed systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 178–187, July 1992.

[6] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of the 93 COMPCON Conference*, pages 528–537. IEEE, February 1993.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] Friedmann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevir Science Publishers B. V., 1989.

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

[15] 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.

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

[17] 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.

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

[19] 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.

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