Causality Tracking in Causal Message-Logging Protocols

Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
Austin, Texas

Karan Bhatia
Entropia Inc.
La Jolla, California

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

Abstract

Casual message-logging protocols have several attractive properties: they introduce no blocking, send no additional messages over those sent by the application, and never create orphans. Causal message logging, however, does require the casual effects of the deliveries of messages to be tracked. The information concerning causality tracking is piggybacked on application messages, and the amount of such information can become large.

In this paper we study the cost of tracking causality in causal message-logging protocols. One can track causality as accurately as possible, but to do so requires piggybacking a considerable amount of additional information. One can reduce the amount of piggybacked information on each message by reducing the accuracy of causality tracking. But then, causal message logging may piggyback the reduced amount of information on more messages.

We specify six different methods of tracking causality, each representing a natural choice based on the specification of causal message logging. We describe how these six methods can be implemented and compare them in terms of how large of a piggyback load they impose. This load depends on the application that is using causal message logging. We characterize some applications for which a given method has the smallest piggyback load, and study using simulation the size of the piggyback load for two different models of applications.

1 Introduction

Message logging [9] 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 received since recording 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 an 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, message logging protocols guarantee—either through careful logging or through a somewhat complex recovery protocol—that after recovery no process is an orphan.

Message logging protocols can be *pessimistic* (for example, [5, 11, 17, 24]), *optimistic* (for example, [12, 22, 23, 26]), or *causal* [4]. Like pessimistic protocols, causal protocols [3, 10] never create orphans, and, like optimistic protocols, they do not log synchronously to stable storage. They are able to do this by piggybacking information onto the ambient message traffic.

Causal message-logging protocols track the causal effects of message deliveries. Let f be the number of concurrent crash failures that are to be tolerated. We have given [4] a generic causal message-logging protocol that tracks causality to determine when information needed for recovery has been delivered and recorded by at least $f+1$ independently-failing processes.

In this paper we study the cost of tracking causality in causal message-logging protocols. This is not an easy problem to address. One can track causality as accurately as possible, but to do so requires piggybacking on the application messages a considerable amount of additional information. One can reduce the amount of piggybacked information on each message by reducing the accuracy of causality tracking. But then, causal message logging may piggyback the reduced amount of information on more messages because the protocol may learn more slowly when the recovery information has been replicated at least $f+1$ times.

Understanding which method piggybacks the least information in a given situation is important for several reasons. First, it is in itself an interesting question, because the tradeoff is complex and there is a temptation either to be as accurate as possible or to use as little information as possible to track causality. As this paper shows, there are times when neither is the best choice in terms of message size. Second, there are environments, such as embedded systems or mobile systems, in which bandwidth is limited. In such systems, limiting the size of messages is important. Third, a significant cost in any protocol is in assembling, processing, and disassembling a message. Piggybacking less information in messages is one way to improve the performance of a causal message-logging protocol.

We consider six different methods of tracking causality. They represent natural choices based on the specification of causal message logging. All of the published causal message-logging protocols track causality using one of these methods. We describe how these six methods can be implemented. We compare them in terms of how large a piggyback load they impose. This load is application dependent: we characterize some applications for which a given method has the smallest piggyback load, and study using simulation the size of the piggyback load for two different models of applications.

We do not consider the effect on the piggyback load when processes periodically checkpoint their states. Frequent checkpointing can reduce the piggyback load because one doesn't need to track causality for events prior to a checkpoint. But, frequent checkpointing imposes another kind of overhead. The results here should be illustrative for executions in which checkpointing is relatively infrequent.

We do not present the protocol that is run when a crashed process recovers. All six protocols in this paper can use the same recovery protocol. A discussion on recovery as well as the actual recovery protocol can be found in [18].

The paper proceeds as follows. In Section 2 we present the system model and in Section 3 we specify causal message logging. Section 4 develops the six causal message-logging protocols and identifies two classes of applications for which the simplest protocol is also the most efficient in terms of piggyback overhead. In Section 5 we measure and compare the piggyback overheads using a synthetic application. Section 6 concludes the paper.

2 System Model

We assume a system $\mathcal{N}$ of n processes that can communicate only by exchanging messages. The system is asynchronous: there exists no bound on the relative speeds of processes, no bound on message transmission delays, and no global time source.

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 [13]. 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 [6]; otherwise it is *inconsistent*.

We assume that processes are *piecewise deterministic* [24] in that the only nondeterminism in a process arises from the nondeterministic order in which messages that have been received 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 ℓth message delivered by p [23]. 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 [19];
- The fixed set of processes that belong to the system is known by all of these processes;
- Channels are point-to-point, FIFO, and fail by intermittently losing messages.

3 Specification of Causal Message Logging

With the assumption that processes are piecewise deterministic, the only non-deterministic choices made during an execution concern the order in which messages are delivered to processes. To recover a process's state, the nondeterministic choices the process makes during recovery should be the same as it made before failing. Hence, we need to represent the order of message deliveries.

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 $\langle m.source, m.ssn, m.dest, m.rsn\rangle$ 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,

$$Depend(m) \stackrel{\text{def}}{=} \left\{ j \in \mathcal{N} \;\middle|\; \begin{array}{ll} \vee & ((j = m.dest) \wedge j \text{ has delivered } m) \\ \vee & (\exists m' \colon (deliver_{m.dest}(m) \rightarrow deliver_j(m'))) \end{array} \right\}$$

where $\rightarrow$ denotes the *happens-before* relationship [13]. 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 [4], we showed that the following property ensures that sufficient information is available to avoid the creation of orphans:

$$\forall m : \Box(Depend(m) \subseteq Log(m)) \tag{1}$$

where $\Box$ is the temporal "always" operator.

We say that $\#m$ is *stable* (denoted $stable(m)$) when $\#m$ cannot be lost because of crashes. Property 1 need hold only for messages with a determinant that is not stable. In [4], we showed that the following property ensures that no set of crashed processes can lead to the creation of orphans:

$$\forall m : \Box(\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m))) \tag{2}$$

If determinants are kept in stable memory, then $stable(m)$ 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 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, Property 2 can be written:

$$\forall m : \Box((|Log(m)| \leq f) \Rightarrow (Depend(m) \subseteq Log(m))) \tag{3}$$

Property 3 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 3:

$$\begin{aligned} &\forall m : \Box(|Log(m)| \leq f \Rightarrow \\ &\qquad Depend(m) \subseteq Log(m) \ \wedge \ \Diamond(Depend(m) = Log(m)) \end{aligned} \tag{4}$$

where $\Diamond$ is the temporal "eventually" operator. This characterization strongly couples logging with causal dependency on deliver events. It requires that as long as $|Log(m)| \leq f$:

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

We call the protocols that implement Property 4 *causal message-logging protocols.*

4 Family Based Logging

Family Based Logging (FBL) is a logging technique that implements Property 4[1]. Conceptually, each process p maintains in its volatile storage a set of determinants DL_p called the *determinant log* of p and defined as follows:

$$DL_p \stackrel{\text{def}}{=} \{\#m : p \in \mathit{Depend}(m)\}.$$

That is, DL_p contains the determinant of all of the delivery events that causally precede p's current state. We denote with $\mathit{UnstableDL}_p$ the subset of DL_p that p does not know to be stable. Whenever p sends a message m' to some process q, process p piggybacks onto m' all the determinants $\#m$ in $\mathit{UnstableDL}_p$ for which $q \notin \mathit{Log}(m)$. Hence, a fundamental issue of implementing FBL is how a process p determines $\mathit{Log}(m)$ for any determinant $\#m$ that p has received. In general, p may not know the exact values of $\mathit{Log}(m)$ and $|\mathit{Log}(m)|$, and so it must estimate these values. We denote p's estimated values for $\mathit{Log}(m)$ and $|\mathit{Log}(m)|$ as $\mathit{Log}(m)_p$ and $|\mathit{Log}(m)|_p$ respectively.

4.1 Estimating $\mathit{Log}(m)$ and $|\mathit{Log}(m)|$

To satisfy Property 4, p must never overestimate $\mathit{Log}(m)$ or $|\mathit{Log}(m)|$. However, if p underestimates $|\mathit{Log}(m)|$, it may then needlessly piggyback determinants that are already stable, making the messages on average significantly larger. By exchanging more information, processes can improve the accuracy of their estimates and avoid piggybacking useless data; piggybacking this extra information can in turn make the messages significantly larger.

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

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

$|\mathit{Log}(m)|_p$ Upon receipt of $|\mathit{Log}(m)|_p$, q can safely infer that $|\mathit{Log}(m)|$ is no smaller than $|\mathit{Log}(m)|_p$. When q receives $\#m$ for the first time, q can further safely infer that $|\mathit{Log}(m)|$ must be at least equal to $|\mathit{Log}(m)|_p + 1$, since q itself could not be counted in $|\mathit{Log}(m)|_p$. Note that this scheme allows q to infer a value for $|\mathit{Log}(m)|$ safely without knowing the identity of the processes in $\mathit{Log}(m)$.

$\mathit{Log}(m)_p$ Upon receipt of $\mathit{Log}(m)_p$, process q can safely infer that $\mathit{Log}(m)_q$ must be at least equal to the union of the current set $\mathit{Log}(m)_q$ and $\mathit{Log}(m)_p$, and can update $|\mathit{Log}(m)|_q$ accordingly. Using this scheme, when process p sends its estimate of $\mathit{Log}(m)$ to process q, it is providing q with the union of all the estimates relative to $\mathit{Log}(m)$ computed by the processes along the causal path that connects process $m.dest$ to process p.

One can define a protocol for each of these different information-exchange schemes. Let

$$\mathit{UnstableDL}_p(q) \stackrel{\text{def}}{=} \{\#m \in \mathit{UnstableDL}_p : q \notin \mathit{Log}(m)_p\}.$$

That is, $\mathit{UnstableDL}_p(q)$ is the the set of determinants in $\mathit{UnstableDL}_p$ that p does not know q already has. Let p send a message m' to q. The three protocols piggyback as follows:

[1] It is conceptually simple, though somewhat cumbersome, to generalize our discussion of FBL so that it implements a more general version of Property 4, i.e. one that uses the more general predicate $\neg \mathit{stable}(m)$ instead of $|\mathit{Log}(m)| \leq f$. We use the latter in this paper to simplify our exposition.

Π_{Det} Process p piggybacks the determinants in $\mathit{UnstableDL}_p(q)$ on m'.

$\Pi_{|Log|}$ For each determinant $\#m$ in $\mathit{UnstableDL}_p(q)$, process p piggybacks both $\#m$ and $|Log(m)|_p$ on m'.

Π_{Log} For each determinant $\#m$ in $\mathit{UnstableDL}_p(q)$, process p piggybacks both $\#m$ and $Log(m)_p$ on m'.

Furthermore, for each of these three protocols, when p receives an acknowledgment from q for message m', p adds q to $Log(m)$ for each determinant $\#m$ piggybacked on m'.

The causal message-logging protocol Manetho [10] is essentially Π_{Det} with $f = n$. That is, Manetho assumes that total failures are possible, which means that a determinant never becomes stable [2].

Hence, a process piggybacks $\#m$ on a message m' to q when p has a copy of $\#m$ and p does not know that q has a copy of $\#m$. [8]

In the three protocols defined above, a process piggybacks information to q only about determinants that are in $\mathit{UnstableDL}_p(q)$. To disseminate more quickly that a determinant has become stable, however, a process can piggyback additional information. The following three protocols, which are analogous to Π_{Det}, $\Pi_{|Log|}$ and Π_{Log}, piggyback such information. Suppose p sends a message m' to q. The three protocols piggyback as follows:

Π^+_{Det} Process p piggybacks the same data as in Π_{Det}. In addition, p informs q of which determinants in DL_p have become stable.

$\Pi^+_{|Log|}$ Process p piggybacks the same data as in $\Pi_{|Log|}$. In addition, if $|Log(m)|_p$ has increased since the last time p piggybacked $\#m$ to q, then p piggybacks $|Log(m)|_p$ on m'.

Π^+_{Log} Process p piggybacks the same data as in Π_{Log}. In addition, if $Log(m)_p$ has increased since the last time p piggybacked $\#m$ to q, then p piggybacks $Log(m)_p$ on m'.

4.2 Comparison of the Protocols

The six 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, the three protocols compute different estimates for $Log(m)$ and $|Log(m)|$:

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 $p_1 = m_4.\mathit{source}$ and process $p_2 = m.\mathit{dest}$. Process p_3 thus sets $Log(m)_{p_3} = \{p_1, p_2, p_3\}$, and $|Log(m)|_{p_3} = 3$.

[2]This is because we equate $\mathit{stable}(m)$ with $|Log(m)| \leq f$. In Manetho, as in any message logging protocol, a determinant can always be made stable by writing it to any other suitable implementation of stable storage, *e.g.* a disk.

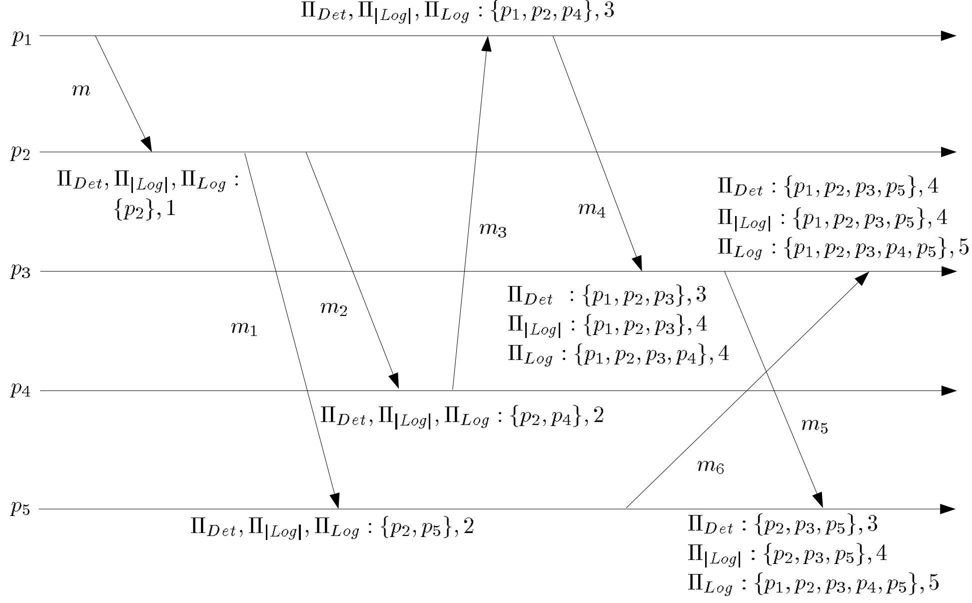


Figure 1: $Log(m)_{p_i}$ and $|Log(m)|_{p_i}$ for Π_{Det}, $\Pi_{|Log|}$ and Π_{Log}.

$\Pi_{|Log|}$ 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 $|Log(m)| \geq 4$.

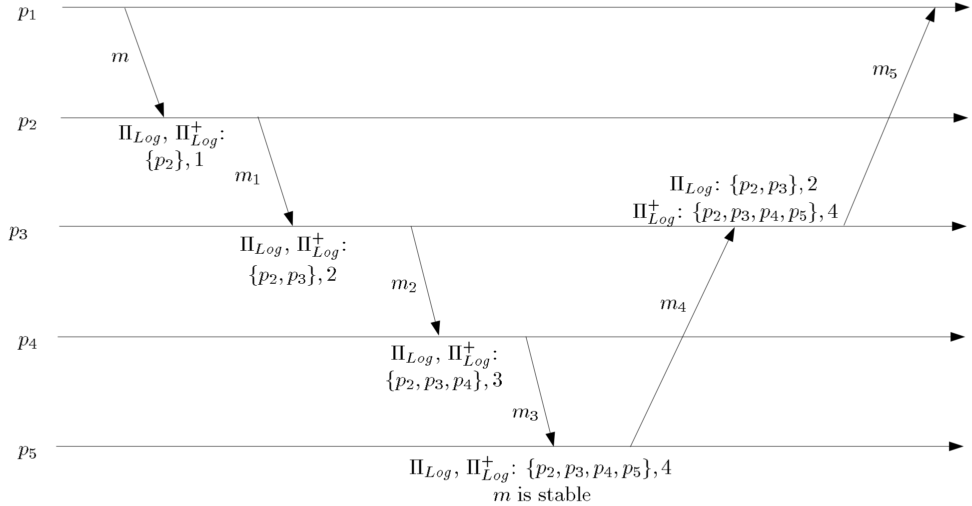


Figure 2: Comparison of Π_{Log} and Π^+_{Log} for $f = 3$.

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 $|Log(m)|$.

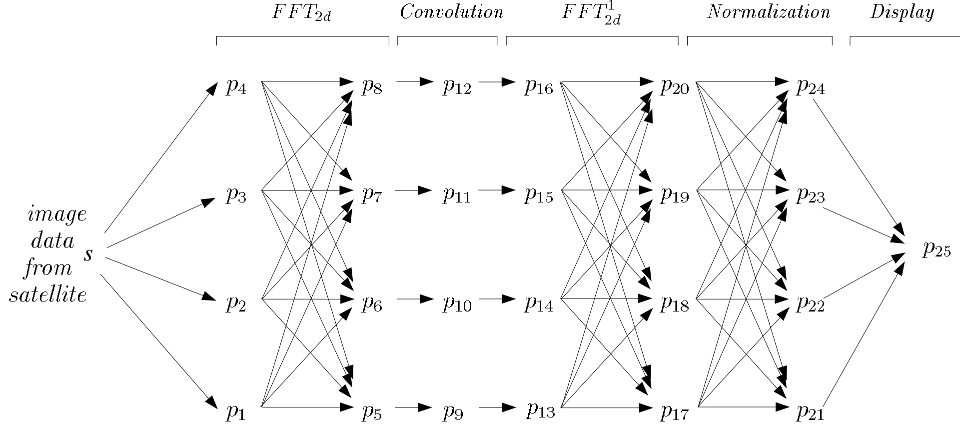


Figure 3: A parallel solution to the Synthetic Aperture Radar problem.

Protocols Π^+_{Det}, $\Pi^+_{|Log|}$ and Π^+_{Log} are similar to Π_{Det}, $\Pi_{|Log|}$ 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 Π^+_{Log} instead, p_5 piggybacks $Log(m)_{p_5}$ even if $\#m$ is already stable. Hence, using Protocol Π_{Log} a message m_5 sent by p_3 to p_1 will contain a piggybacked value of $\#m$, while using Protocol Π^+_{Log} it will not. Similar scenarios can be constructed with the other two pairs of protocols.

Consider again the execution shown in Figure 1. As long as $|Log(m)|$ is small, the protocols have the same estimates of $Log(m)$. This suggests that for small values of f, one should use Π_{Det} because it piggybacks the least possible amount of information per message. We examine this hypothesis in Section 4.2.2. There are applications, however, with 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 $f = n$. The application is a parallel solution to the Synthetic Aperture Radar problem (SAR) [15] 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 from the environment and destinations of application messages sent to the environment, and edges are used to represent the direction that application messages are sent.

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

The channel graph of Figure 3 is shortcut-free. The following theorem characterizes one set of applications for which Π_{Det} performs as well as Π^+_{Log} when $f = n$.

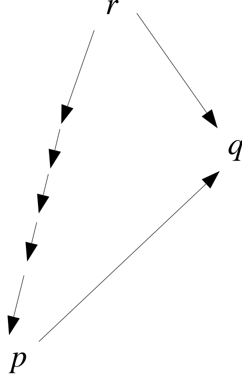


Figure 4: The channel graph obtained assuming that Π_{Log}^{+} estimates $Log(m)_{p_i}$ better than Π_{Det}.

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

Proof: When $f = n$ the antecedent of Property 4 is trivially true, and so with any FBL protocol a process p will piggyback a determinant $\#m$ when sending a message m' to q if and only if $p \in Depend(m)$ and $q \notin Log(m)_p$. Whether or not p is in $Depend(m)$ does not depend on the specifics of a particular FBL protocol, but is determined solely by the application messages. Hence, we can prove the theorem by showing that $q \notin Log(m)_p$ under Π_{Det} if and only if $q \notin Log(m)_p$ under Π_{Log}^{+}.

Assume that $q \notin Log(m)_p$ under Π_{Log}^{+}. Since Π_{Log}^{+} piggybacks a superset of the information piggybacked by Π_{Det}, p under Π_{Log}^{+} will estimate $Log(m)$ at least as accurately as Π_{Det}: $Log(m)_p$ under Π_{Det} is a subset of $Log(m)_p$ under Π_{Log}^{+}. Hence, $q \notin Log(m)_p$ under Π_{Det}.

Assume that $q \notin Log(m)_p$ under Π_{Det}. For $q \in Log(m)_p$ to hold under Π_{Log}^{+}, there must exist a causal path from node q to node p carrying this information. This path cannot be made solely of application messages, or the channel graph would contain a cycle and therefore would not be shortcut-free. Hence, the dependency must have been carried by an acknowledgment from process q to a third process r. Furthermore, $r \neq p$ since Π_{Det} and Π_{Log}^{+} do not differ in how they use acknowledgments to estimate $Log(m)$ and by assumption under Π_{Det} $q \notin Log(m)_p$. Furthermore, since q sent an acknowledgment to r, an application message was sent by r to q. We conclude that in order for q to be a member of $Log(m)_p$ under $\Pi_{Log}^{+}$the channel graph must contain (i) an edge from r to q, (ii) a path from r to p, and (iii) an edge from p to q.

Figure 4 shows such a channel graph. To show that this graph cannot be shortcut-free, we observe that there are two paths of different length that connect r and q: the first consists only of the edge from r to q while the second goes through p.

We conclude that for all shortcut-free channel graphs, if Π_{Det} estimates that $p \notin Log(m)_p$, then so does Π_{Log}^{+} . $\Box$

4.2.2 Piggyback Overhead

Protocols like Π_{Det} that exchange less information may dramatically underestimate $Log(m)$ and $|Log(m)|$, possibly leading to excessive piggybacking of $\#m$. On the other hand, by piggybacking less information, the piggyback load per message may be smaller. Hence, there is a trade-off between the amount of information carried in each message versus the number of unnecessary piggybacks.

This trade-off is complex, since it depends both on the application's pattern of communication and on the network's responsiveness in delivering acknowledgments: we explore it in detail in Section 5. Even a simple qualitative analysis, however, shows that, while for Π_{Det}, $\Pi_{|Log|}$, and Π_{Log} the amount of piggybacked information is proportional to the number D of determinants $UnstableDL_p(q)$, for Π^+_{Det}, $\Pi^+_{|Log|}$, and Π^+_{Log} this information may in the worst case be proportional to the number N of determinants in $DL_p(q)$.

In the worst case, both D and N can only be bound by the total number of delivery events 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 Π^+_{Det}, $\Pi^+_{|Log|}$ and Π^+_{Log} appear even less practical. Hence, it could be advantageous to represent the extra information 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 $|Log(m')|$. One can then, for example, also piggyback an f element array x where $x[i]$ is the number of determinants that have $|Log(m')|$. The array x can also be run encoded should it be sparse. 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 [2, 3], which can dramatically reduce the size of the piggyback. Furthermore, while this this approach can also be applied to $\Pi^+_{|Log|}$, it can not be applied to Π_{Log} or Π^+_{Log}.

In the next section we introduce a data structure, called a *dependency matrix*, that allows us to implement Π^+_{Det}, $\Pi^+_{|Log|}$, and Π^+_{Log} with an incremental cost over Π_{Det} that is independent of D or N.

4.3 Dependency Tracking

We know from Property 4 that as long as $|Log(m)| \leq f$, each process ensures that $Depend(m) \subseteq Log(m)$. Hence, a process can use $Depend(m)$ to estimate $Log(m)$. We can take advantage of techniques for tracking dependencies to compute $Depend(m)$. The most widely-used technique is based on *vector clocks* [16].

A vector clock is an n-element vector that counts the number of relevant events in the causal past of a process for some definition of relevant. Let V_p be the vector clock associated with process p. The value $V_p[p]$ counts the number of relevant events that p has executed, and $V_p[q], q \neq p$ counts the number of relevant events that p knows that q has executed. Hence, given two relevant events e_p of process p and e_q of process q,

$$e_p \rightarrow e_q \equiv V_p(e_p)[p] \leq V_q(e_q)[p] \quad (5)$$

where $V_p(e_p)$ and $V_q(e_q)$ are the vector clocks of process p and q when they execute e_p and e_q respectively.

Vector clocks are easy to implement. When a process p executes a relevant event, it increments $V_p[p]$. And, when a process p executes $receive_p(m)$, then for all $r : 1 \leq r \leq n, r \neq p$: $V_p(receive_p(m))[r]$ is set to the maximum of p's previous value for $V_p[r]$ and $V_q(send_q(m) \text{ to } p)[r]$. This second rule requires the sending process q to piggyback the current value of its vector clock on m.

Strom and Yemini [23] were the first to use vector clocks with message logging when they introduced the notion of a *dependency vector*. A dependency vector DV_p is a vector clock where

the relevant events are delivery events. Specifically,

> $DV_p(e)[p]$ is the index of the state interval that contains the event e. This is the same as the receive sequence number of the last message delivered by p through the execution of e.

> $DV_p(e)[q]$ is the highest index of any state interval of process q that process p depends upon through the execution of event e.

Specializing Equation 5 to dependency vectors, we get:

$$\begin{aligned}&\mathit{deliver}_p(m) \rightarrow \mathit{deliver}_q(m') \equiv \\ &DV_p(\mathit{deliver}_p(m))[p] \leq DV_q(\mathit{deliver}_q(m'))[p]\end{aligned} \tag{6}$$

Dependency vectors track arbitrary dependencies between delivery events. In the context of FBL, we are interested in determining which processes depend on event $\mathit{deliver}_p(m)$ only when $|\mathit{Log}(m)| \leq f$. We therefore define an abstraction, which we call *weak dependency vector WDV*, that satisfies the following weaker version of Condition 6:

$$\begin{aligned}&\mathit{deliver}_p(m) \rightarrow \mathit{deliver}_q(m') \wedge |\mathit{Log}(m)| \leq f \Rightarrow \\ &WDV_p(\mathit{deliver}_p(m))[p] \leq WDV_q(\mathit{deliver}_q(m'))[p]\end{aligned} \tag{7.a}$$

$$\begin{aligned}&WDV_p(\mathit{deliver}_p(m))[p] \leq WDV_q(\mathit{deliver}_q(m'))[p] \Rightarrow \\ &\mathit{deliver}_p(m) \rightarrow \mathit{deliver}_q(m')\end{aligned} \tag{7.b}$$

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

From Properties 7.a and 7.b, the definition of $\mathit{Depend}(m)$, and the fact that $\mathit{Depend}(m) \subseteq \mathit{Log}(m)$ it follows that, for any given message m for which $|\mathit{Depend}(m)| \leq f$ one can determine if q is in $\mathit{Depend}(m)$ from q's current weak dependency vector. In particular, the following conditions hold:

$$\begin{aligned}&q \in \mathit{Depend}(m) \wedge |\mathit{Depend}(m)| \leq f \Rightarrow \\ &\quad WDV_q[m.\mathit{dest}] \geq m.\mathit{rsn}\end{aligned} \tag{8.a}$$

$$WDV_q[m.\mathit{dest}] \geq m.\mathit{rsn} \Rightarrow q \in \mathit{Depend}(m) \tag{8.b}$$

One can define useful vector clocks that are weaker than weak dependency vectors. For example, it is useful to define a vector clock $\mathit{PBC}(m)$ that is constructed from the set of determinants piggybacked on a message m. This vector clock, which only satisfies Condition 8.b, is constructed as follows:

$$\mathit{PBC}(m)[p] \stackrel{\text{def}}{=} \begin{cases} \ell & \text{where } \ell \text{ is the largest value of } m'.\mathit{rsn} \text{ for all} \\ & \text{determinants } \#m' \text{ piggybacked on } m \text{ such} \\ & \text{that } m'.\mathit{dest} = p \\ 0 & \text{if there is no determinant } \#m' \text{ piggybacked on } m \\ & \text{such that } m'.\mathit{dest} = p \end{cases}$$

An element $\mathit{PBC}(m)[p]$ may be zero for three reasons: (1) there are no messages m' for which $p \in \mathit{Depend}(m')$; (2) for all such messages $|\mathit{Depend}(m')| > f$; (3) for all such messages p knows that $q \in \mathit{Log}(m')$. If the first reason holds for all zero elements, then $\mathit{PBC}(m)$ is a dependency vector, and if either the first or second reason hold for all zero elements, then $\mathit{PBC}(m)$ is a weak dependency vector.

Dependency tracking proceeds as follows. Each process $p \in \mathcal{N}$ maintains an $n \times n$ *dependency matrix* $DMat_p$, defined as follows [3]:

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

for $q \in (\mathcal{N} - \{p\})$ and where $DMat_p[i, *]$ denotes the ith row of matrix $DMat_p$.

A process p's estimate of the weak dependency vector of another process q will lag behind q's actual weak dependency vector, and so $DMat_p[q, *]$ will not in general be able to satisfy Condition 8.a. However, it is straightforward to design update rules that satisfy Condition 8.b. Here is one such set of rules:

1. When process p receives a message m from q:
   (a) p generates $\#m$. To do so, it increments $DMat_p[p, p]$ by one. $DMat_p[p, p]$ is now the value of the receive sequence number of m. This is the vector clock update rule used when a process executes a relevant event.
   (b) p sets $DMat_p[p, *]$ to the component-wise maximum of the current value of $DMat_p[p, *]$ and $PBC(m)$. This is the vector clock update rule used when a process receives a piggybacked vector clock.
   Even though $PBC(m)$ is weaker than a weak dependency vector, the resulting value of $DMat_p[p, *]$ is still a weak dependency vector. As noted above, a component $PBC(m)[q]$ can be zero for three different reasons. If one of the first two reasons hold, then for that component $PBC(m)$ is a weak dependency vector. If the third reason holds, then q has already piggybacked the non-zero value of this component that would make it a weak dependency vector.
   (c) p sets $DMat_p[q, *]$ to be the component-wise maximum of the current value of $DMat_p[q, *]$ and $PBC(m)$. Doing so ensures that p's estimate of q's dependency vector is up to date. As above, even though $PBC(m)$ is not a weak dependency vector, the resulting value of $DMat_p[q, *]$ is a weak dependency vector.
   (d) For all values of $i : 1 \le i \le n$, p sets $DMat_p[i, i]$ to the maximum of $DMat_p[i, i]$ and $PBC(m)[i]$. This is done because p may learn about some process i reaching a state interval indirectly from q rather than directly from i.
2. When process q receives an acknowledgment for message m from p, it sets $DMat_q[p, *]$ to be the component-wise maximum of the current value of $DMat_q[p, *]$ and $PBC(m)$.

Given Condition 8.b, it is simple for p to estimate $Depend(m)$ and therefore $Log(m)$: $Log(m)_p$ contains the processes q such that $DMat_p[q, m.dest]$ is at least $m.rsn$. And, process p can consider $\#m$ to be stable when more than f entries of $DMat_p[*, m.dest]$ are greater than or equal to $m.rsn$.

The set of rules given above implements Protocol Π_{Det}. In the next section we describe three more sets of rules that implement Π^+_{Det}, $\Pi^+_{|Log|}$ and Π^+_{Log}.

[3] 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 [4].

4.4 Piggybacking the Dependency Matrix

As it turns out, it is simpler to present the set of rules that implement Π^{+}_{Det}, $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} by starting from the last protocol and working our way backwards to the first. The reason lies in the observation that the dependency matrix of process q can be used to compute $Log(m)_q$ for all messages m for which q is a member of $Depend(m)$. So, to implement Π^{+}_{Log}, q can simply piggyback its dependency matrix on every message it sends.

4.4.1 Implementing Π^{+}_{Log}

The update rules for Π^{+}_{Log} are as follows:

1. When process p receives a message m from q:

   (a) p generates $\#m$. To do so, it increments $DMat_p[p,p]$ by one. $DMat_p[p,p]$ is now the value of the receive sequence number of m.

   (b) p sets $DMat_p[p,*]$ to the component-wise maximum of the current value of $DMat_p[p,*]$ and $DMat_q[q,*]$.

   (c) For all values of $i : 1 \leq i \leq n$, p sets $DMat_p[q,*]$ to the component-wise maximum of the current value of $DMat_p[i,*]$ and the piggybacked $DMat_q[i,*]$.

2. When process q receives an acknowledgment for message m from p, it sets $DMat_q[p,*]$ to be the component-wise maximum of the current value of $DMat_q[p,*]$ and $PBC(m)$.

The resulting protocol implements Π^{+}_{Log} and piggybacks n^2 additional data over Π_{Det}, which is independent of the number of determinants in both DL_p and $UnstableDL_p$.

4.4.2 Implementing $\Pi^{+}_{|Log|}$

A second set of update rules can be used to derive an implementation of $\Pi^{+}_{|Log|}$ that is analogous to $\Pi_{|Log|}$ and that piggybacks $O(f \times n)$ additional data per message. Consider the following data structure that is extracted from the dependency matrix:

Stability Matrix: $SMat_p$ is a $(f+1) \times n$ matrix of integers. For all processes q in $\mathcal{N}$, $SMat_p[i,q]$ is the highest receive sequence number of any message m delivered by q for which $|Log(m)|_p = i$.

The stability matrix is a compact way of representing $|Log(m)|_p$. Specifically,

$$|Log(m)|_p = \begin{cases} \text{at least } f+1 & \text{when } m.rsn \leq SMat_p[f, m.dest] \\ i : 1 \leq i \leq f & \text{when } SMat_p[i+1, m.dest] < m.rsn \leq SMat_p[i, m.dest] \end{cases}$$

The stability matrix can be computed directly from the dependency matrix. Consider the column $DMat_p[*,q]$. The values in this column are a multi-set[4] of receive sequence numbers for messages that were delivered by q. Let ℓ be the first largest value in this multi-set. ℓ is also the receive sequence number of the last message that p knows q has delivered. Thus, for all messages m

[4]A *multi-set* S is a set in which the same value may occur more than once. The kth *largest* value in S is defined recursively as follows: the first largest value in S is the largest value that occurs in S, and the kth largest value in S is the $(k-1)^{st}$ largest value of the multi-set of S with the first largest value removed. Thus, the first and second largest values of $\{2,1,2\}$ are both 2, and the third largest value is 1.

delivered by q, if if $m.rsn \leq \ell$ then $|Log(m)|_p \geq 1$ and if $m.rsn > \ell$ then $|Log(m)|_p = 0$. Thus, $SMat_p[1, q] = \ell$. Generalizing this observation, $SMat_p[i, q]$ is the ith largest value of $DMat_p[*, q]$.

In protocol $\Pi^+_{|Log|}$, all processes piggyback their stability matrix instead of their dependency matrix. Doing so allows a process p to compute a more accurate value of $SMat_p$. The set of update rules is:

1. When process p receives a message m from q:

   (a) p generates $\#m$. To do so, it increments $DMat_p[p, p]$ by one. $DMat_p[p, p]$ is now the value of the receive sequence number of m.

   (b) Consider a determinant $\#m'$ piggybacked on m. If $m'.rsn > DMat_p[p, m'.dest]$ then p is receiving $\#m'$ for the first time. Call such a determinant *new to p*. For any determinant $\#m'$ new to p, $p \notin Log(m')_q$ and so $|Log(m')|_p$ is set to $|Log(m')|_q{+}1$. Process p computes a new value $SMat_q{}'$ of $SMat_q$ that reflects this fact. Specifically, p first sets $SMat_q{}'$ to $SMat_q$. Then, for each determinant $\#m'$ new to p, let s be $|Log(m')|_q$ as computed from the piggybacked stability matrix, i.e., $SMat_q[s{+}1, m'.dest] < m'.rsn \leq SMat_q[s, m'.dest]$. Process p sets $SMat_q{}'[s+1, m'.dest]$ to $\max(SMat_q{}'[s+1, m'.dest], m'.rsn)$.

   (c) p sets $DMat_p[p, *]$ to be the maximum of the current value of $DMat_p[p, *]$ and $PBC(m)$.

   (d) p sets $DMat_p[q, *]$ to the component-wise maximum of the current value of $DMat_p[q, *]$ and $PBC(m)$.

   (e) For all values of $i : 1 \leq i \leq n$, p sets $DMat_p[q, i]$ to be the component-wise maximum of the current value of $DMat_p[q, i]$ and $PBC(m)[i]$. This brings p's estimate of q's weak dependency vector up to date.

   (f) For all values of $i : 1 \leq i \leq n$, p sets $DMat_p[i, i]$ to the maximum value of $DMat_p[i, *]$.

   (g) p sets $SMat_p[i, q]$ to the larger of its current value and of the ith largest value of $DMat_p[r, q]$ for all $r \in \mathcal{N}$. This is the rule given above for generating a stability matrix from a dependency matrix.

   (h) Finally, for all values of $i : 1 \leq i \leq n$, p sets $SMat_p[i, *]$ to the component-wise maximum of the current value of $SMat_p[i, *]$ and the modified version of the piggybacked $SMat_q{}'[i, *]$ obtained from rule (b).

2. When process q receives an acknowledgment for message m from p, it sets $DMat_q[p, *]$ to be the component-wise maximum of the current value of $DMat_q[p, *]$ and $PBC(m)$.

4.4.3 Implementing Π^+_{Det}

Protocol Π^+_{Det} requires process p to inform q of which determinants have become stable. Recall that for each process j, $SMat_p[f + 1, j]$ is the highest receive sequence number of any message delivered by j that p knows to be stable (i.e. that p knows to have been logged by at least $f + 1$ processes). Hence, p can fulfill its requirement simply by piggybacking row $f + 1$ of its stability matrix on the messages it sends to q. We call the vector corresponding to $SMat_p[f + 1, *]$ process p's *stability vector*, or SV_p.

In addition to the steps (a)—(d) of Π_{Det}, in protocol Π^+_{Det} a process p that receives a message m from q takes the following steps:

(e) For all values of $i : 1 \leq i \leq n$, p sets $SV_p[i]$ to the $f + 1$st largest value in $DMat_p[*, i]$, the i-th column of p's dependency matrix.

(f) For all values of $i : 1 \leq i \leq n$, p sets $SV_p[i]$ to the component-wise maximum of the current value of $SV_p[i]$ and the piggybacked $SV_q[i]$.

Π^+_{Det}'s management of acknowledgments is identical to that of Π_{Det}.

Π^+_{Det} uses the stability vector to get a more accurate estimate of which determinants should be part of $UnstableDL_p$. For a determinant $\#m$ to belong to $UnstableDL_p$, both of the following conditions must now hold: (1) f or fewer entries of $DMat_p[q, m.dest]$ are greater that $m.rsn$ (just as in Π_{Det}), and (2) $m.rsn > SV_p[m.dest]$.

5 Comparing Piggyback Overheads

We start our comparison of the different protocols by examining their asymptotic piggyback overheads. These bounds are expressed in terms of the number D of determinants piggybacked on the message and the size w of a determinant. These two values are not independent: w must be larger than the log of D. In the worst case D can be as large as the number of receive events any process can execute. Additionally, w must be larger than $log(n)$ since the determinant encodes the source and destination of a message, but it is not hard to imagine runs in which D is much larger than n. (In any real implementation, w is most likely a constant, such as 64 bits.)

- Π_{Det}: only the determinants are added, and so the overhead is $O(Dw)$.
- $\Pi_{|Log|}$: with each piggybacked determinant for some message m' the estimate $|Log(m')|$ is included. Since we can express this estimate in $\log(f)$ bits, the overhead is $O(D(w + \log(f)))$.
- Π_{Log}: with each piggybacked determinant for some message m' the estimate $Log(m')$ is included. This estimate cannot include more than f process ids, and so the overhead is $O(D(w + f \log(n))$.
- Π^+_{Det}: the stability vector is piggybacked on each message. A stability vector contains n elements, where each element is a receive sequence number. If we use w bits to represent a receive sequence number, then the overhead is $O((D + n)w)$.
- $\Pi^+_{|Log|}$: the stability matrix is piggybacked on each message. Again, if we use w bits to represent a receive sequence number, then the overhead is $O((D + nf)w)$.
- Π^+_{Log}: the dependency matrix is piggybacked on each message. Again, if we use w bits to represent a receive sequence number, then the overhead is $O((D + n^2)w)$.

At this level of abstraction one might tempted to conclude, for example, that $\Pi_{|Log|}$ should be a better choice than Π_{Det} because the former tracks causality better while piggybacking only a logarithmic number of bits more per determinant than the latter. And, given that D can be huge, the last three protocols appear attractive because the additional number of bits used to increase the precision of causal tracking over than of Π_{Det} is independent of D. Whether these observations hold in practice, though, depends strongly on the communication pattern exhibited by the application.

To understand the relative performance of the different protocols, we developed a synthetic application model that we call the *BBL application model.* This model specifies how bursty communication is (*burstiness*), what percentage of the total number of processes process communicates with (*branchiness*), and how slowly acknowledgments return (*latency*). We construct synthetic applications for different combinations of these three parameters. For each constructed application, we measure the piggyback overhead for each protocol for different values of f.

We then construct three other synthetic applications not within the BBL model and again measure the piggyback overhead for the FBL protocols. These three applications have communication structures that resemble specific system structures.

5.1 The BBL Model

The BBL communication model is similar to other models that have been proposed (for example, [3, 7, 21]). The model assumes that processes do not crash and that channels are reliable and maintain FIFO ordering. Each process alternates between two stages of operation: a *communication stage* during which the process sends messages, and a *computation stage* during which the process receives and acknowledges messages. During any communication stage a process never sends more than one message to any other process. The processes to which process p sends messages in a run are called the *neighbors* of p for that run.

The model is parameterized by the five-tuple $\langle n, M, \overline{bu}, \overline{br}, \overline{l}\rangle$ where n is the number of processes in the system and M is the total number of messages sent in the system. The value of $\overline{br}$ determines the size of the set of neighbors. At the beginning of each run, each process p is assigned a random set of neighbors. This size of this set is pulled from a restricted uniform distribution $n \times U(\overline{br})$.[5] For example, if the random variable $\overline{br} = 0.1$, then on average each process will have as neighbors 10% of the remaining processes.

The value of $\overline{bu}$ determines the number of messages a process sends in each communication stage. Specifically, let $bu_{p,i}$ be a random variable that indicates the fraction of neighbors to which process p sends messages during the ith communication stage. The value of $bu_{p,i}$ is pulled from the restricted uniform distribution $U(\overline{bu})$. For example, if $\overline{br} = 0.1$ and $\overline{bu} = 0.5$, then on average each process will send messages to 5% of the other processes during each communications stage. The message recipients are selected randomly without replacement from the process' neighbors.

The value of $\overline{l}$ models the speed of the underlying communication system. This parameter determines how quickly, on average, acknowledgment are received by the sender. The time is measured in terms of the number of events the sender executes between sending the message and receiving the acknowledgment. Specifically, let $l_{p,i}$ be a random variable that determines the number of events processed by p before it receives the acknowledgment for the ith message that it sent. The value of this random variable is pulled from the restricted uniform distribution $\lfloor 2n * U(\overline{l})\rfloor$.

Consider a point in the five-dimensional space that has coordinates $n, M, \overline{bu}, \overline{br}$, and $\overline{l}$. Let a *communication graph* be a run of a synthetic application, represented as a partial ordering of events of the n processes and generated stochastically from a distribution defined by the tuple $\langle n, M, \overline{bu}, \overline{br}, \overline{l}\rangle$. By generating many communication graphs for different points in this space, we can evaluate the performance of the message-logging protocols as a function of the parameters of the model.

We fixed the number of processes n at 10 and the number of messages M at 500. We found that larger values of M did not significantly change our evaluation. Thus, the space is reduced to

[5]The restricted uniform distribution $U(m)$ is a uniform distribution that has an expected value of m and a maximum value of $2m$:

$m = 0.5$:	the uniform distribution from 0 to 1
$0 < m < 0.5$:	the uniform distribution from 0 to $2m$
$0.5 < m < 1$:	the uniform distribution from $2m - 1$ to 1

a 3-dimensional subspace of the original model with axes $\overline{bu}$, $\overline{br}$, and $\overline{l}$, whose values range from 0 to 1. We examine the 64 points $(.2, .4, .6, .8) \times (.2, .4, .6, .8) \times (.2, .4, .6, .8)$ in this subspace.

We generated 21 communication graphs for each of these 64 points and ran each of the six causal logging protocols with four values of $f \in \{2, 3, 4, 9\}$. This resulted in over 32,000 runs. The performance of the protocols at each point in the application space was averaged over the 21 communication graphs. The results presented are accurate to 95% confidence. We use small values of f since for real systems of ten processes the probability of having more than a few failures at any time is very small. We include $f = 9$ since this allows recovery from total failures.

Table 1 summarizes the parameters of the BBL model.

parameter	meaning	values used
n	number of processes	10
M	number of messages sent in run	500
$\overline{bu}$	burstiness of communication	0.2, 0.4, 0.6, 0.8
$\overline{br}$	branchiness of communication	0.2, 0.4, 0.6, 0.8
$\overline{l}$	communication latency	0.2, 0.4, 0.6, 0.8

Table 1: Parameters of the BBL model

5.1.1 Exploring the BBL Space

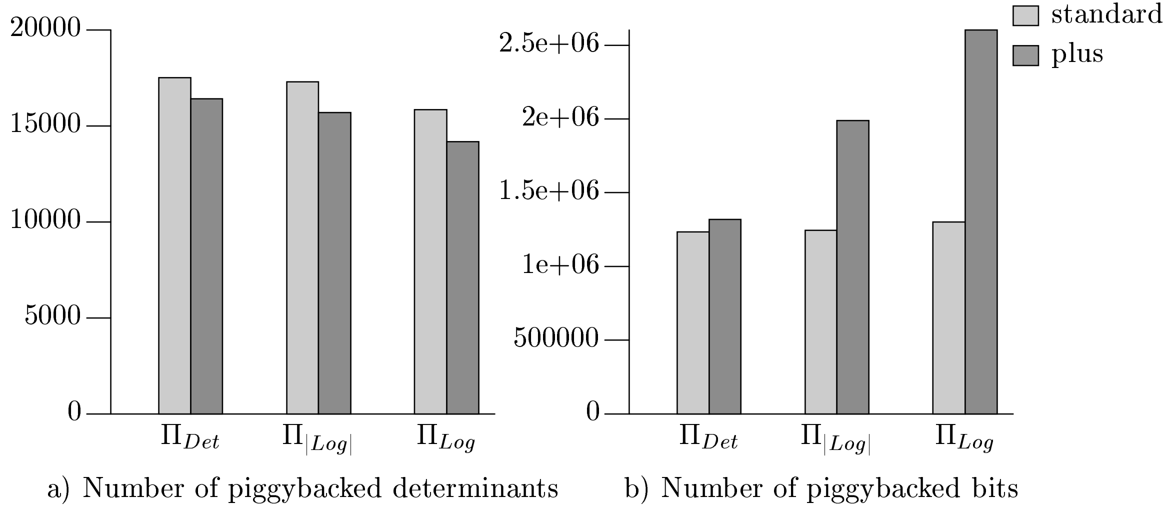
Π^+_{Det}, $\Pi^+_{|Log|}$, and Π^+_{Log} augment Π_{Det}, $\Pi_{|Log|}$, and Π_{Log} by sending information about stable determinants. Two questions regarding these protocols are:

1. How much does information about stable determinants reduce the number of piggybacked determinants?

2. Does this reduction in determinants, if any, lead to a reduction in the overall number of bits piggybacked?

Figure 5 shows summary statistics for each protocol averaged over the sampled application space. The first graph shows the average number of determinants piggybacked over the course of the run. Protocols Π^+_{Det}, $\Pi^+_{|Log|}$ and Π^+_{Log} send 6.3%, 9.1% and 10.6% fewer determinants respectively than the corresponding standard protocol. This shows that the extra information that these protocols send is useful. However, as Figure 5.b shows, the cost of sending this extra information can far exceed the benefit. Protocol Π^+_{Det} sends 6.9% more bits than protocol Π_{Det}. Protocols $\Pi^+_{|Log|}$ and Π^+_{Log} send 59.8% and 100.1% more bits than their corresponding standard protocol.

Recall that protocols Π^+_{Det}, $\Pi^+_{|Log|}$, and Π^+_{Log} add fixed sized data structures to each message. For Π^+_{Det} this data structure is a vector of size n, for $\Pi^+_{|Log|}$ a matrix of size $f \times n$, and for Π^+_{Log} a matrix of size n^2. Assuming $n = 10$ and 32 bit words, this overhead is between 320 and 3,200 bits. Since 500 messages are sent in a run, the accumulated overhead is between 160,000 and 1,600,000 bits. The latter value, which is the overhead of Π^+_{Log}, accounts for 61.5% of the average number of bits sent. This overhead is directly related to the $O(n)$ size of vector clocks, which has been shown to be a lower bound [20]. So, at least for the part of the BBL space that we consider, $\Pi^+_{|Log|}$ and Π^+_{Log} are not competitive.



a) Number of piggybacked determinants　　b) Number of piggybacked bits

Figure 5: A comparison of the performance of the standard vs. "plus"' protocols

To understand the relative performance of the protocols we compared them as follows: for each point sampled in the BBL space, we counted the number of times one protocol significantly outperformed the other. One protocol significantly outperformed the other when it piggybacked on average fewer bits and the 95% confidence intervals did not overlap. Table 2 shows the pairwise comparison of the six protocols. The values in the table represent the number of points in the BBL space where the protocol in the column outperformed the protocol in the row. For example, the value of 43 in the first column, second row, is the number of points at which protocol Π_{Det} piggybacked on average significantly fewer bit than protocol Π^+_{Det}.

	Π_{Det}	Π^+_{Det}	$\Pi_{\vert Log\vert}$	$\Pi^+_{\vert Log\vert}$	Π_{Log}	Π^+_{Log}
Π_{Det}	-	0	0	0	0	0
Π^+_{Det}	43	-	25	0	25	0
$\Pi_{\vert Log\vert}$	0	0	-	0	0	0
$\Pi^+_{\vert Log\vert}$	256	256	256	-	256	24
Π_{Log}	59	20	56	0	-	0
Π^+_{Log}	256	256	256	192	256	-

Table 2: Pairwise comparison of the relative performance of the protocols as measured by the total number of bits piggybacked.

These results again show that $\Pi^+_{|Log|}$ and Π^+_{Log} are not competitive with the other protocols. (The table also shows that, over most of the space, $\Pi^+_{|Log|}$ outperforms Π^+_{Log}. We expected this for small f but not for $f = 9$). Because they are not competitive, we exclude protocols $\Pi^+_{|Log|}$ and Π^+_{Log} from the rest of the discussion and concentrate on the performance of the four remaining protocols.

Protocol Π_{Det} Table 2 shows that over all the points sampled, no protocol ever piggybacks significantly fewer bits than Π_{Det}. This suggests that if no knowledge about the application's characteristic is known then Π_{Det} is a good choice. We therefore use Π_{Det} as our baseline protocol.

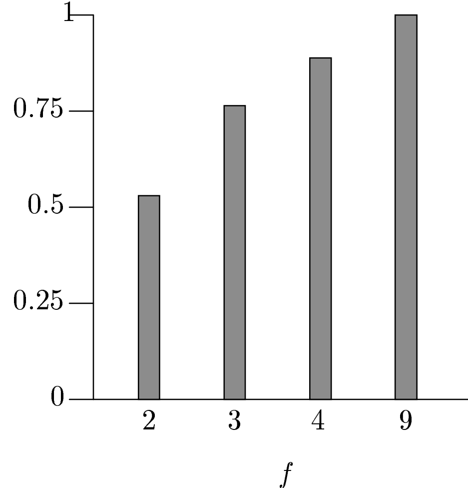


Figure 6: Piggyback overhead of Π_{Det} as function of f, normalized to Manetho's piggyback overhead.

The regression equation for the number of bits piggybacked by Π_{Det} is:[6]

$$\textit{number of bits} = 237,000\ \overline{bu} + 481,100\ \overline{br} - 4942\ \overline{l} + 860,100\ (f/10) + C_{det}.$$

The R-Squared significance test of this regression of 0.63.

This regression equation suggests that the performance of Π_{Det} is dominated by the value of f for the protocol. Recall that Manetho is essentially Π_{Det} instantiated with $f = n$. Since there is no difference in the number of piggybacked bits for $f = n - 1$ and $f = n$,[7] Π_{Det} piggybacks for $f = 9$ the same number of bits as Manetho. However, given the high sensitivity to f, Π_{Det} appears to piggyback much fewer bits than Manetho when f is small.

Figure 6 examines this issue. In this figure, we compare the number of piggybacked bits of Π_{Det} and Manetho as a function of f. The figure shows that for $f = 2$, protocol Π_{Det} sends 47% fewer bits than Manetho. These performance gains decrease as f increases. As we show in Section 5.2, this is an artifact of n being relatively small: with Π_{Det}'s relative inaccuracy in tracking causality, it does not take long for a determinant to be piggybacked to a substantial fraction of the ten processes.

The regression equation also shows that Π_{Det} is relatively insensitive to the latency of the underlying communication system, and moderately sensitive to the size of the communication neighborhood and the burst frequency. This makes sense intuitively. When the neighborhood size is small, processes send more messages to the same recipients, resulting in tighter synchronization among them. Once a process has sent a determinant to its neighbor, it never needs to send it to the same neighbor again. When the size of the neighborhood is small, the neighborhood quickly becomes saturated with the determinant. The reasoning for the sensitivity to $\overline{bu}$ is similar. When this parameter is high, processes broadcast messages to a high percentage of their neighbors, therefore saturating their neighborhood.

[6] We write the equation in terms of $(f/10)$ rather than f so that the ranges of all of the independent variables are between 0 and 1.

[7] In both cases, a process p piggybacks a determinant $\#m$ to q when $q \notin Log(m)_p$.

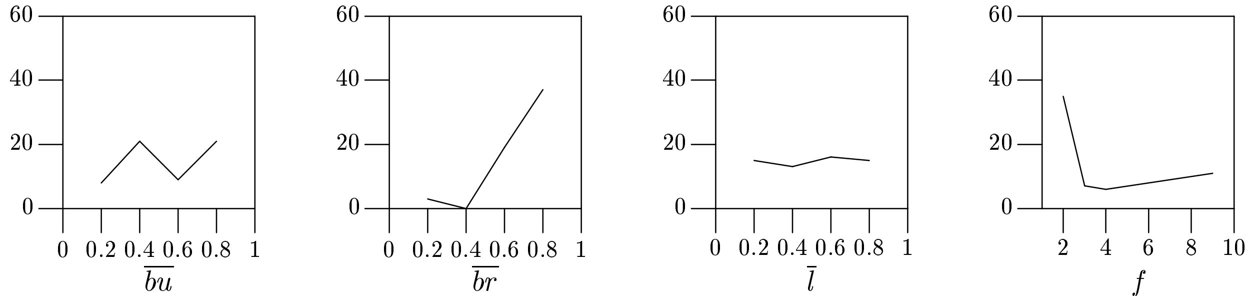


Figure 7: Performance of Π_{Log}

Protocol $\Pi_{|Log|}$ The performance of protocol $\Pi_{|Log|}$ is statistically indistinguishable from our baseline Π_{Det}. The extra integer per determinant piggybacked in $\Pi_{|Log|}$ can reduce the number of piggybacked determinants when the communication graph has long linear paths. In the sampled applications there are few linear paths since each process sends messages in each round. Overall, $\Pi_{|Log|}$ is able to send only 1.2% fewer determinants than Π_{Det}, not enough to reduce the extra cost associated with this protocol.

One might, in fact, argue from Table 2 that Π_{Det} is slightly better than $\Pi_{|Log|}$, since there are 59 points where Π_{Det} significantly outperforms Π_{Log} and only 56 points where $\Pi_{|Log|}$ significantly outperforms Π_{Log}. Similarly, Π_{Det} does better more often in comparison to Π^{+}_{Det} than $\Pi_{|Log|}$.

Protocol Π_{Log} Figure 5 shows that overall, the extra information carried by Π_{Log} reduces the number of piggybacked determinants by over 10% as compared with Π_{Det}. Looking at the pairwise comparison between Π_{Log} and Π_{Det}, we see that at 197 points Π_{Log} is statistically indistinguishable from Π_{Det}, and at 59 points Π_{Log} performs significantly worse than Π_{Det}.

Figure 7 shows the breakdown of the 59 points where Π_{Log} performs poorly as a function of f, $\overline{br}$, $\overline{bu}$ and latency. This figure shows that the performance of Π_{Log} as compared with Π_{Det} is linearly correlated with $\overline{br}$. As $\overline{br}$ increases, there are more cases where Π_{Log} performs poorly.

In addition, Π_{Log} is also affected by the value of f. We sampled 64 points for $f = 2$, and for 35 of these Π_{Log} performs worse than Π_{Det}.

Figure 7 also shows that the performance of Π_{Log} is independent of the latency, and indeterminate with respect to $\overline{bu}$.

Protocol Π^{+}_{Det} Overall, Π^{+}_{Det} is indistinguishable from Π_{Det} over most of the runs. In 43 of the 256 points, Π^{+}_{Det} does significantly worse than Π_{Det}. Figure 8 shows how the relative performance varies as a function of each of the dimensions.

Like protocol Π_{Log}, Π^{+}_{Det} is inversely sensitive to $\overline{br}$. Unlike Π_{Log}, Π^{+}_{Det} seems to perform relatively better for low values of f.

5.2 The Client/Server Model

The BBL model has the processes communicate asynchronously in a bursty manner. While this is not unusual for many scientific applications, many other applications are more synchronous in their communications. Hence, we construct three additional synthetic applications. Each application uses 40 processes.

CS1 This application has a client-server-like communication structure. A process is chosen at random without replacement from the 40 processes. The chosen process sends a message to

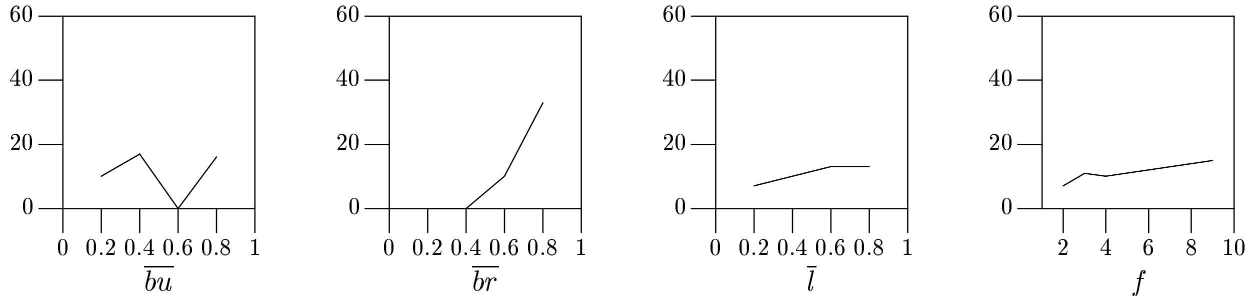


Figure 8: Performance of Π^+_{Det}

another process, again chosen randomly without replacement. The message chain continues until 20 processes are selected. The twentieth process sends a reply message to the nineteenth process. This reply chain continues until the first process receives a reply message. This process of generating request and reply chains of depth 20 is repeated 20 times.

CS3 This application also has a client-server-like communication structure. Instead of generating chains of length 20, though, this application generates ternary trees of depth four (and hence, containing 40 processes). A non-leaf process sends three messages, one each to three processes chosen at random without replacement. A leaf process immediately sends a reply to its parent, and a non-leaf process sends a reply to its parent once it receives the three replies from its children. The application generates 20 of these trees.

SG This application has a group-based communication structure. A process is chosen at random without replacement. The chosen process selects eight processes and sends each of them a message without waiting for acknowledgments; thus, a degree-eight tree of depth one is constructed. Each process sends a reply to the original process. When the original process receives the eight replies, a new tree with a randomly-chosen process is constructed. The application generates 20 of these trees.

All three applications repeatedly generate trees, which are trivially shortcut-free. Given this simple pattern, one might be tempted to conclude that Π_{Det} would be the best protocol. However, as Figures 9 and 10 show, Π_{Log} performs significantly better than the other protocols for all but the smallest values of f. Each figure shows the piggyback overhead of the four protocols Π_{Det}, $\Pi_{|Log|}$, Π_{Log} and Π^+_{Det} as a function of f for $f \in \{2, 3, 10, 20, 30, 40\}$.

The reason for this behavior is that the communication graph is in fact not a tree: a process p that receives a message from q in one iteration may in another iteration send a message to q. A determinant may follow a very complex path, which, as we saw in the BBL model, is a situation for which Π_{Log} performs well. In addition, Manetho performs relatively poorly both overall and in comparison with Π_{Det}. It is not until $f = 20$ that Π_{Det} effectively piggybacks determinants to all processes.

For SG, however, Π_{Log} (and Π^+_{Det}) do poorly. $\Pi_{|Log|}$ and Π_{Det} have similar piggyback loads, with Π_{Det} edging out $\Pi_{|Log|}$ for larger values of f. In fact, by $f = 10$ the piggyback load for all protocols has reached 80% of their piggyback load for $f = n$. In SG, the process at the root of each tree quickly learns that the determinants it piggybacks are logged in at least nine different processes. For smaller values of f, the additional information provided by $\Pi_{|Log|}$ is helpful in spreading the fact that these determinant are stable, but for larger values of f determinants spread quickly around the system, in which case Π_{Det} does best.

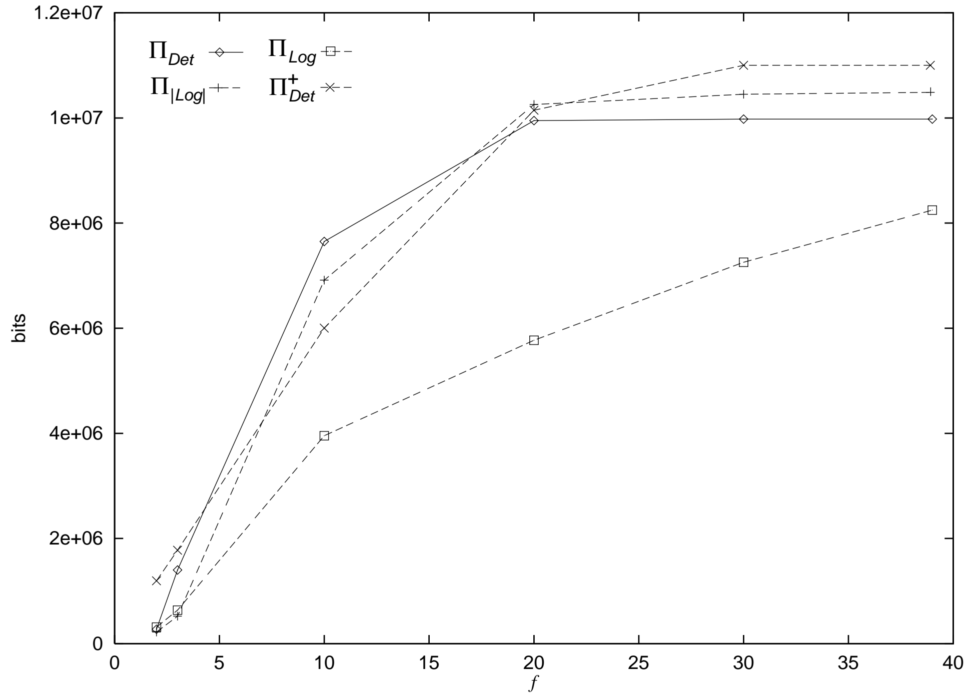


Figure 9: Piggyback overhead for CS1 as a function of f.

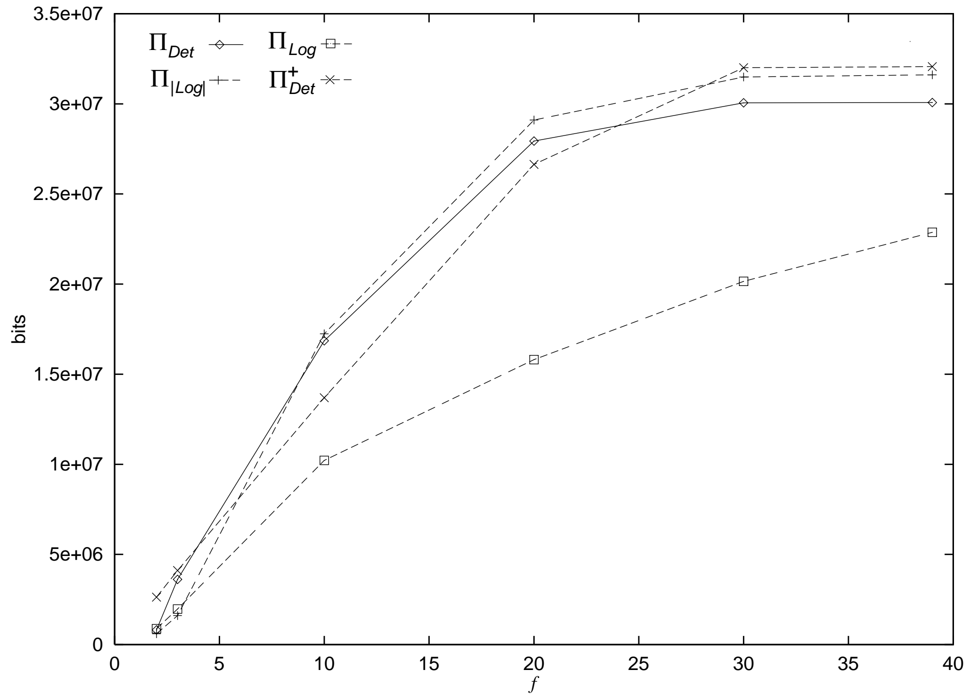


Figure 10: Piggyback overhead for CS3 as a function of f.

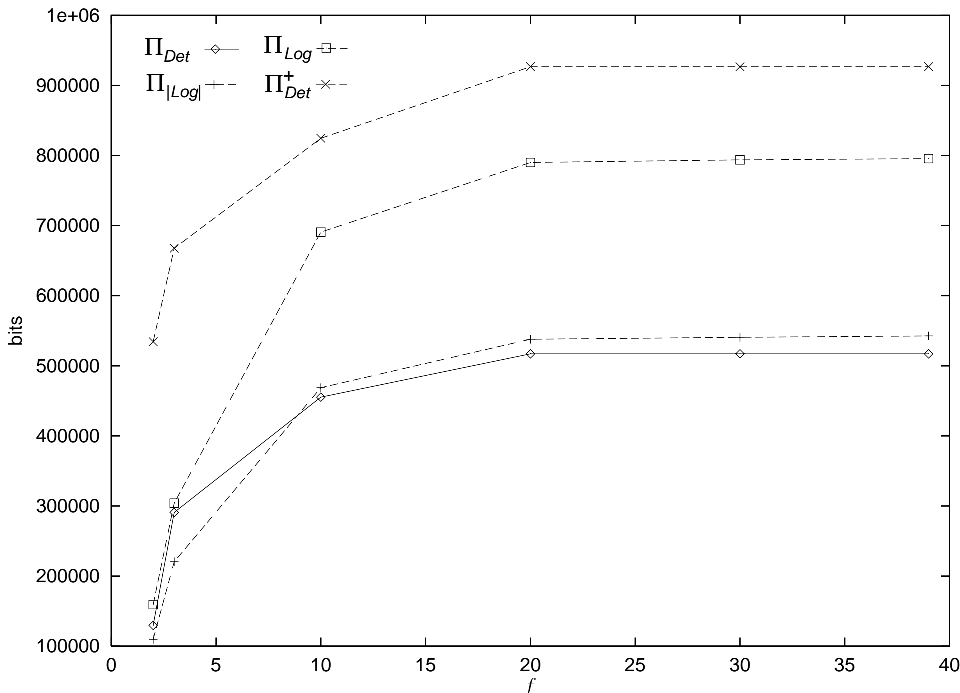


Figure 11: Piggyback overhead for SG as a function of f.

5.3 Discussion

The results of the simulations are specific to the application models in which they were run. Here, we give some general intuition that could help application programmers choose among the protocols.

The "plus" protocols are theoretically attractive because they convey so much information using a representation whose size is independent from the number of determinants piggybacked on a message. The results from our simulations indicate, though, that there are no situations where $\Pi^+_{|Log|}$ and Π^+_{Log} are appropriate.

Protocol $\Pi_{|Log|}$ performs very similarly to Π_{Det}, and is somewhat better when the average fanout of messages is low and f is small. However, if it is known that the application fanout is low, then Π_{Log} is a more logical choice since it does much better in this case and is less sensitive to f.

These recommendations would most likely change for larger values of n and for other patterns of communications. Another issue worth studying is how the results change with the frequency of checkpointing. The more frequent checkpointing occurs, the more determinants can become stable via checkpointing. Frequent checkpointing, though, imposes an overhead both on storage and on computation.

6 Conclusions

In causal message logging protocols, each process tracks causality to estimate both the number and the identities of processes that store a copy of a determinant. We have shown that the tradeoff between excess piggybacking due to inaccurate causality tracking and the extra piggybacked information to increase the accuracy of causality tracking is both complex and application specific.

We have given some situations in which the simplest of the FBL protocols is the best choice with respect to piggyback overhead, and then given some heuristics for when to use other protocols. The choice almost always comes down between the simplest protocol, Π_{Det}, and one of the more accurate protocols, Π_{Log}.

The piggyback overhead of causal logging can become large, and so understanding how to reduce the piggyback overhead is important. Further reduction can be accomplished by compressing the information that is piggybacked (see, for example, [3]). We don't believe that such compression would change the relative rankings we have found for the various FBL protocols. If there is considerable locality in the communications patterns, though, then large parts of the dependency matrix may not change very frequently, and so compression of the dependency matrix based on difference encoding might make Π^{+}_{Log} competitive. This question (and the related one concerning compression of the stability matrix) would be best explored by considering real, rather than synthetic, applications.

Piggyback overhead is not the only metric with which one could compare the different FBL protocols. For example, the overhead of processing individual determinants may make protocols like Π^{+}_{Log} advantageous. A more detailed comparison, however, would most likely depend on very specific environmental factors, such as the relative processor speed with respect to the communication bandwidth and the overhead (and hence the frequency) of checkpointing.

Putting these results in a broader context, causal message logging protocols are related to causal multicast [25] which in turn are related to global state detection [1, 14]. All of these protocols track causal dependencies to implement some level of distributed knowledge about the execution history of some application. For example, in [4] we showed how a causal message logging protocol can be derived starting from causal multicast. And, if $f = n$ then casual message logging ensures that each process has stored locally all of the nondeterministic choices made in the causal past of that process. A simple extension to causal message logging would allow a process to have locally all "important" events in its causal past available for debugging or for global state detection purposes. Hence, the tradeoffs explored in this paper should be useful to those studying other protocols that build upon causality tracking.

The simulator and the data we generated for the analysis in this paper is available from the authors upon request.

Acknowledgments We thank Bruce Hoppe and Fred Schneider for their help in refining our ideas and both Wanda Chiu and Alessandro Amoroso for their detailed comments on our work. We also thank Elmootazbellah N. Elnozahy for his comments on an earlier draft of this paper and for helping us with implementation details of Manetho. Finally, we thank the anonymous referees for their careful reading and their constructive criticism of the original manuscript.

References

[1] S. Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *Information Processing Letters*, 50:311–316, June 1994.

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

[3] L. Alvisi, B. Hoppe, and K. Marzullo. Nonblocking and Orphan-Free Message Logging Protocols. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, June 1993.

[4] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[5] A. 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.

[6] 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.

[7] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a Communications Characterization Methodology for Parallel Applications. In *Proceedings Third International Symposium on High-Performance Computer Architecture*, pages 310–319, February 1997.

[8] E. N. Elnozahy. Personal communication, 12 September 1997.

[9] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, 1999.

[10] 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.

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

[12] D.B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] K. Marzullo and G. Neiger. Detection of global state predicates. In *Proceedings of Fifth International Conference on Distributed Algorithms*, pages 257–272, October 1991.

[15] D. Massonet and F. Adragna. Synthetic Aperture Radar: New Processing Concepts. In *Proceedings of the 10th International Geoscience and Remote Sensing Symposium*, pages 1323–1326, May 1990.

[16] F. Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[17] 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.

[18] S. Rao, L. Alvisi, and H. Vin. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, March/April 2000.

[19] F. B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[20] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[21] J. P. Singh, E. Rothberg, and A. Gupta. Modelling Communications in Parallel Algorithms: a Fruitful Interaction Between Theory and Systems? In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures*, pages 189–199, June 1994.

[22] 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.

[23] R. B. Strom and S. Yemeni. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.

[24] R. E. Strom, D. F. Bacon, and S. A. Yemini. Volatile Logging in n-Fault-Tolerant Distributed Systems. In *Proceedings of the 18th Annual International Symposium on Fault-Tolerant Computing*, pages 44–49, 1988.

[25] R. van Renesse. Why bother with CATOCS? *Operating Systems Review*, 28(1):22–27, January 1994.

[26] S. Venkatesan and T.Y. Juang. Efficient Algorithms for Optimistic Crash Recovery. *Distributed Computing*, 8(2):105–114, June 1994.