The Relative Overhead of Piggybacking in Causal Message Logging Protocols

Karan Bhatia Keith Marzullo* Lorenzo Alvisi†
Dept. of Computer Science and Eng. Dept. of Computer Sciences
University of California, San Diego University of Texas at Austin

Abstract

Message logging protocols ensure that crashed processes make the same choices when re-executing nondeterministic events during recovery. Causal *message logging protocols achieve this by piggybacking the results of these choices (called* determinants*) on the ambient message traffic. By doing so, these protocols do not create orphan processes nor introduce blocking in failure-free executions. To survive f failures, they ensure that determinants are stored by at least $f+1$ processes.*

Causal logging protocols differ in the kind of information they piggyback to other processes. The more information they send, the better each process is able to estimate global properties of the determinants, which in turn results in fewer needless piggybacking of determinants. This paper attempts to quantify the tradeoff between the cost of sending more information and the benefit of doing so.

Keywords: Fault-tolerance, Message logging, Causal logging, Rollback recovery

1. Introduction

Message logging protocols [5] ensure that crashed processes make the same choices when re-executing nondeterministic events during recovery. *Causal* message logging protocols [2, 6] achieve this by piggybacking the results of these choices (called *determinants*) on the ambient message traffic. By doing so, these protocols do not create orphan processes nor introduce blocking in failure-free executions. To survive f failures, causal logging protocols ensure that determinants are stored by at least $f+1$ processes. Once a determinant is stored at $f+1$ sites, it is *stable* and does not need to be piggybacked further. So, if $C(e)$ is the set of processes that have stored a copy of the determinant of event e, piggybacking can stop as soon as $|C(e)| > f$.

Unfortunately, $C(e)$ is a global variable that can only be estimated locally by each process. Let $C_p(e)$ be process p's estimate of $C(e)$. To ensure the availability of e's determinant during recovery, local estimates must be conservative: $C_p(e) \subseteq C(e)$. To increase the accuracy of $C_p(e)$, more information may need to be piggybacked. For example, suppose in a system of n processes we wish $C_p(e)$ to include all of the processes in any causal path leading from the execution of e to the current state of q. To do so, $O(n^2)$ information must be piggybacked on all messages [7]. This amount of information may dwarf the overhead that would be imposed by simply redundantly piggybacking v_e. In general, there appears to be a tradeoff between the piggyback load used to estimate $C(e)$ with high accuracy and the piggyback load generated by redundant piggybacks of v_e because of an inaccurate estimate of $C(e)$.

*This research was supported by AFOSR grant F49620-96-1-0215.
†Lorenzo Alvisi was supported in part by the National Science Foundation (CAREER award CCR-9734185, Research Infrastructure Award CDA-9734185, and Research Award CCR-9707056).

This paper is a first attempt to study this tradeoff. We define six different techniques for estimating $Log(m)$, the set of processes that hold logging information on message m. These techniques are based on three methods of encoding the estimate (none, as $|Log(m)|$, and as $Log(m)$) and a choice on whether or not to include estimates for $Log(m)$ when $|Log(m)| > f$. In order to restrict our domain, we ignore the effect of checkpointing and we only consider systems that consist of ten processes. We examine causal logging protocols in a three-dimensional space of communication patterns that captures properties of communication networks and application properties.

2. The Protocols

We assume a general asynchronous distributed computing model, in which processes communicate by exchanging point-to-point messages and communication channels are reliable and FIFO. Processes fail independently by crashing.

We also also assume that the only non-determinism in the system is in the order in which messages are delivered. Therefore, for process recovery, it suffices to record the information necessary reproduce the original message delivery order. For each message m, this information consists of (1) the process $m.sender$ that sent m, (2) the ordinal value $m.ssn$ of m, (3) the process $m.receiver$ that delivered m, and (4) the ordinal value $m.rsn$ of m. $m.ssn = \ell$ means that m was the l^{th} message sent by $m.sender$), and $m.rsn = \ell$ means that m was the l^{th} message delivered by $m.receiver$. This information is called the *determinant* of m and denoted $\#m$.

If process p stores $\#m$ in its volatile memory and sends a message m' to process q, p piggybacks $\#m$ on m' if p believes that m may be unstable and p believes that q may not yet have a copy of $\#m$: $|Log(m)|_p \leq f \wedge q \notin Log_p(m)$. It continues to piggyback $\#m$ in this way until $\#m$ is stable.

The simplest of the protocols is P_{det}, in which the sender sends no extra information aside from the unstable determinants. Figure 1 shows a simple run with $f = 3$. The communication graph contains a chain starting from process p and ending at process t. At point A, message m is stable since its determinant is stored at processes $\{q, r, s, t\}$. However, with P_{det}, no process knows that m is stable. Process t at point A can only infer that $Log_t(m) = \{q, s, t\}$. t can infer q from the information in the determinant $\#m$ and s since s sent the message on which $\#m$ is piggybacked.

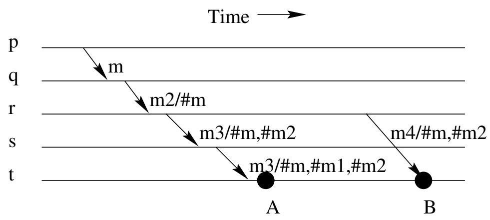

Figure 1. A run with 5 processes $\{p, q, r, s, t\}$.

In addition to the determinants, the sending process can add information to a message to help the receiver make better estimates of the global value of $Log(m)$. This could include it's local estimate of $|Log(m)|_p$ as in protocol $P_{|log|}$, or it's estimate of $Log_p(m)$ as in P_{log}. $P_{|log|}$ adds a cost of $log(f)$ bits per determinant to the piggyback. In the example in Figure 1, using $P_{|log|}$ enables process t to determine that m is stable at point A. However, in general, without the knowledge of the actual processes in $Log(m)$, optimization is difficult. Right before point B in the example, process t estimates that $|Log(m)|_t = 4$ without knowing which four. After point B, process t must act conservatively and assume that process r was already one of the four it knew about. P_{log} adds N bits per determinant but can help greatly to reduce the number of unnecessary piggybacking by adding to the knowledge of the sender without being forced to make conservative estimates.

The protocols described so far only send information about unstable determinants. If a sender knows a determinant is stable, it keeps this knowledge to itself.

We can extend each of the three protocols described to send complementary information about unstable determinants. P_{det} is extended by adding a constant size stability vector per message, not per determinant. $P_{|log|}$ is extended by adding a constant size stability matrix sm to the message that encodes $|Log(m)|_p$ information. And P_{log} is extended to P_{log}^{+} by adding a constant size dependency matrix dm to each message. Each row in dm is a weak dependency vector (or vector clock) as described in [1]. The full causal past is encoded in the dependency matrix and subsumes both the stability matrix and the stability vector.

3. The BBL Communications Model

The relative piggyback overhead of the various protocols depends strongly on the actual communication pattern used. The communication pattern is in turn dependent on the application, the communication library, the network characteristics and the physical hardware. In order to develop some guidelines as to which protocols to use, we use a general parameterized communications model that we call the *BBL Model*. We generate different communications patterns within this parameter space and compare the relative performance of the causal message logging protocols as a function of the coordinates in this space.

The communications model we use is similar to other models that have been proposed (for example, [4]). The model posits n processes communicating through point-to-point messages. Messages are given to some underlying communication system that is reliable and enforces FIFO order. An application is parameterized by five dimensions: n, *message count*, *burstiness*, *branchiness*, and *latency*. The message count is the total number of messages sent in the system, burstiness is a measure of the computation to communication ratio for a process, branchiness is a measure of the spatial distribution of messages, and latency is a measure of the speed of the underlying communication system. The last three quantities are specified as average values.

Each process in this model cycles through two stages of operation: a communication stage during which the process sends out messages, and a computation stage during which messages are received and acknowledged. The burstiness parameter determines the average number of messages sent in each communication stage. Burstiness is a real number between 0 and 1, where the higher the number the more messages are sent, up to a predefined maximum. For each communication stage, the actual number of messages sent is a random variable with a restricted uniform distribution with mean of (burstiness × maximum burstiness). For our experiments, the maximum burstiness was 10.

Message recipients are selected randomly out of a set of *neighbor* processes that are fixed over the run. The size of the neighbors set is determined by the branchiness parameter, which is also a real number between 0 and 1. For each process, the size of its neighbors set is a random variable with a restricted uniform distribution with mean of (branchiness $\times(n-1)$).

The latency parameter determines how fast, on average, an acknowledgment is returned to the sender. Latency is a real number between 0 and 1, where the higher the value the slower the acknowledgment. The delay is measured in terms of the number of communication events that occur before the acknowledgment event. For each message sent, the number of events before the acknowledgment arrives is a random variable with mean (latency × maximum latency). For our experiments, the maximum latency was 20.

The communication graphs for a given set of parameters are generated using a round robin algorithm: starting from a process, one first determines the number of messages to send based on the burstiness. Then, for each message in this burst, a process is chosen at random from the neighbors set. The messages are sent. Then, this procedure is repeated for the next process chosen using the round robin order. This loop continues until the desired number of messages have been sent.

protocol	minimum	maximum	average
P_{det}	809	29,435	10,879
$P_{\lvert log \rvert}$	809	29,435	10,524
P_{log}	809	26,951	8,833
$P^{+}_{\lvert log \rvert}$	795	29,435	8,861
P^{+}_{log}	771	25,270	6,760
P^{+}_{det}	795	29,435	9,407

Table 1. Number of determinants piggybacked for each protocol

protocol	minimum	maximum	average
P_{det}	87,498	2,050,780	778,429
$P_{\lvert log \rvert}$	87,498	2,077,100	759,056
P_{log}	95,592	2,140,040	727,540
$P^{+}_{\lvert log \rvert}$	566,538	3,653,470	1,523,023
P^{+}_{log}	1,684,140	3,637,710	2,101,130
P^{+}_{det}	246,539	2,211,050	838,008

Table 2. Average communication overhead per run for each protocol (in bits)

protocol	f	burstiness	branchiness	latency
$P_{\lvert log \rvert}$	8.9	2.1	6.4	7.8
P_{log}	7.3	2.7	5.7	8.1
P^{+}_{det}	10.3	1.9	4.7	7.0

Table 3. Linear Regression coefficients

4. Exploring the BBL Space

We generated over 600 different BBL communication patterns with varying parameters. Each pattern has ten processes and roughly 500 messages. Branchiness, burstiness, and latency vary with five different values each: 0.1, 0.3, 0.5, 0.7, and 0.9. In addition, at each parameter setting, five different random communication patterns were generated. The word size for these experiments was 32 bits.

Each of the six FBL protocols was run with each communication pattern and with an f value of 2, 3, 4 and 9. Thus, a total of 15,000 runs were executed. We chose 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 included $f = 9$ since this allows to compare our protocols with Manetho [6], a causal protocol that sends the same information as P_{det} and is designed to tolerate total failures.

4.1. The Importance of Being Stable

We first measured the piggybacking load imposed by each of the six protocols. Our results, summarized in Tables 4.1 and 4.1, indicate that information about stable determinants greatly reduces the unnecessary piggybacking. Unfortunately, our experiments also show that the cost of sending this information is very high. The poor performance in terms of bandwidth is understandable since the "plus" protocols send large fixed-size arrays with each message. For P^{+}_{log} , for example, the size of the matrix is $10 \times 10 \times 32 = 3,200$. Protocol P_{log} , however, while sending more information per determinant, is able to make up that handicap and on average, P_{log} has the lowest bandwidth requirements. P^{+}_{det} is somewhat of a surprise, and perhaps represents a compromise. It sends a n length vector to summarize the stability information, yet the total bandwidth is still competitive with the simpler protocols.

4.2. Sensitivity to Parameters

Using the UnixStat statistics package, we performed linear regression analysis on the remaining protocols to determine the effects of different factors on the total communication overhead. The results are summarized in Table 4.2. It should be noted that the coefficients for f were adjusted to reflect the fact that the range of f is 10 times the range of the other parameters. (The R-Squared statistic for the regressions was 0.73, 0.78, and 0.72 for $P_{\lvert log \rvert}$, P_{log} , and P^{+}_{det} respectively, indicating a reasonable fit.)

The regression results indicate that both f and the speed of the underlying communication system have a strong positive influence on the communication overhead. Intuitively this is reasonable. As f increases, more redundancy is needed in the system and therefore more determinants are piggybacked. P^+_{det} is more sensitive to f because as f increases and the amount of redundancy saturates the system, information on which determinants are stable becomes less important.

4.3. Understanding the Relative Performance of the Protocols

To understand the relative strengths and weaknesses of the remaining three protocols as a function of the application properties, we generated *data cubes* from the raw data collected. Each data cube is a unit cube with the three application properties as axis: branchiness, burstiness, and latency. Each point in the cube represents the set of applications with the corresponding properties. The 600 runs that were generated fall onto 125 points forming a $5 \times 5 \times 5$ grid in the data cube. For each grid point, we calculate the average performance of each of the three protocols at that point and use a marker to represent the protocol that uses the least bandwidth. The grey level of the marker represents the difference between the performance of the best and the second best protocol normalized over the four values of f. The darker the mark, the greater the difference.

Figure 2 shows the overall relative performance of $P_{|log|}$, P_{log}, and P^+_{det} for $f = 4$ (the cubes for the remaining values of f can be found in [3]). The cube shows which areas of the applications space each of the protocols excels. In general, P_{log} does fairly well, especially in the middle of the application space. When branchiness is very low, however, the communication graphs resemble linear chains and the number of unnecessarily piggybacked determinants is bounded and low. A process that is communicating with only one other process might unnecessarily send it a determinant, but it will only do so once. In this case, $P_{|log|}$ does just as well as P_{log} but consumes less bandwidth.

When branchiness and latency are high, determinants

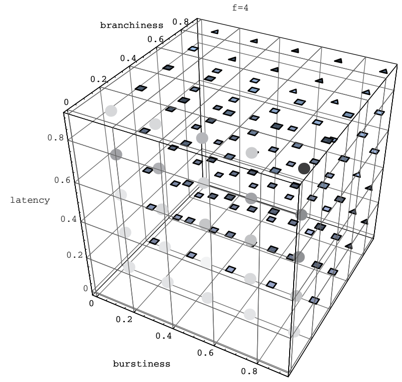

Figure 2. Overall performance of $P_{|log|}$ (sphere), P_{log} (square), and P^+_{det} (triangle) for $f = 4$

are disseminated widely throughout the system. In this case, having better estimates of $Log(m)$ should have a drastic effect. However, as we see from Figure 2, an equally important factor in containing the dissemination of determinants is to send information about which determinants are stable. In the region of the application space in which branchiness, latency, and burstiness are high, P^+_{det} is the best protocol.

As f increases, however, the advantage of using P^+_{det} decreases, until it virtually disappears for $f = 9$, when all 10 of the processes must store a copy of the determinant and containing dissemination is pointless. $P_{|log|}$ is still the winner even at high f [3].

We also compared the overall relative performance of $P_{|log|}$, P_{log}, P^+_{det} and Manetho. Independent of the value of f, Manetho forwards copies of determinants to all processes, and hence performs relatively poorly when f is low. As f increases, Manetho does well under the same circumstances in which $P_{|log|}$ excels, that is, when branchiness is low. For large f, Manetho also does well when latency and burstiness are high. In this region, applications send lots of messages in each burst, and, because the network is slow to acknowledge the messages, a lot of redundant determinants are sent. For these applications, there is no advantage in building accurate estimates of $Log(m)$: the best policy is to send the minimum amount of information, which is precisely what Manetho does.

5 Conclusions

This paper explores the basic tradeoff between the benefit of having information and the cost of obtaining that information in the context of message logging protocols in a synthetic application setting. The BBL application model attempts to model scientific applications parameterized along three dimensions: branchiness, burstiness, and latency.

After running a total of 15,000 simulations with varying parameters and protocol options, we can conclude that while full causal information greatly reduces the bandwidth overhead of causal logging, the cost of sending this information is too high to recommend using. We also conclude that $P_{|log|}$ is overall a better choice than P_{det}. This is somewhat surprising and contrary to our initial intuition. While P_{det} piggybacks fewer bits more of the time, $P_{|log|}$ does only a little worse, and occasionally, when it beats P_{det} it does so by a significant amount.

If the only property that can be fixed ahead of time is the value of f, then for small f one should use $P_{|log|}$ and for large f one should use P_{log}. If one can also say that the latency or branchiness is large, then P_{det}^{+} is the best choice for small f. If one has to choose a protocol with little knowledge of the communication properties, then P_{log} would be a reasonable choice since P_{log} wins most often.

Manetho appears to be the best choice only when a large number of failures must be masked and either branchiness is low or burstiness and latency are high. Otherwise, to tolerate a large number of failures, once again, P_{log} is the best choice.

The applications within the BBL model are very sensitive to latency, and to a lesser degree, branchiness. In particular, it is very sensitive to the speed of the acknowledgments. This sensitivity is greater as branchiness increases.

It should be noted that these results are only valid for the BBL application model. Communication patterns generated in other ways could well lead to different results. These recommendations would most likely change for larger values of n, but we have not yet studied these cases. As n grows, we expect that P_{det} and $P_{|log|}$ will do relatively better than the rest, especially P_{det}^{+}. For sufficiently large n, we believe that $P_{|log|}$ will be the best choice for small f and P_{det} (or Manetho) will be the best choice for large f. However, experiments are needed to verify this intuition.

References

[1] L. Alvisi and K. Marzullo. Tradeoffs in implementing causal message logging protocols. In *Proceedings of the 15th ACM Symposium on the Principles of Distributed Computing*, pages 58–67, June 1996. Philadelphia, PA.

[2] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[3] K Bhatia, K. Marzullo, and L. Alvisi. The relative overhead of piggybacking in causal message logging protocols. Technical Report ????, UCSD CSE, January 1998.

[4] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a communications characterization methodology for parallel applications. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 310–319, February 1997. San Antonio, TX.

[5] E. Elnozahy, D. Johnson, and Y.M.Wang. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, September 1996.

[6] 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.

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