The Relative Overhead of Piggybacking in Causal Message Logging Protocols

Karan Bhatia
University of California, San Diego
Dept. of Computer Science and Eng.
kbhatia@cs.ucsd.edu

Keith Marzullo
University of California, San Diego
Dept. of Computer Science and Eng.
marzullo@cs.ucsd.edu

Lorenzo Alvisi
University of Texas, Austin
Dept. of Computer Sciences
lorenzo@cs.utexas.edu

Abstract

In order to restore crashed processes to a consistent state, message logging protocols ensure that these processes, when recovering, make the same choices when re-executing nondeterministic events. Causal message logging protocols ensure this by piggybacking the results of these choices (called determinants) on the ambient message traffic. By doing so, causal message logging protocols do not create orphan processes nor introduce reliability-generated blocking in failure-free runs of a distributed program.

Family-based logging is a set of causal message logging protocols. These protocols use different techniques to determine when a determinant should be piggybacked on an application message. Given an application's pattern of communication, however, it is not obvious which family-based logging protocol is more efficient. In this paper, we examine the efficiency of family-based logging protocols under different application and environmental assumptions.

This paper contains approximately 5,000 words.

Keywords: Fault-tolerance, Message logging, Causal message logging, Family-based logging, Performance Evaluation

1 Introduction

Message logging is a popular technique used in distributed systems for masking crash failures. Message logging protocols save information about received messages so that when a process crashes, it can be restarted from a checkpoint, re-sent the same sequence of messages it had previously received after the checkpoint, and have any regenerated messages captured and discarded. Examples of message logging protocols include [1, 5, 7, 9, 10, 12, 15, 16, 17]

A difficulty in message logging protocols is avoiding the creation of *orphan processes*. Orphan processes are correct processes whose state is inconsistent with the recovered state of a crashed process. These inconsistencies arise from the inability to reproduce during recovery some of the non-deterministic choices that a process performed before crashing. For instance, suppose process p crashes and is restarted. If during recovery p makes a different nondeterministic choice than it did before crashing, then another process q may find itself in a state that reflects the earlier nondeterministic choice. In this case, the states of p and q would be inconsistent, and q would be an orphan.

Message logging protocols typically assume that the only nondeterministic choice a process makes is in selecting, among the received messages, which message to deliver next. Hence, if p delivers the same sequence of messages it delivered before crashing, then no orphan processes will be created.

Thus, a critical design decision in message logging protocols is in how to record the order in which a process delivers messages. *Pessimistic* protocols ensure that this order is recorded to stable storage before the process sends a further message. This can introduce blocking in failure free runs. *Optimistic* protocols avoid this blocking by recording the ordering information asynchronously. This can improve performance, but if a crash occurs, then orphans may be created. Hence, recovery in optimistic message logging protocols is significantly more complicated than recovery in pessimistic protocols. In addition, any orphan processes must roll back their states.

Causal protocols also avoid the blocking introduced by pessimistic protocols, but do it in such a way that orphans are never created. As long as the ordering information associated with a message m is at risk of being lost due to failures, this information is replicated in the volatile memory of those processes whose state causally depends on the delivery of m by some process p. Hence, the processes that are at risk of becoming an orphan should p crash keep the information necessary to recover p to a consistent state.

Causal message logging protocols distribute determinants by piggybacking them on messages exchanged by processes. Let m be a message, and let $\#m$ (read "the *determinant* of m") be the ordering information associated with m. Let $Log(m)$ be the processes that have recorded $\#m$ in their volatile memory. Suppose that a process $p \in Log(m)$ sends a message m' to process q. Once q delivers m' its state causally depends on the delivery of m. Therefore, if $\#m$ is at risk of being lost due to failures, then q should be in $Log(m)$ by the time it delivers m'. If we further assume that no more than f processes can be crashed at any time, then $\#m$ cannot be lost once $|Log(m)|$ is larger than f. Hence, p needs to piggyback $\#m$ on m' when $|Log(m)| \leq f \wedge q \notin Log(m)$.

Unfortunately, $Log(m)$ is not easily determined by p. Instead, p must maintain conservative estimates $Log_p(m) \subseteq Log(m)$ and $|Log(m)|_p \leq |Log(m)|$ and then piggyback $\#m$ on m' when $|Log(m)|_p \leq f \wedge q \notin Log_p(m)$. The value $|Log(m)|_p$ is p's estimate of the size of $Log(p)$, which can simply be the size of p's estimate $Log_p(m)$ of $Log(m)$.

Family-based logging (FBL) is a family of causal message logging protocols that use different techniques for computing $Log_p(m)$ and $|Log(m)|_p$. The five FBL protocols presented in [3, 4] differ in how accurately they compute $Log_p(m)$ and $|Log(m)|_p$. They also differ in the amount of information they piggyback on messages: roughly speaking, the more information that is piggybacked, the more accurate the estimates are. It is not obvious, however, which of these five protocols imposes the smallest average piggyback load for a given communication pattern. One can construct communication patterns in which the least accurate protocol has the smallest average piggyback load and other communication patterns in which the most accurate protocol has the smallest average piggyback load. In general, it is very hard to predict from a given communications pattern which protocol will have the smallest average piggyback load.

This paper is a first attempt to develop a technique to make such a prediction. In order to restrict our domain, we ignore checkpointing and only consider systems that consist of ten processes. We examine family-based logging protocols in a three-dimensional space of communication patterns that captures properties of communication networks and application properties. For each protocol and point in this space, we look at values of f from the set $\{2, 3, 4, 9\}$ in order to understand how they behave for small (and usually reasonable) values of f and for the largest value of f. We consider all the family-based protocols described in [4] after making a few small modifications in order to remove some obvious inefficiencies.

We also consider two other protocols: a new family-based logging protocol that proves to be efficient in part of the space that we examine, and Manetho [7], a causal message logging protocol that is built explicitly for the case in which total failures are possible.

The paper is organized as follows. Section 2 reviews the system model and the family-based message logging protocols we consider in this paper. Section 3 describes the communications model we use in the evaluation of these protocols. Section 4 contains the evaluation and Section 5 summarizes our results.

2 Family-Based Message Logging Protocols

The set of family-based message logging protocols all operate in roughly the same manner. The determinant $\#m$ of a message m contains:

- the process $m.sender$ that sent m;
- the ordinal value $m.ssn$ of m. $m.ssn = \ell$ means that m was the l^{th} message sent by $m.sender$;
- the process $m.receiver$ that delivered m;
- the ordinal value $m.rsn$ of m. $m.rsn = \ell$ means that m was the l^{th} message delivered by $m.receiver$.

A message m is *stable* when its determinant $\#m$ is stored in the volatile memory of at least $f + 1$ processes; otherwise, it is *unstable*. If process p stores $\#m$ in its volatile memory and sends a message m' to process q, then 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 $\#m$: $|Log(m)|_p \leq f \wedge q \notin Log_p(m)$. The members of the family-based protocols differ in how they compute $|Log(m)|_p$ and $Log_p(m)$. They do so by piggybacking different information along with determinants.

The simplest protocol, P_1, piggybacks only determinants. This protocol has the advantage of simplicity, but shares no additional information to improve the estimates $|Log(m)|_p$ and $Log_p(m)$. For example, suppose $f = 5$ and p delivers a message m. Then, p sends m_1 to q. Message m_1 piggybacks $\#m$. If q then sends m'' to r, m_2 also piggybacks $\#m$ (as well as $\#m_2$). Once r receives m, r sets $Log_r(m) = \{p, q, r\}$ because r clearly has a copy of $\#m$, the receiver $p = m.receiver$ must have a copy of $\#m$, and q must have a copy of $\#m$ since q piggybacked it to r. If r then sends a message m_3 to s, then m_3 also piggybacks

$\#m$. Process s then sets $Log_s(m)$ to $\{p, r, s\}$ using the same reasoning as before. Note that process q is not included; there is not sufficient information for s to determine that q has a copy of $\#m$.

With protocol P_2, a process p includes with each determinant $\#m$ the value $|Log(m)|_p$. This is only $\log(f)$ additional bits per determinant. In the above example, when r sends m_3 to s it includes with $\#m$ the value of $|Log(m)|_r$, which in this case is 3. When s receives m_3, it still sets $Log_s(m)$ to $\{p, r, s\}$ but can set $|Log(m)|_r$ to 4, because r's estimate could not have included s since otherwise r would not have piggybacked $\#m$ on m_3.

Instead of $|Log(m)|_p$, under protocol P_3 a process p piggybacks $Log_p(m)$ with $\#m$. This takes more bits ($Log_p(m)$ is encoded as an n–bit vector) but further improves the estimates. In the example above, process s sets $Log_s(m)$ to $\{p, q, r, s\}$ when it receives m_3.

The next three protocols send constant size matrices that encode information about all determinants, both stable and unstable, in the causal past. The additional piggyback load is independent of the number of determinants piggybacked on a message. Let n be the number of processes in the system. Protocol P_4 is an extension of P_2. Instead of piggybacking the value $|Log(m)|_p$ with each determinant, P_4 piggybacks on each message a *stability matrix* SM of size $(f+1) \times n$. $SM_p[i, q]$ contains the highest receive sequence number of any message m delivered by q for which the number of processes logging $\#m$ is i. The last row of the matrix is the *stability vector* and encodes all the known stable messages. Protocol P_6, a novel causal protocol, differs from P_4 in that it only sends the n-word stability vector. Finally, protocol P_5 extends P_3 by piggybacking a *dependency matrix* of size $n \times n$. This matrix is a vector of vector clocks [13] based on receive sequence numbers, and therefore encodes exactly $Log(m)$ restricted to the causal past of p. Further details of protcols P_1 through P_5 can be found in [3, 4]. Table 1 summarizes the piggybacking overhead of the six family-based logging protocols. In this table, W is the number of bits needed to represent a "word" that can hold an entry in the stability matrix, a component of a vector clock, a send sequence number, and so on.

Manetho [7] is a variation of protocol P_1. Manetho assumes that total failures are possible, which means that a message is stable only when its determinant is logged in every process' volatile memory. In Manetho, a process piggybacks $\#m$ on a message m' to q when p has a copy of $\#m$ and $q \notin Log_p(m)$. Process p computes $Log_p(m)$ as in P_1 and no additional information is piggybacked on messages [8].

protocol	per determinant		per message	
	information	bits	information	bits
1	$\#m$	$2\log(n) + 2W$		
2	$\{\#m, \|Log(m)\|_p\}$	$2\log(n) + 2W + \log(f)$		
3	$\{\#m, Log_p(m)\}$	$2\log(n) + 2W + n$		
4	$\#m$	$2\log(n) + 2W$	stability matrix	$(f+1)nW$
5	$\#m$	$2\log(n) + 2W$	dependency matrix	n^2W
6	$\#m$	$2\log(n) + 2W$	stability vector	nW

Table 1: Six Family-Based Logging Protocols

3 The BBL Communications Model

The relative piggyback overhead of the various family-based logging 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, [1, 6, 14]). 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* where 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 no 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 $\times$ 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 $\times$ 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.

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 communication patterns were generated with varying random seeds. We also generated a few longer communication patterns in order to ensure that the results we obtained did not change significantly with more messages.

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 taken. We chose small values of f since for real systems of ten processes the probability of having more than a few failures

protocol	minimum	maximum	average
1	112,127	1,876,600	778,429
2	112,190	1,899,390	762,576
3	123,985	1,848,680	727,540
4	112,127	3,483,110	1,547,280
5	1,707,560	3,083,070	2,101,130
6	269,672	2,037,230	838,008

Table 2: Overall communication overhead for each protocol

protocol	minimum	maximum	average
1	926	21,507	8,711
2	926	21,507	8,465
3	926	18,443	7,065
4	891	21,508	7,378
5	879	16,654	5,412
6	891	21,508	7,532

Table 3: Number of determinants piggybacked for each protocol

at any time is usually very small. We included $f = 9$ since this allows recovery from total failures (for both $f = 9$ and $f = 10$ a determinant is stable only when it is stored in each process' volatile memory). In addition, the measurements for P_1 with $f = 9$ also apply to Manetho.

4.1 Winner Statistics

For each run, both the total number of determinants and the total number of piggybacked bits were measured. The latter value ranged from approximately 112 Kbits to approximately 3.4 Mbits (7 words to 211 words on average per message). Tables 2 and 3 show the minimum, maximum, and average number of bits of communication overhead and number of determinants for each protocol.

The poor performance of P_4 and P_5 is understandable because they send a relatively large

	P_1	P_2	P_3	P_4	P_5	P_6	
# wins	652	122	1,469	0	0	257	total=2,500

Table 4: Number of runs in which a protocol sends the fewest bits

and fixed size matrix with each message. For P_5 the cost is an extra $10 \times 10 \times 32 = 3,200$ bits per message. Sending this matrix has some benefits, since P_5 sends by far the fewest number of determinants. However, the savings achieved by sending less determinants are not enough to make up for the size of the matrix. On average, P_5 piggybacks three times the number of bits as the best protocol for each parameter setting, and in some cases P_5 piggybacks as much as 18 times more bits than the best protocol. P_4 performs similarly, and so we eliminate both P_4 and P_5 from further evaluation.

P_2 is somewhat of a surprise: in almost 5% of the cases it is the best performer. We had initially believed that the extra information carried in P_2 over P_1 was not that useful. Further, P_3 carries slightly more information than P_2 that we believed would be much more useful. Our initial beliefs seem to have been wrong. In fact, looking at the cases where P_1 wins, we see that P_2 is always close behind: 633 times out of 652, P_2 is second best. And in the cases where P_1 wins, it does so by a very small amount, only 0.4% better than the second-best protocol. When P_2 wins, it wins by a more significant amount. Comparing just the average difference in bits between P_1 and P_2, we see that, on average, P_1 actually does 2.5% worse than P_2. We conclude that the value of the extra few bits is well worth their price. For further examination, we therefore only concentrate on protocols P_2, P_3, and P_6.

4.2 Regression Analysis

Using the UnixStat [11] statistics package, we performed regression analysis on the remaining protocols to determine the effects of different factors on the total communication overhead. The results are summarized in Table 5. 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 0.78, 0.83, and 0.75 for P_2, P_3, and P_6 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

protocol	f	burstiness	branchiness	latency
P_2	8.9e+05	2.0e+05	6.5e+05	7.8e+05
P_3	7.3e+05	2.7e+05	5.7e+05	8.1e+05
P_6	10.3e+05	1.9e+05	4.7e+05	7.0e+05

Table 5: Linear Regression

this is reasonable. As f increases, more redundancy is needed in the system and therefore more determinants are piggybacked between processes. P_6 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.

As latency increases, the number of events between sending a message and the corresponding acknowledgment increases. Messages sent in this interval will unnecessarily piggyback some determinants, causing an increase in communication overhead.

Between the burstiness and branchiness factors, it should be noted that burstiness, a measure of the computation to communication ratio, is less significant than the size of the communication neighborhood. In general, when a process sends two messages to the same recipient, the second message need only include the "new" determinants that were not included in the first message. This trimming effect is more pronounced as the size of the neighbors set decreases.

4.3 Understanding the relative performance of the protocols

While giving some idea about the sensitivity of the individual protocols to the coordinates of the BBL model, the regression analysis does not effectively convey the relative strengths and weaknesses of the protocols. To address this, we display our results in a *color cube*, which is a unit cube with burstiness, branchiness and latency as the axes. Each point in the color cube represents a point in the BBL communications space.

Each protocol is represented by a color: magenta for P_2, cyan for P_3 and green for P_6. For each point in the cube at which simulation results were collected, the color of the point reflects the protocol that piggybacked the least amount of data. The deeper the color, the less amount of data it piggybacked compared to the second-best protocol.

Figure 1 is a color cube comparing just protocols P_2 and P_3. For small f, protocol P_2

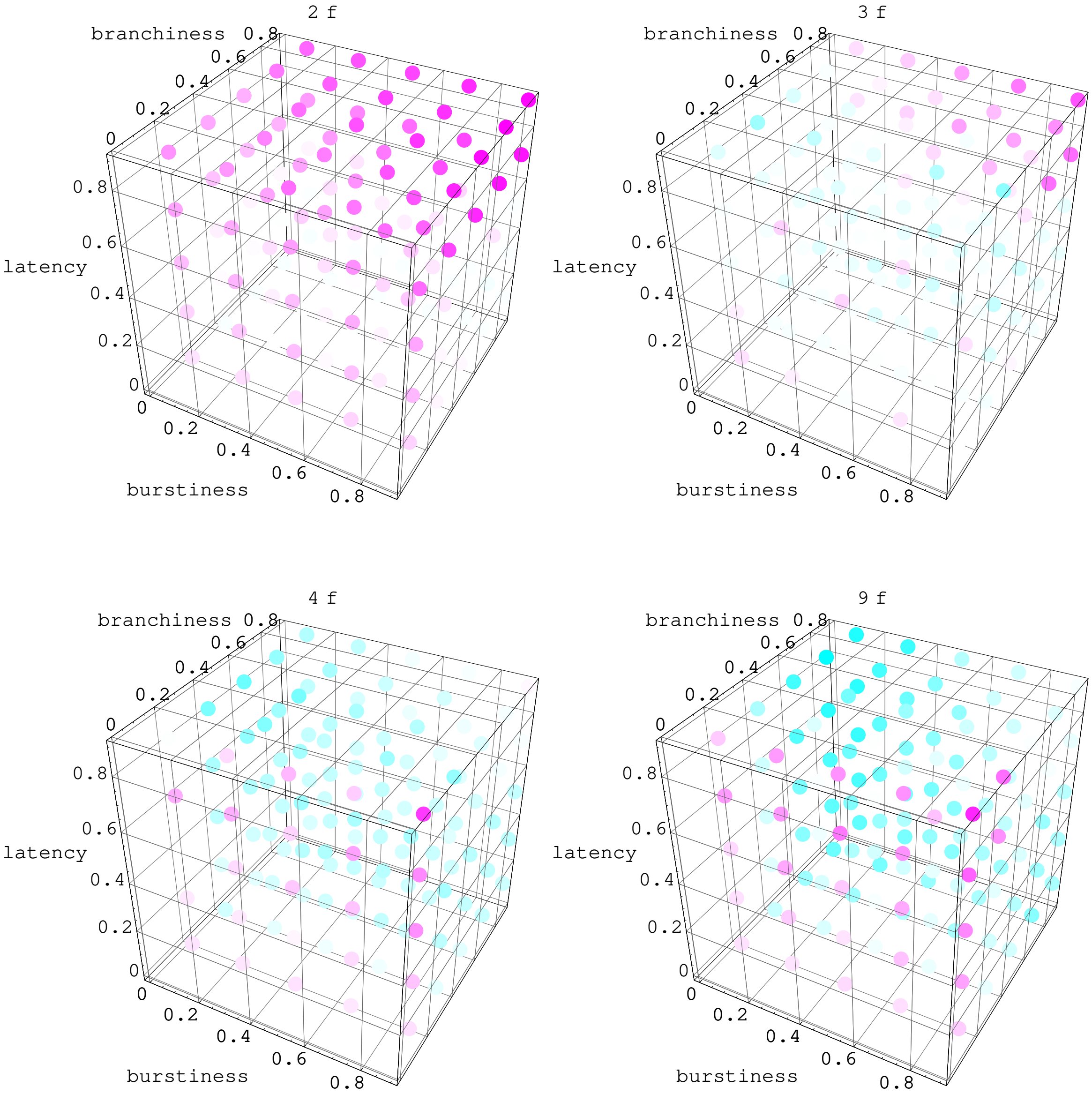

Figure 1: Relative performance of P_2 (magenta)and P_3 (cyan)

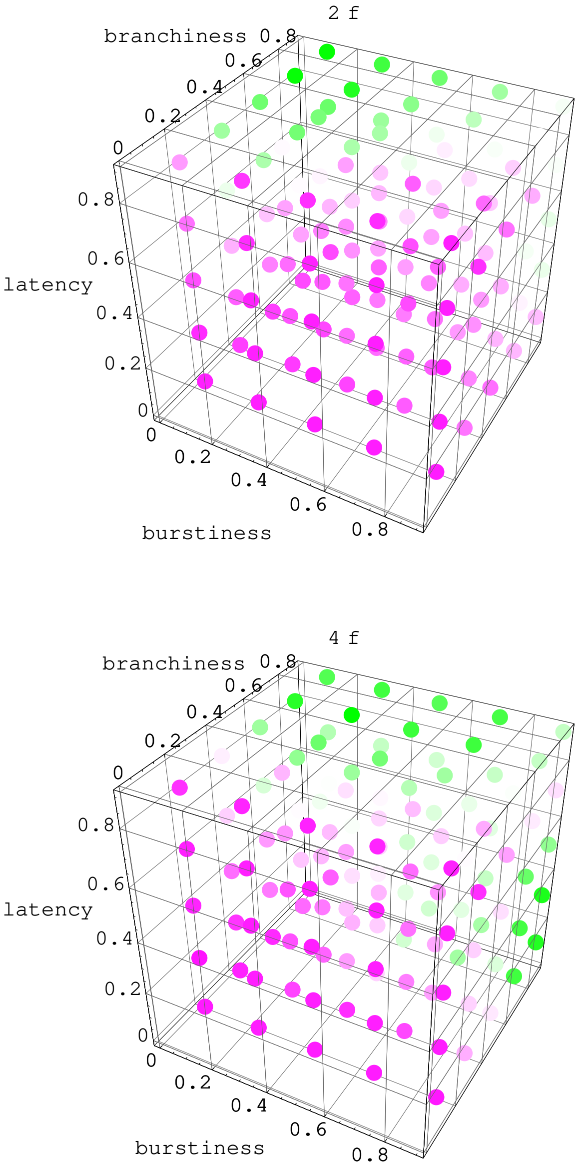

Figure 2: Relative performance of P_2 (magenta)and P_6 (green)

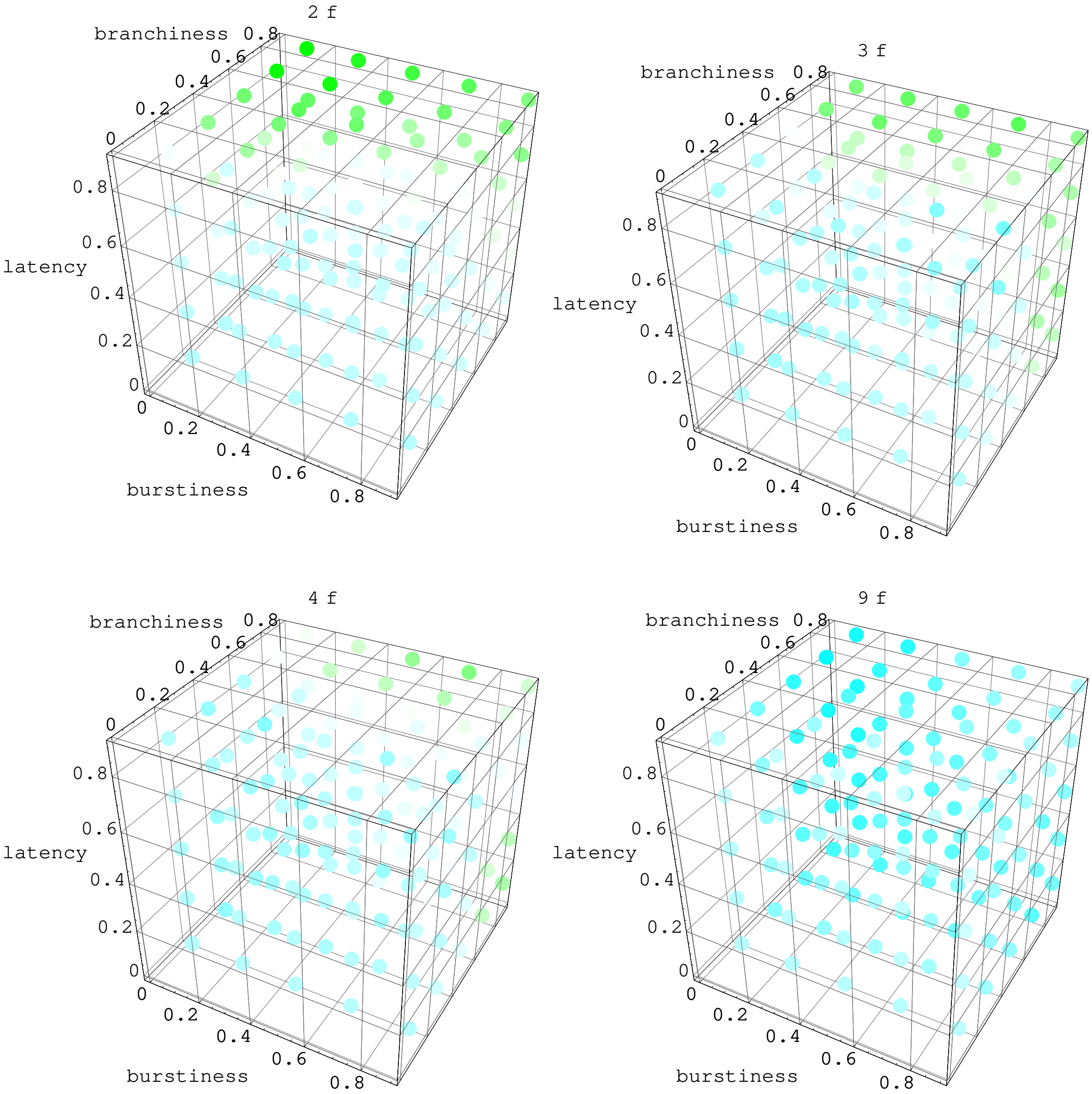

Figure 3: Relative performance of P_3 (cyan)and P_6 (green)

does well compared with P_3; indeed, for $f = 2$, it is a better overall choice. As f increases, P_3 does increasingly better throughout most of the BBL space, except where branchiness is very low. In this case, even with high f, P_2 seems to do well. When branchiness is very low, processes are communicating with a small fixed set of processes. In this scenario, 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. As branchiness increases, the number of times a determinant is unnecessarily piggybacked also increases. We conclude, therefore, that when branchiness is very low, the value of sending additional information is not as high, because the cost of not having that information is low.

Figure 2 is a color cube comparing just protocols P_2 and P_6. We can see that P_2 outperforms P_6 over most of the space. P_6 is better where branchiness and latency is high and f is low. In this situation, determinants will be spread around quickly and will become stable relatively soon. However, with P_2 most of the processes will be unable to determine that a determinant has become stable. Protocol P_6, however, piggybacks information about message stability with each message. As soon a message becomes stable, the news will spread as fast as the unstable determinant was spread. With a small f, this saves bandwidth, but as f increases the savings diminishes. The same effect can be seen in the color cube of Figure 3 that compares protocols P_3 and P_6.

Figure 4 shows the overall relative performance of the three protocols P_2, P_3, and P_6. For $f = 2$, P_2 is the best choice unless latency is high. As f increases, P_3 becomes more appropriate for the points where P_2 was better except when branchiness is very small. For $f = 9$, P_3 is the best choice for all points except where branchiness is very small or burstiness is very large.

Figure 5 shows the overall relative performance of P_2, P_3, P_6 and Manetho. Manetho is displayed in black. In order for this to be a fair comparison, we only consider $f = 9$, since as can be seen from the linear regression, the other protocols will piggyback much less data for smaller f. Manetho is the best protocol for small branchiness or for high latency and high burstiness. However, for most of the space, protocol P_3 is the best choice. This is especially true for high branchiness and low burstiness; that is, when a process communicates with many other processes, but it sends only a few messages at a time.

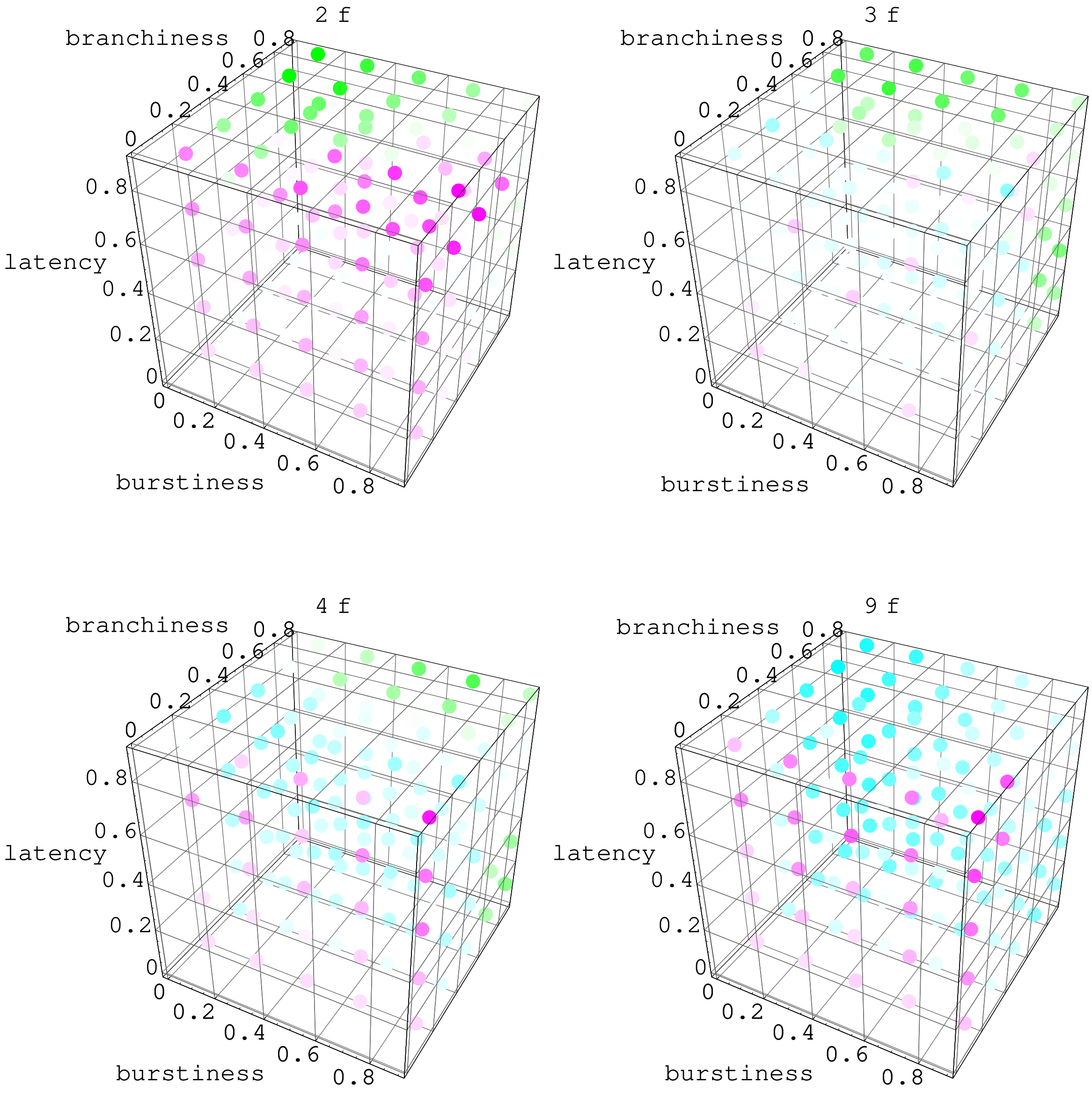

Figure 4: Overall performance of P_2 (magenta), P_3 (cyan), and P_6 (green)

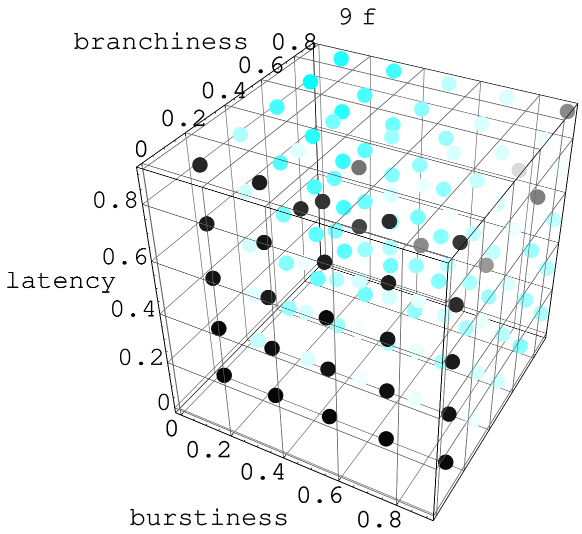

Figure 5: Overall performance of P_1 (black), P_2 (magenta), P_3 (cyan), and P_6 (green) for $f = 9$

5 Conclusions

The piggybacking overhead of causal message logging strongly depends on the communication and system properties. The most important property is the number of failures that must be masked. The second most important property is the latency of the underlying communication system. It appears worthwhile to determine these properties before selecting the causal message logging protocol to use.

If the only property that can be fixed ahead of time is the value of f, then for small f one should use P_2 and for large f one should use P_3. If one can also say that the latency is large, then P_6 is the best choice for small f; otherwise, the previous recommendations hold.

Manetho appears to be the best choice only when a large number of failures must be masked and either branchiness is low or burstiness is high. Otherwise, to tolerate a large number of failures, P_3 is the best choice.

If one has to choose a protocol with little knowledge of the communication properties, then P_3 would be a reasonable choice since, as can be seen from Table 4, P_3 wins most often.

These recommendations would most likely change for larger values of n, but we have not yet studied these cases. As n grows, we would expect that P_1 and P_2 will do relatively better than the rest, especially P_6. For sufficiently large n, we believe that P_2 would be the best choice for small f and P_1 (or Manetho) would be the best choice for large f. However, experimentation is needed to verify this intuition.

Another factor that is worth studying is how the results change with the number of messages, that is, with the frequency of checkpointing. We have ignored the effect of checkpointing in this version because it is not clear how to evaluate its value. Frequent checkpointing reduces the total number of message determinants that need to be stored, but imposes an overhead both on storage and on computation.

All 15,000 runs that were used in the analysis in this paper are publicly available at `ftp://ftp.cs.ucsd.edu/pub/grad/kbhatia/fbl12-3.data`. Interested parties can ask the authors for format information and for access to our simulator.

References

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

[2] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal and Optimal. *IEEE Transactions on Software Engineering*, February 1998, to appear.

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

[4] L. Alvisi and K. Marzullo. Tradeoffs in implementing optimal message logging protocols. In *Proceedings of the Fifteenth Symposium on Principles of Distributed Computing*, pages 58–67. ACM, June 1996.

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

[6] 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 Compuoptcentraljzter Architecture*, pages 310–319, San Antonio, TX, February 1997.

[7] 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.

[8] E.N. Elnozahy. Private correspondence, 12 September 1997.

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

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

[11] G. Perlman and F. L. Horan. Report on the UNIX/STAT release 5.1: Data analysis programs for UNIX and MSDOS. In *Proceedings of the Fifteenth Annual Meeting of the Society for Computers in Psychology*, pages 168–176, Boston, MA, 21 November 1985.

[12] 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.

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

[14] 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, Cape May, NJ, June 1994.

[15] 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.

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

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