UNDERSTANDING THE MESSAGE LOGGING PARADIGM FOR MASKING PROCESS CRASHES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Lorenzo Alvisi

January 1996

UNDERSTANDING THE MESSAGE LOGGING PARADIGM FOR MASKING PROCESS CRASHES

Lorenzo Alvisi, Ph.D.

Cornell University 1996

Message logging is a popular technique for building systems that can tolerate process crashes and transient channel failures. The technique, which was first developed in the mid-80s, is popular because message-logging protocols are relatively simple and require process replication only when a process fails. Surprisingly, however, very little attention has been given to the formal specification of the consistency property that these protocols implement in order to be able to recover failed processes to a consistent state.

This dissertation presents the first such formal specification. From this specification, the two major classes of message-logging protocols, namely *optimistic* and *pessimistic*, are characterized. A third and new class of message-logging protocols, called *causal*, is introduced. A notion of optimality, based on three important performance metrics, is proposed, and it is shown that optimal implementations of causal message-logging protocols exist. In particular, it is shown that

causal message-logging protocols combine the positive aspects of optimistic and pessimistic message logging.

A subclass of causal message-logging protocols, called *family-based logging*, is developed. Family-based logging protocols are optimal and have the additional attractive characteristic that the smaller the maximum number of concurrent failures, the lower their overhead. Furthermore, several compression techniques can be used to reduce this overhead. Finally, it is shown that family-based logging protocols can be implemented in order to take advantage of the different patterns of communication that systems exhibit.

Biographical Sketch

Lorenzo Alvisi was born in Bologna, Italy, on October 13, 1961, of Franco and Giovanna. Three years later, he was introduced to Claudia, the most wonderful sister one can imagine to have. He spent endless summers playing soccer and eating Nutella, until he entered high school at the Liceo Classico Luigi Galvani. In the following five years, he tried to survive its Latin tests and at the same time to understand the meaning of life, sharing long afternoons of hard study, discussions, and laughs with his friend Fabio Fontana.

He was 17, when one magical afternoon he skillfully maneuvered to sit, during his English class, next to Irene Eibenstein, the prettiest girl he had ever seen. She still is.

After graduating from high school in 1980 at the top of his class, Lorenzo entered the University of Bologna, where he graduated *Summa cum Laude* in 1987 in Theoretical Physics. He received a post-graduate degree in Physics in 1988, and won a fellowship in Theoretical Chemistry the same year. Feeling finally financially secure with a stipend of no less than $1000 per month, Irene and Lorenzo were married on December 26, 1988.

A few months before he had met Ozalp Babaoglu, who had just moved to the University of Bologna from Cornell as a full professor in the Department of Mathematics. Ozalp was patient enough to introduce Lorenzo to Computer Science, and to encourage the dream, which Lorenzo had entertained since high school, to pursue a Ph.D. in the US. After one year at CMU, Lorenzo joined the Cornell Computer Science Department in August 1991.

Cornell has been kind to Lorenzo and Irene — who is pursuing a Ph.D. in Romance Studies — and they have been heard saying that the years in Ithaca have been the happiest of their lives. Maria Lucia joined them on February 21, 1994, in one of the few snow-less days of a very cold winter. Her eyes will help them remember Cayuga Lake.

Lorenzo received a M.S. in Computer Science in August 1994, and was awarded a Doctor of Philosophy degree in January 1996. He is starting a new adventure as an Assistant Professor in the Department of Computer Sciences of the University of Texas at Austin.

A Irene, la compagna dei miei sogni

Acknowledgements

The dream of every graduate student is to find in his advisor a true Mentor. Keith Marzullo has been that, and much more. He has guided me during these years with a light but sure touch, allowing me to grow in my enthusiasm and in my confidence. Our long hours of technical discussions have not only taught me the ropes of the art of doing research, but have been pure fun. But Keith, in addition to being a great advisor and a role model, has also become a true, precious friend. In short, Keith is amazing. Without his support, humor, guidance, friendship and enthusiasm this thesis, the happiest of Thanksgivings with tortellini in brodo, a real Italian espresso at Caesar's Palace, the decision to become an assistant professor and the amputation of my thumb cutting Parmesan Cheese (I know, I should use the canned stuff) during an unforgettable canoe trip would not have been possible.

Fred Schneider, throughout my career at Cornell, has been a constant source of inspiration and support, both on a financial and on a more personal level. I am thankful to Fred for having supported me for several semesters as a research assistant; much more though, I am deeply grateful to him for his personal warmth and support through happy and difficult times. Even on sabbatical, Fred has

always found the time to give me wise and caring advice and to "debrief" me after each of my job interviews. His commitment to always strive for the best possible result, both as a researcher and as a teacher, is a lesson that I will try to apply to the rest of my career.

Sam Toueg has served as the chairman of my committee after Keith left for UCSD, and has supported me for a semester as a research assistant. In our technical conversations I have thoroughly enjoyed Sam's passionate enthusiasm: If anyone is having fun doing research, Sam is. I have also learned to admire Sam's ability to articulate his arguments in a way that is rigorous, and yet does not indulge in unnecessary complications. I hope that I can carry with me a little of that elegance.

By accepting my non-standard request to minor in Music, David Rosen has made one of my most sincere wishes come true. Guiding my independent study in Counterpoint, he has always been patient, despite my many tritones. I just regret that I did not have the opportunity to take his course on Opera.

My career in Computer Science, let alone this thesis, would not have been possible without the support of Ozalp Babaoglu. Ozalp has introduced me to Computer Science and to research in Distributed Systems, and his help and confidence in me have gone beyond what I could ever hope. He has my deepest gratitude.

Ken Birman has followed my progress with interest and warmth that I sincerely appreciated. Many times he has stopped me in the corridors of Upson, giving me advice or simply asking about my research. He has volunteered to write one of my letters of recommendation, and has knowledgeably advised me during my job

search.

Roberto Odorico, my Physics undergraduate advisor at the University of Bologna, has been for me an early example of dedication to research and enthusiasm.

I am grateful to Bruno Marzocchi, my elementary school teacher. I have always believed that his creativity and passion in teaching are largely responsible for my love of mathematics and science.

Bruce Hoppe has been a friend and a great colleague. He was a partner in the early, roaring days of FBL, and the techniques for compressing the information piggybacked by FBL are largely due to him. Even after leaving FBL to dedicate himself to the noble mission of rescuing passengers from sinking ships, he has always been willing to listen to my ideas, and I have greatly benefited from his insights and comments. Discussing technical problems or just making silly jokes with Scott Stoller has been a constant pleasure. Annie Yanhong and Scott have also introduced me to the wonders of home-made Chinese cuisine. Alessandro Panconesi has been a source of intelligent conversations and whole-hearted laughs during my first two years. I really miss him. I have been fortunate to have had special office-mates during these years: Funda Ergun has been a smart, kind, funny, supportive, and liberal friend. They just don't make office-mates like her anymore. Tom Bressoud has been a friendly, competent technical interlocutor, and one tough basketball opponent. Navindra Gambhir has often amazed me with his smartness, and made me laugh with his pungent humor. He saved my life several times with his insights into CS theory, and I will cherish our "Theory Talks" (or *whatever*) for a long time. These years would not have been the same without the friendship of and the stimulating exchanges with David Karr, Pei-Hsin Ho, Rod Moten, Takako Hickey, Robbert van Renesse, Nikos Pitsianis, Sofocles Efremidis, Gurney Hunt and Roy Friedman.

I would also like to thank Becky Personius, Linda Competillo, Diana Catley, Carol Ayer, Pauline Cameron, Cay Wilson and the other members of the Department Staff for the help that they have always been ready to provide, assisting me in all of my administrative problems.

Aleta Ricciardi was at the airport to pick me up the day I arrived in Ithaca, and has found a way to help me at critical times ever since. I am looking forward to share Austin with her and Michael Ogg.

How much we miss Susie Armstrong becomes more evident every time Keith comes briefly back to Ithaca. I am still amazed at how convincing she can be when she claims to be American. Good for her that the INS has never checked her ciambella.

David Corrigan and Mary Cockram have been another reason why these years have been so special. David is also responsible for my first hockey game, my first football game, and my first hot dog. Ithaca just is not been the same since they left.

Getting to know and becoming a friend of Margaret Driscoll is one of the best things that ever happened to me. She is one of the most beautiful human beings I have met, and her apple pie has no equal.

Fabio Fontana, Marco Dore, Franco Semeria and Kim Wagner are a precious treasure. Fabio has been in my teens just short of a brother; Marco is like the home built on the rock; Franco is the only one who can survive — with only minor mental damage — a vacation with Irene, my sister and me; Kim's humor and friendship are treats that unfortunately I get to enjoy all too rarely.

My parents have been an endless source of love, happiness and support. They have first been before me and then at my side every step of the way, even when it meant long years of separation. Grazie, Babbo e Mamma. I just can't imagine my life without my little sister Claudia. She is one of the smartest, sweetest, funniest, most sincere people I've ever met. I miss her so much! Cioni, you are the best.

Maria's smiles have often been for me like SuperPippo's peanuts: they kept me going when I thought I had had it.

Finally, Irene has been during these years my love, my joy and my strength. She is the love of my life and my best friend. This dissertation is dedicated to her.

Table of Contents

List of Figures

Chapter 1

Introduction

Section 1.1 presents the issues that motivate this research. Section 1.2 briefly discusses message logging, the technique for building fault-tolerant distributed systems that has been the focus of our investigation. Section 1.3 briefly outlines the contributions of this dissertation, and Section 1.4 outlines its organization.

1.1 The Challenge

Architectures that connect powerful workstations through high-speed communication networks hold great potential for high-performance parallel computing. In fact, the amount of raw computing power present in a typical modern distributed system with dozens, if not hundreds, of general-purpose workstations may well exceed that of an expensive, special-purpose supercomputer. Programming these systems, however, poses challenges that are not presented by a traditional supercomputer. One of the major challenges is providing fault-tolerance.

Today's important computing applications (e.g. genome analysis) require days or weeks to execute on networks with dozens of workstations [DGB+96]. Hours of computing can be wasted if there are hardware failures, or even if one of the processors is turned off, rebooted, or disconnected from the network by an unknowing user. Such events are actually much more likely to occur than real hardware failures, since workstations are often under the control of different administrative domains — i.e. the individuals who have the workstations on their desks.

Traditionally, the applications requiring fault-tolerance have been those for which failures can have potentially catastrophic consequences. These *critical applications* (such as monitoring a nuclear plant, or an airplane) must be able to recover from arbitrary failures, and the cost of achieving the necessary level of fault-tolerance — in terms of used resources and/or diminished performance — is regarded as secondary concern.

In contrast, most of the applications that one can foresee running on a network of workstations will not be critical in nature. For such applications, fault-tolerance is a desirable option, rather than a necessity, and, as with any option, its benefits must be carefully weighted against its costs. Any technique providing fault-tolerance must:

- Tolerate common kinds of failures.

- Require few additional resources and have a negligible impact on performance during failure-free executions.

- Scale their cost depending on the severity and number of failures that need to be tolerated.

- Integrate with applications in a way transparent to the application programmer.

1.2 Message Logging

Message logging is a technique for providing fault-tolerance that has the potential to address the requirements outlined above. Message-logging protocols are used to build systems that can tolerate the common type of failure in which processes fail by halting. The mechanisms necessary to recover a faulty process combine checkpointing with careful recording of the messages received by each process, and are implemented so that they are transparent to the application.

Message logging is not the only technique for building systems that tolerate this kind of process failures. For example, active replication [Sch90] or primary-backup [BMST92] are other commonly-used techniques. What makes message logging attractive is its simplicity and low cost. A further advantage of message logging is that it can be readily applied to any inter-process communication structure, while both active replication and primary-backup are typically applied in a client-server setting.

The major limitations of message logging are two:

1. The time necessary to recover a process using message logging tends to be higher than the time necessary to recover a process using active-replication or primary-backup.

2. Reaching a consistent global state [CL85] upon recovery of a faulty process is complicated, unless blocking is introduced during failure free execution.

The first limitation can be addressed by increasing the frequency with which checkpoints are taken (although at the cost of slowing down the application). The second limitation has proven to be less easy to address. The consistency condition that must hold in a message-logging protocol upon recovery of a faulty process is often expressed in terms of *orphan processes*. These are correct processes whose state is inconsistent with the recovered state of a process. Consistency requires that when recovery is complete there be no orphan processes. In order to satisfy this requirement, message-logging protocols have traditionally presented a classical trade-off between pessimistic and optimistic protocols. Pessimistic protocols prevent the creation of orphan processes, but do so at the cost of introducing blocking: they offer simple recovery at the cost of relatively poor performance. Optimistic protocols, in order to avoid blocking, may create orphan processes when a failure occurs. Optimistic protocols perform better than pessimistic in the absence of failures, but recovery becomes complex, since when a failure occurs orphan processes must be detected, and their states must be rolled back to achieve consistency.

1.3 Contributions of the Dissertation

As we noted above, the main problem in message logging derives from the difficulty of satisfying the consistency condition that must hold upon recovery of a faulty process in a way that is both simple and efficient. Yet, in spite of the numerous message-logging protocols [SY85,JZ87,JV87,SBY88,SW89,JZ90,Eln93,VJ94], no precise specification of what such consistency condition requires has been presented in the literature. In this dissertation, we present the first formal specification of a necessary and sufficient condition for avoiding orphan processes. After showing how pessimistic and optimistic protocols relate to this condition, we use it to specify a new class of message-logging protocols, that we call *causal.* We propose a set of metrics to evaluate message-logging protocols and characterize the protocols that are optimal with respect to them. Finally, we present family-based logging, a novel set of message-logging protocols that are both causal and optimal. Family-based logging (FBL) protocols combine the positive aspects of both pessimistic and optimistic protocols by preventing orphan processes without introducing blocking. FBL protocols can be customized to meet the needs of different applications. The overhead they exact depends on the number of failures that an application is willing to tolerate, and can be minimized by exploiting the pattern of inter-process communication exhibited by a specific application.

1.4 Outline of the Dissertation

The dissertation is organized as follows. Chapter 2 describes the system model assumed throughout the dissertation. Chapter 3 provides an informal introduction to message logging. Chapter 4 gives a specification of message logging, and characterizes causal message-logging protocols. Chapter 5 defines optimal message-logging protocols, and derives the structure of an optimal causal protocol. Chapter 6 describes family-based logging. Chapter 7 concludes the dissertation. In the Appendix three family-based logging protocols are presented in detail.

Chapter 2

System Model

This chapter defines the system model and states the assumptions used in the remaining of the dissertation.

2.1 System Model

Consider a distributed system consisting of n processes. Processes communicate only by exchanging messages, and a process never sends a message to itself. 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 *local history* of a process p during an execution is a (possibly infinite) sequence of events $h_p = e_p^1, e_p^2...$ where e_p^1 denotes the first event executed by process p, e_p^2 denotes the second event executed by process p and so on. The *global history* of an execution is a set $H = \bigcup_{p=1}^{n} h_p$.

For any events e_i of process i and e_j of process j, we write $e_i \prec e_j$ (read "e_i

immediately precedes e_j") if one of the following two conditions is true:

- $i = j$ and e_i was executed before e_j;
- e_i is a send event and e_j is the corresponding receive event.

The transitive closure of relation $\prec$ is denoted by $\rightarrow$ [Lam78]. If $e_i \rightarrow e_j$ then we say that "e_i happens before e_j", "e_i causally precedes e_j", "e_j causally follows e_i" or "e_j depends on e_i".

Execution of the system is represented by a *run*, which is an irreflexive partial ordering of the send events, receive events and local events in the execution defined by the pair $(H, \rightarrow)$. Define the *causal history* of event e in a run $\rho = (H, \rightarrow)$ as the set:

$$\theta(e) = \{e' \in H : e' \rightarrow e\} \cup \{e\}$$

We define a *causal path* between an event e^1 and an event e^k to be a sequence of events $e^i : 1 \leq i \leq k$ such that $\forall i : 1 \leq i < k : e^i \prec e^{i+1}$ and call i the *index* of event e^i in the causal path. Given a causal path $e^i : 1 \leq i \leq k$, its associated *causal chain of processes* is built as follows.

1. The first process in the chain is p if the first event in the sequence is e^1_p.
2. Suppose that we have scanned the events through event e^i_p and p is the j-th process in the chain. We find the $j + 1$-st process by finding the smallest $h : i < h \leq k$ such that the process q in e^h_q is different from p.

The *length* of a causal chain of processes is one less than the number of elements in the chain.

For each process p, a special class of events local to p are called *deliver events*. These events correspond to the delivery of a message to the application running on p. For any message m from process p_1 to process p_2, we assume that p_2 delivers m only if it has received m and that p_2 delivers m at most once. Furthermore, we assume that a correct process will eventually deliver all messages it has received.

The *state* of a process is a mapping from program variables and implicit variables (such as program counters) to values. The state of a process does not include the variables defined in the underlying communication system, such as the queues of messages that have been received but not yet delivered to processes. Consider states s_p and s_q of processes p and q, $p \neq q$. 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 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, defines a *global state*. A global state is *consistent* if all pairs of states are mutually consistent [CL85];[1] otherwise it is *inconsistent*.

Given an initial state and a local history h_p of a process p, one can construct a local state history σ_p of p. Doing so allows one to represent an event e of process p as the pair of states (s_e, s'_e) of p such that e takes p from state s_e to s'_e. One can then define $\prec$ over states:

1. for all events e, $s_e \prec s'_e$;

2. for all pairs of events e_1 and e_2, $e_1 \prec e_2$ implies that $s_{e_1} \prec s'_{e_2}$.

[1]This definition is different from that of [CL85] in that it is defined in terms of *deliver* events rather than *receive* events. Our usage corresponds to the literature on message-logging protocols.

The transitive closure of $\prec$ over states yields the happens-before relation $\rightarrow$ over states.

Processes are *piecewise deterministic* [SBY88]: execution of a process consists of a sequence of deterministic intervals of execution, joined by non-deterministic events. For each process, the first interval of execution begins with the process' initial state; subsequent intervals begin with each nondeterministic event.

Only deliver events are non-deterministic [2]: a process can choose non-deterministically the next message to deliver from among those that it has received but not delivered. Hence, execution of a process consists of a sequence of intervals, the beginning of each interval being defined by the initial state of the process and by the delivery of a message. Such intervals are called a *state intervals*. Given the first state of a state interval and the message whose delivery defines the beginning of the interval, the remaining states in the interval are uniquely determined.

For any message m delivered by process p, the *receive sequence number* of m, denoted $m.rsn$, encodes the order in which m was delivered: $m.rsn = \ell$ iff m is the ℓ^{th} message delivered by p [SY85]. The state interval that p initiates with the delivery of m is denoted $p[\ell]$ where ℓ, the *index* of $p[\ell]$, is equal to $m.rsn$. State interval $p[0]$ is defined to be the sequence of states of p from its initial state to the state immediately before delivery of the first message.

We make the following further assumptions:

- Channels between processes are point-to-point, FIFO, and can fail only by

[2] We are going to adopt the assumption that message delivery ordering is the only source of non-determinism for the rest of this dissertation. However, the results of this dissertation can be easily extended in principle to arbitrary types of non-deterministic events.

transiently losing messages.

- Processes fail independently according to the fail-stop model [Sch84]. In this model, processes fail by halting, and the fact that a process has failed is eventually detected by all non-faulty processes.

- Each process knows the identities of the fixed set of processes comprsing the system.

- *Stable storage*, a system utility used to atomically and reliably log data [Gra77], is available across the system and persists across failures.

- There are sufficient resources to restart a faulty process eventually.

Chapter 3

The Message-Logging Approach

This chapter provides an informal introduction to the message-logging technique for providing fault-tolerance. The chapter is structured in five sections. Section 3.1 gives motivations that led to the development of message logging and argues why the technique remains attractive today. Section 3.2 defines the *no-orphans consistency condition*, which guarantees that after a failure the distributed system will be restored to a consistent global state. The section then discusses how message logging implements this condition. Section 3.3 discusses the role played by non-determinism and introduces the notion of the *determinant* of a non-deterministic event. Section 3.4 outlines the general structure of a message-logging protocol. Finally, Section 3.5 discusses central issues that arise in the design and implementation of message-logging protocols.

3.1 Background and Motivation

Fault-tolerance can be achieved only through some form of redundancy. In *temporal redundancy*, an application is restored to a previous consistent state when it fails, and the lost computation is then repeated. This procedure is called *recovery*. In *spatial redundancy*, several independent copies of the application are executed concurrently on different processors, and the final result is determined through some form of voting [Sch86]. The most appropriate form of redundancy for a specific application depends on the nature of the application and the type of failures that must be tolerated.

For instance, spatial redundancy, which guarantees that no computation will ever need to be repeated because of a failure, is well suited for applications that have strict timing constraints. Spatial redundancy is also the only way to tolerate failures that exhibit arbitrarily malicious behavior (sometimes called *Byzantine failures*) [LSP82]. Spatial redundancy, however, requires a considerable investment of computing resources. To tolerate a single malicious failure, it is necessary to execute copies of the same application on three independent processors [vN56]. Applications that do not have strict timing constraints or for which failures do not have potentially catastrophic consequences may not warrant this kind of investment. Consider, for instance, parallel scientific applications. If a parallel scientific application needs to be restarted from its initial state because of a failure, time and money may be lost but there should be no danger to public safety. Hence, parallel scientific applications usually do not warrant the dedication of two thirds

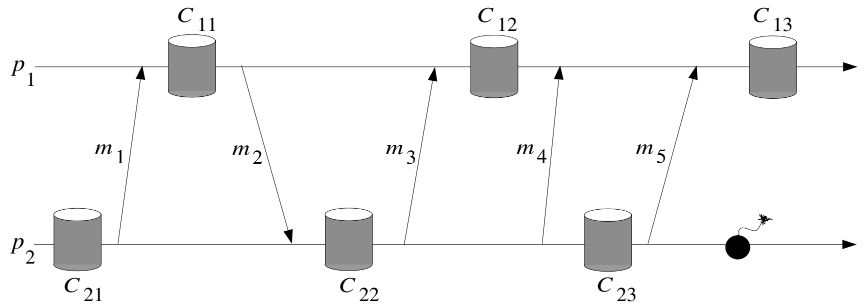

Figure 3.1: Processes taking independent checkpoints.

of the available computing resources in order to tolerate a single malicious failure. For such applications, temporal redundancy is the more appropriate approach.

It is relatively easy to implement fault-tolerance based on temporal redundancy for a sequential application running on a single processor. The state of the application is periodically saved on stable storage through an action called *checkpointing*. Each saved state is called a *checkpoint*. If a failure occurs, then the application is resumed from the last checkpoint.

The situation is considerably more complex, however, for a distributed application. The global state of such an application comprises the union of the local states of the processes running the application. A checkpoint of the global state of the application can therefore only be obtained by taking local checkpoints of the application processes. Unfortunately, an arbitrary set of local checkpoints need not constitute a consistent global state [CL85].

For example, consider Figure 3.1. Process p_1 and process p_2 exchange messages and take periodic and independent checkpoints of their local states, denoted as C_{1i} and C_{2i} for $1 \leq i \leq 3$. Suppose process p_2 fails some time after having sent message

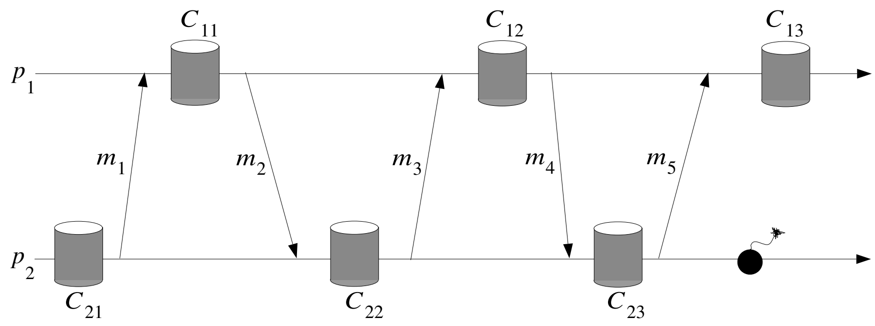

Figure 3.2: An execution leading to the domino effect.

m_5 to p_1, and consider the global state obtained by combining C_{13} and C_{23}, the latest checkpoints of p_1 and p_2. This global state is inconsistent, since C_{13} and C_{23} are not mutually consistent: C_{13} reflects the delivery of message m_5, but C_{23} does not reflect the sending of m_5. Hence, restoring the system to this global state would not yield a previous consistent global state of the system. The global state obtained combining C_{12} and C_{23}, however, is consistent. To restore the system to this consistent global state, the state of p_1 must be *rolled back* to C_{12}.

In general, there is no guarantee that, by rolling back processes, a non-initial consistent global state will be found eventually. Indeed, if processes take checkpoints independently, then there exist runs in which the only consistent global state to which the application can be rolled back is the initial state. An example of such a run is presented in Figure 3.2.

As in Figure 3.1, when process p_2 fails, process p_1 rolls back to checkpoint C_{12}, which preceded the delivery of m_5. In this case, though, C_{12} and C_{23} do not constitute a consistent global state because p_2 has delivered m_4, which p_1 sent after checkpoint C_{12}. Hence, p_2 must be rolled back to C_{22}. Unfortunately, C_{12} and C_{22}

do not constitute a consistent global state either. In fact, the only result of rolling back p_2 is to reverse the roles of p_1 and p_2: now it is p_1 that must roll back, this time to undo the delivery of m_3. The pattern of communication of Figure 3.2 will eventually force p_1 and p_2 back to their initial states. This uncontrolled cascade of rollbacks is called the *domino effect* [Ran75].

The domino effect can be prevented if processes coordinate their checkpoints, rather than checkpointing their states independently. Through coordination, it is possible to guarantee that a set of local checkpoints will indeed constitute a consistent global state [CL85]. Coordination, however, comes at a cost in an asynchronous system: since processes cannot rely on a common clock to coordinate, they need to exchange messages, at a considerable overhead for environments in which communication is expensive [BLL90].

The original motivation for message logging was precisely to avoid the overhead of checkpoint coordination while still preventing the domino effect. Message logging achieves this result by requiring each process to save on stable storage some additional data collected between taking consecutive independent checkpoints.

A second motivation for message logging was to reduce overhead associated with the application communicating with the environment. Before an application can send output to the environment, a special protocol, commonly called *output commit protocol*, is run to ensure that, in case of a failure, the application will never roll back to a state prior to the one in which the output was produced. Following [EZ94], we call the time necessary to execute such a protocol the *output latency*. In the absence of message logging, the output commit protocol requires a coordinated checkpoint. With message logging, the output commit protocol requires only that the process communicating with the environment save a small portion of its local state on stable storage. This weaker requirement results in potentially much lower output latency.

Recently, Elnozahy and Zwaenepoel have experimentally compared the performance of message logging and coordinated checkpointing for compute-intensive applications running on networks of workstations [EZ94]. Their data confirm that message logging substantially outperforms coordinated checkpointing in output latency. However, their work also suggests that the cost of coordinating checkpoints is comparable with the overhead incurred in maintaining the additional information required by message logging. Message logging appears, therefore, to be ideally suited for applications that require substantial communication with the environment.

3.2 Consistency in Message-Logging Protocols

When temporal redundancy is used, upon recovery the application must be in a consistent state. In a distributed system, this means that, upon recovery, the application processes need to be in mutually consistent states: for each message m delivered by an application process q, there must be an application process p that, in a past state, sent m to q.

When a process fails and recovers, mutual consistency may not hold. Consider again the execution in Figure 3.1. When p_2 fails, state stored in its volatile memory is lost. Process p_2 is rolled back to a state prior to the one in which it sent message

m_5 to p_1, and mutual consistency between the states of p_1 and p_2 is lost. If during recovery p_2 cannot be restored to a state following the send of message m_5, then m_5 is called an *orphan message*, and process p_1, which delivered m_5, is called an *orphan process*. Consider a message m sent by a process q that subsequently fails. Assume that q cannot be recovered to the state s in which m was sent. Any process in a state s' such that $s \rightarrow s'$ is an orphan.

The consistency condition required by fault-tolerant schemes based on temporal redundancy is, therefore, the following:

> In the global state from which the application is resumed after a failure, no process is an orphan process.

We call this condition the *eventually-no-orphans consistency condition.*

In the case of Figure 3.1, eventually-no-orphans consistency condition can be satisfied by rolling p_1 back to checkpoint C_{12}. As we saw in the previous section, however, without checkpoint coordination, it may be impossible to roll back the application processes to a non-initial global state in which no process is an orphan.

Message logging offers an alternative way of satisfying the no-orphan consistency condition. Coordinated checkpointing avoids orphans by rolling back the state of all the application processes to a consistent global state; message logging instead attempts to save information that a failed process can use during recovery to reenter states that were entered before the failure. In the case of Figure 3.1, for example, a message-logging protocol could save the information necessary to ensure that p_2 can be recovered to the state in which m_5 was sent and therefore the protocol would prevent p_1 from becoming an orphan process.

3.3 Recovery and Non-Determinism

To determine which information must be saved to satisfy the no-orphan consistency condition, consider applications α and α'. Assume that the processes executing α are deterministic but that the processes executing α' make non-deterministic choices. Let ρ be a run of α, and let ρ' be a run of α'. To simplify the discussion and without loss of generality, assume that no checkpoints are taken during ρ or ρ'.

We want to determine what information must be saved during ρ and ρ' in order to recover α and α' to a consistent state after a failure.

First, consider α. Since no process executing α makes non-deterministic choices in ρ, if processes do not read input from the environment, then it is not necessary to save any information about what occured during ρ to recover α to a consistent state after a failure. The processes of α, restarted from their initial states, will deterministically reproduce ρ. All messages exchanged during ρ, and in particular all messages directed to the environment, will be regenerated, and α will be restored to a consistent state. Note that if processes do read from the environment, then any data read during ρ must be saved, since it is needed to reproduce ρ during recovery.

Now, consider α'. Since processes make nondeterministic choices in ρ', it is no longer sufficient to restart the processes from their initial states to ensure that ρ' will be reproduced. Because processes may not repeat the non-deterministic choices they made during ρ', some of the messages exchanged during ρ', and in particular some of the messages directed to the environment, may not be regenerated. Unless we can reproduce ρ' during recovery, α' may not be restored to a state consistent with the state of the environment. Therefore, enough information must be saved about each non-deterministic event e to guarantee that, after executing e during recovery, α' will again enter the same state it entered during the original execution. We call such information the *determinant* of e.

Message-logging protocols save determinants for non-deterministic choices made and use these determinants during recovery. Traditionally, message-logging protocols are designed under the assumption that a piecewise deterministic process makes only one kind of non-deterministic choice: a process non-deterministically selects, among the received messages, the next one to deliver. Hence, message-logging protocols save the information necessary to guarantee that each process, during recovery, will deliver the same sequence of messages originally delivered. To characterize this information precisely, we introduce the following notation.

For each message m delivered during an execution, 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 delivery of message m by process $m.dest$. The tuple $\langle m.source, m.ssn, m.dest, m.rsn \rangle$ determines m and the order in which m was delivered by $m.dest$ relative to the other messages delivered by $m.dest.$. Note that the text $m.data$ of m is not part of the tuple. [1] We call this tuple the *determinant* of event $deliver_{m.dest}(m)$, and we denote the tuple as $\#m$.

[1]If $deliver_{m.dest}(m)$ refers to a message m read from the environment, then the tuple must also include $m.data$.

For simplicity, we often refer to $\#m$ as the determinant of m.

If the determinants of all the messages delivered during ρ' are available during recovery, then ρ' can be reproduced during recovery. This observation leads to an interesting and somewhat surprising conclusion: message-logging protocols do not really need to log messages. That is, in principle message-logging protocols do not need to save the contents $m.data$ of a message m delivered during ρ' — except for messages read from the environment — in order to guarantee that m will be regenerated and delivered at the correct point during recovery. This is because, since execution during recovery will reproduce ρ', any message m that was sent and delivered during ρ' will be sent again during recovery. Hence, message-logging protocols need only log message determinants. We use this insight in Chapter 4, when we derive a specification for message logging.

Of course, even if the messages delivered during ρ' can be regenerated by restarting all the processes from their initial states, having the message contents quickly available for replay can speed up recovery. In practice, message-logging protocols cache the contents of messages exchanged during a run. In Section 3.5, we discuss different techniques to implement such caching.

3.4 Structure of Message-Logging Protocols

Message-logging protocols employ two fundamental components:

1. A *logging component*, which is responsible for saving determinants, caching messages, and checkpointing process states.

2. A *recovery component*, which uses the information saved by the logging component to recover the application to a state that satisfies the no-orphan consistency condition.

Checkpoints can be independent [SY85,KT87,SW89,JZ87,SBY88,JV87,JZ90, Joh93,WF92,EZ92,VJ94] or coordinated [EZ94].
The storage abstraction implemented by the logging component is called a *log*. To emphasize the fundamentally different roles played by message determinants and message contents, we explicitly distinguish between storing messages and storing determinants. We will therefore talk of a *determinant log*, distinct from a *message log*. The action of saving data to either log is called *logging*.

The recovery component is invoked after a process p fails. Recovery typically involves three steps:

1. Create a new instance of process p.

2. Initialize p to the latest checkpointed state, and restart p.

3. Replay to p the messages p originally delivered after the latest checkpoint, according to the order specified by the corresponding message determinants.

We assume the existence of suitable mechanisms to implement Steps 1 and 2 [EJZ92], and concentrate on the issues that arise in the implementation of Step 3.

The logging and recovery components are implemented by a layer of software that resides between the application processes and the rest of the system. This layer intercepts send and deliver operations, so that applications run transparently on top of message logging.

In the next section, we look at the most significant issues that arise in the design and implementation of the logging and recovery components.

3.5 Implementing Message-Logging Protocols

Message-logging protocols must balance two conflicting goals:

1. To minimize the overhead added by the logging component during failure-free periods.

2. To give the recovery component the information necessary to recover quickly from a failure.

Different message-logging protocols emphasize one or the other of these goals. The balance that is chosen affects:

- the implementation of the log and

- the policy used to save message determinants in the log.

We discuss these two aspects below.

3.5.1 Implementing the log

Since determinants must be logged but messages themselves need not be logged, the implementation of the determinant log can be quite different from the implementation of the message log.

The determinant log must be available during recovery. Therefore, the determinant log must be implemented using stable storage. The implementation of stable storage depends in turn on the number of failures that a protocol is designed to mask.

For example, if a protocol is designed under the assumption that no more than one process will fail at any time, then stable storage can be implemented by keeping a copy of each determinant in the volatile memory of two processes. This implementation of stable storage is used in *sender-based* protocols, where the determinant of a message m is maintained in volatile memory by process $m.dest$ and process $m.source$ [JZ87,JV87]. The advantage of sender-based protocols is that stable storage is fast and inexpensive; the disadvantage is that at most one faulty process can be tolerated. One of the results of this dissertation is to show that it is possible to implement stable storage efficiently using volatile memory and tolerate an arbitrary number of failures. A similar technique is independently developed in [Eln93].

If a protocol is designed to tolerate an arbitrary number of failures, then stable storage is traditionally implemented using stable memory, such as replicated disks or battery-backed RAM. In this case, the responsibility for logging the determinant of a message m rests with the process that delivers m because that process determines the value of $m.rsn$. Protocols based on this implementation are called *receiver-based protocols* [BBG83,PP83,SY85,KT87,SW89,SBY88,JZ90,VJ94]. The advantage of receiver-based protocols is that they can tolerate an arbitrary number of failures; the disadvantage is that either special hardware is used or the cost of accessing stable storage will be comparatively high.

As noted earlier, the message log does not need to be available during recovery. Therefore, it is not necessary to save the message log on stable storage. Most protocols, however, do so in order to expedite recovery [SY85,SW89,SBY88,PP83, SBY88,JZ90,VJ94]. The disadvantage is added overhead. An alternative scheme is to save the message log in the volatile memory of the senders [JZ87,JV87,SBY88, EZ92]. In this scheme, if the process $m.dest$ that delivered m fails, then $m.dest$ requests $m.source$ to replay m's contents during recovery.

The advantage of saving the message log in the volatile memory of the sender is that it considerably reduces overhead, since the log is stored in volatile memory. The disadvantage is that, if $m.source$ and $m.dest$ fail at the same time, then the message log entry for m will be lost, and it will not be possible to replay m to $m.dest$ quickly. However, if the determinants of the messages delivered by $m.source$ before sending m are available during recovery, m will be regenerated during the recovery of $m.source$, and it will eventually be replayed to $m.dest$. Hence, logging a single copy of m in the volatile memory of $m.source$ is prudent. A scheme that stores the message log on volatile memory has been shown experimentally to outperform significantly a scheme that saves the message log on stable storage. [EZ94]

3.5.2 Choosing a logging policy

The policy used to log determinants affects how the application is restarted upon recovery of the failed processes. Recall, no process may be an orphan in the global state from which the application is resumed after a failure. This condition can be enforced in one of two ways: a protocol can either prevent the creation of orphans or it can take action during recovery to eliminate orphans created by a failure. The corresponding logging policies, and the protocols that implement them, are called *pessimistic* and *optimistic*, respectively.

Pessimistic logging protocols [BBG83,PP83,JZ87,SBY88] synchronize logging with communication. They require that each process log determinants for all messages it has delivered before sending a message. This policy ensures that if a process p fails, then the determinants of all the messages delivered by p are available during recovery. Therefore, p will be able to replay the execution it followed before failing and, in particular, p will reenter all the states from which it sent messages before failing. Hence, there can be no orphan messages or orphan processes.

The problem with pessimistic protocols is that they may introduce blocking even in runs that experience no failures: before sending a message, a process p must wait until the determinants of all the messages it has delivered are logged. Considering that pessimistic protocols typically use replicated disks to implement stable storage, such blocking may unacceptably slow down system performance.

Optimistic logging protocols [SY85, KT87, JZ87, SBY88, SW89, JV87, JZ90, WF92,VJ94], in contrast, do not synchronize logging and communication. They allow processes to send messages even when the determinants of previously delivered messages are not yet logged. Optimistic protocols only require that determinants be logged eventually. Since these protocols do not introduce blocking, optimistic protocols can perform efficiently in failure-free executions.

The problem with optimistic protocols is that they have more complex recovery protocols. If determinants of some messages delivered by p are not logged when p fails, then it may not be possible to regenerate some of the messages previously sent by p. This causes otherwise correct processes to become orphans; these orphans must be identified and rolled back to a state prior to the delivery of any of the orphan messages. Furthermore, optimistic protocols must take appropriate measures to avoid the domino effect.

Chapter 4

Specification of Message Logging

Chapter 3 introduced the eventually-no-orphans consistency condition and discussed how pessimistic and optimistic protocols preserve it in the presence of failures. Neither solution, as we saw in Section 3.5.2, is truly satisfactory. Our approach for deriving new and more efficient solutions is to examine more carefully the notion of an orphan and to specify the *always-no-orphans* condition, which guarantees that no orphans are created during an execution and therefore implies the eventually-no-orphans consistency condition.

In Section 4.1 we present a formal specification of the orphan-free condition. In Section 4.2, we relate pessimistic and optimistic protocols to our specification. We show that pessimistic protocols implement a condition that is stronger than what the specification requires, while optimistic protocols implement a condition that is weaker than what the specification requires. In Section 4.3, we derive from our specification a new message-logging scheme that we call *causal message logging.*

Efficient implementation of causal message logging is the topic of Chapters 5 and 6. Finally, in Section 4.4 we extend our specification to include the case in which a single *logging site* is used to maintain logs of multiple processes. Such a situation arises when several processes share a processor.

4.1 The Always-No-Orphans Condition

Consider a set $\mathcal{N}$ of n processes. We want to derive a condition, which we call *always-no-orphans*, that guarantees that no orphans are generated over a run of $\mathcal{N}$. In order to do so, we define two subsets of $\mathcal{N}$. The first set, $\mathit{Depend}(m)$, contains all processes whose state reflects delivery of an application message m: $\mathit{Depend}(m)$ contains the destination of message m, plus any process that delivered an application message sent causally after the delivery of m. Formally,

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

The second set, $\mathit{Log}(m)$, is used to characterize the set of processes that have a copy of $\#m$ in their volatile memory. One can imagine protocols where no single process knows the value of $\#m$ but a set of processes collectively do. An example of such a protocol is given in [JV87], where the value of $\#m$ for some message m may be inferred by "holes" in the sequence of logged receive sequence numbers. Since this is the only such case that we know of and it can tolerate at most two process failures at a time, we do not consider further this more general method of logging. Process $m.\mathit{dest}$ becomes a member of $\mathit{Log}(m)$ when it delivers m. Suppose that a set $\mathcal{F}$ of processes fails, where $\mathcal{F} \subseteq \mathcal{N}$ holds. The determinant of a deliver event

$deliver_{m.dest}(m)$ is lost if $Log(m) \subseteq \mathcal{F}$. Assume that $\#m$ is lost. By definition, process $m.dest$ was in $Log(m)$ before the failure. Therefore, $m.dest$ must be in $\mathcal{F}$, hence $m.dest$ will need to be recovered. Since $\#m$ is lost, $m.dest$ may not be able to deliver m in the correct order during recovery. Consequently, as we saw in Section 3.2, $m.dest$ may not be able to regenerate some of the messages it sent during the original execution. All processes whose state depends on process $m.dest$ delivering message m in the correct order will become orphan processes.

We say that a process p becomes an orphan of a set $\mathcal{F}$ of processes that fail when p itself does not fail and p's state depends on the delivery of a message m whose determinant has been lost. Formally:

$$p \ orphan \ of \ \mathcal{F} \stackrel{\text{def}}{=} \begin{pmatrix} \wedge & (p \in \mathcal{N} - \mathcal{F}) \\ \wedge & \exists m : ((p \in Depend(m)) \wedge (Log(m) \subseteq \mathcal{F})) \end{pmatrix} \tag{4.1}$$

Negating (4.1) and quantifying over p gives the following necessary and sufficient condition for there being no orphans created by the failure of a set of processes $\mathcal{F}$:

$$\forall m : ((Log(m) \subseteq \mathcal{F}) \Rightarrow (Depend(m) \subseteq \mathcal{F})) \tag{4.2}$$

The derivation of (4.2) is given in Figure 4.1.

Our goal is to derive a property that guarantees that no set $\mathcal{F}$ of faulty processes results in the creation of orphans. Quantifying (4.2) over all $\mathcal{F}$, we obtain:

$$\forall m : (Depend(m) \subseteq Log(m)) \tag{4.3}$$

Since we want (4.3) to hold for every state, we require the following property

$$\forall m : \Box(Depend(m) \subseteq Log(m)) \tag{4.4}$$

$$\forall p : \neg(p\ \mathit{orphan\ of}\ \mathcal{F})$$

$$= \langle\langle \text{Definition 4.1 of } p\ \mathit{orphan\ of}\ \mathcal{F} \rangle\rangle$$

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

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

$$\forall p : \begin{pmatrix} \vee & \neg(p \in \mathcal{N} - \mathcal{F}) \\ \vee & \neg(\exists m : ((p \in \mathit{Depend}(m)) \wedge (\mathit{Log}(m) \subseteq \mathcal{F}))) \end{pmatrix}$$

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

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

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

$$\forall p, m : ((p \in \mathcal{F}) \vee \neg(p \in \mathit{Depend}(m)) \vee \neg(\mathit{Log}(m) \subseteq \mathcal{F}))$$

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

$$\forall p, m : ((\mathit{Log}(m) \subseteq \mathcal{F}) \Rightarrow (\neg(p \in \mathit{Depend}(m)) \vee (p \in \mathcal{F})))$$

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

$$\forall p, m : ((\mathit{Log}(m) \subseteq \mathcal{F}) \Rightarrow ((p \in \mathit{Depend}(m)) \Rightarrow (p \in \mathcal{F})))$$

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

$$\forall m : ((\mathit{Log}(m) \subseteq \mathcal{F}) \Rightarrow (\mathit{Depend}(m) \subseteq \mathcal{F}))$$

Figure 4.1: Derivation of (4.2).

where $\Box$ is the temporal "always" operator [Pnu77]. Property (4.4) is a safety property: it must always hold during an execution in order to ensure that no orphans will be created. It states that, to avoid orphans, it is sufficient to guarantee that if the state of a process p depends on the delivery of message m, then p must keep a copy of m's determinant in its volatile memory.

We say that $\#m$ is *stable* (denoted $stable(m)$) when $\#m$ cannot be lost. Condition 4.3 must hold only for messages with a determinant that is not stable. Hence 4.3 need only hold when $\#m$ is not stable:

$$\forall m : \Box(\neg stable(m) \Rightarrow (Depend(m) \subseteq Log(m))) \tag{4.5}$$

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 — where $|\mathcal{F}| \leq f$ — can fail concurrently, then $stable(m)$ holds as long as $f+1$ processes have a copy of $\#m$ in their volatile memory. In the latter case, (4.5) can be written:

$$\forall m : \Box((|Log(m)| \leq f) \Rightarrow (Depend(m) \subseteq Log(m))) \tag{4.6}$$

4.2 Pessimistic and Optimistic Protocols Revisited

As we saw in Section 3.5, pessimistic message-logging protocols do not create orphans. Therefore, they must implement Property (4.6). Indeed, they implement the following stronger property:

$$\forall m : \Box(\neg stable(m) \Rightarrow (|Depend(m)| \leq 1)) \tag{4.7}$$

Before discussing the meaning of (4.7), we show that (4.7) implies (4.5).

Theorem 4.1 *Property (4.7) implies Property (4.6).*

Proof.

⟪Hypothesis assumed⟫

1. $\forall m : \Box(\neg \mathit{stable}(m) \;\Rightarrow (|\mathit{Depend}(m)| \leq 1))$

⟪Definition of $\mathit{Depend}(m)$⟫

2. $\forall m : \Box((|\mathit{Depend}(m)| \leq 1) \Rightarrow (\mathit{Depend}(m) \subseteq \{m.\mathit{dest}\}))$

⟪Definition of $\mathit{Log}(m)$⟫

3. $\forall m : (\{m.\mathit{dest}\} \subseteq \mathit{Log}(m))$

⟪Transitivity of Set Inclusion, using 2 and 3⟫

4. $\forall m : \Box(|\mathit{Depend}(m)| \leq 1) \Rightarrow (\mathit{Depend}(m) \subseteq \mathit{Log}(m))$

⟪Transitivity of Implication, using 1 and 4⟫

5. $\forall m : \Box(\neg \mathit{stable}(m) \;\Rightarrow (\mathit{Depend}(m) \subseteq \mathit{Log}(m)))$

□

In practice, (4.7) does not allow the process that delivers message m to send any messages until the determinant of m is stable.[1] To see this, suppose process p_1 has just delivered m. $\mathit{Depend}(m)$ contains only p_1, so $|\mathit{Depend}(m)| = 1$. Suppose now that p_1 sends a message m' to process p_2. Since $\mathit{deliver}_{p_1}(m) \rightarrow \mathit{deliver}_{p_2}(m')$, when process p_2 delivers m' it becomes a member of $\mathit{Depend}(m)$, and $|\mathit{Depend}(m)|$

[1] In pessimistic sender-based logging [JZ87] process $m.\mathit{dest}$ increases $|\mathit{Log}(m)|$ by sending the value of $m.\mathit{rsn}$ to process $m.\mathit{source}$, piggybacked on the acknowledgment of message m. Notice however that $m.\mathit{dest}$ is still not allowed to send any *application* messages until it is certain that $m.\mathit{source}$ has become a member of $\mathit{Log}(m)$.

= 2. Thus, as soon as m' is sent, the consequent in (4.7) can become false. The only way p_1 can preserve (4.7) is to ensure that $|Log(m)| > f$ holds before sending m'. If the time during which $|Log(m)| \leq f$ is brief, then it is unlikely that p will attempt to send a message while $\#m$ is not stable, so any performance loss by inhibiting message sends is small.

The reasoning behind optimistic protocols starts from the same assumption used by pessimistic protocols: the time during which $|Log(m)| \leq f$ holds is brief. Hence, (4.3) holds trivially nearly all the time. Therefore, it is not practical to incur the performance cost that pessimistic protocols suffer. Instead, optimistic protocols take appropriate actions during recovery to re-establish (4.3) in the unlikely event that it is violated as a result of the failure of a set of processes $\mathcal{F}$.

Optimistic protocols implement the following property:

$$\forall m : \Box(\neg stable(m) \;\Rightarrow ((Log(m) \subseteq \mathcal{F}) \Rightarrow \Diamond(Depend(m) \subseteq \mathcal{F}))) \qquad (4.8)$$

where $\Diamond$ is the temporal "eventually" operator [Pnu77].

Property (4.8) is weaker than Property (4.2), and therefore weaker than Property (4.6). Property (4.8) permits the temporary creation of orphan processes, but guarantees that, by the time recovery is complete, no surviving process will be an orphan and Property (4.2) will hold. This is achieved during recovery by rolling back orphan processes until their states do not depend on any message whose determinant has been lost. In other words, $Depend(m)$ is made smaller (or, equivalently, $\mathcal{F}$ is made larger) until $Depend(m) \subseteq \mathcal{F}$ and Property (4.2) is restored.

4.3 Causal Message-Logging Protocols

Pessimistic protocols implement a property stronger than Property (4.4) and, therefore, never create orphans. However, in order to implement this stronger property, blocking in failure-free runs may be introduced. Optimistic protocols avoid this blocking by satisfying a property weaker than Property (4.4). It is natural to ask whether a protocol can be designed that implements Property (4.4) — and therefore creates no orphans — yet does not implement a property as strong as Property (4.6) — and thus does not introduce any blocking. We consider this question in Chapter 5. Before doing so, though, we refine Property (4.6).

Consider the following property:

$$\forall m : \Box((|\mathit{Log}(m)| \leq f) \Rightarrow (\mathit{Depend}(m) = \mathit{Log}(m))) \tag{4.9}$$

Property (4.9) strengthens (4.6), and so protocols that implements (4.9) prevents orphans. Furthermore, such protocols disseminate the least number of copies of $\#m$ needed in order to satisfy (4.6), thereby conserving storage and network bandwidth.

Unfortunately, satisfying Property (4.9) requires processes to be added to $\mathit{Log}(m)$ and $\mathit{Depend}(m)$ simultaneously. Satisfying this requirement would result in complicated protocols. Thus, we consider a different strengthening of Property (4.6), resulting in a property that is weaker than Property (4.9) but still bounds $\mathit{Log}(m)$:

$$\forall m : \Box \left(\begin{array}{c} (|Log(m)| \leq f) \Rightarrow \\ \left(\begin{array}{ll} \wedge & (Depend(m) \subseteq Log(m)) \\ \wedge & \Diamond(Depend(m) = Log(m)) \end{array} \right) \end{array} \right) \tag{4.10}$$

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

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

We call the protocols that implement Property (4.10) *causal* message-logging protocols.

4.4 Sharing the Log

The specification derived above assumes that a process logs determinants either in its own volatile memory or to stable memory. Hence, determinants may be lost only when a process fails, and since processes fail independently, different copies of the same determinant are lost independently. A more general approach is to regard determinant logs as being first-class objects. By doing so, one can model a set of processes sharing storage for logging purposes and thereby decouple the failure of processes from the loss of determinants. For example, one reasonable approach would be to implement the logging component of a message-logging protocol so that a single, shared log is maintained for all the processes that run on the same processor. As we discuss in Section 6.6.1, this approach can result in a more efficient implementation of causal message logging.

Define a *logging site* to be a storage object that a process can read and write. For this derivation, we assume that logging sites do not have stable memory. Like processes, logging sites fail and recover. When a logging site fails, the values written to the logging site are lost. Each process uses a logging site for its determinant log. When a logging site fails, all processes using that logging site also fail. A process may fail without its associated logging site failing, however.

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

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

$$L^{\cup}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcup_{p \in \mathcal{P}} L(p) \tag{4.11}$$

$$P^{\cup}(\mathcal{S}) \stackrel{\text{def}}{=} \bigcup_{\ell \in \mathcal{S}} P(\ell) \tag{4.12}$$

We now repeat, under these new assumptions, the derivation that in Section 4.1 led us to Property (4.6), which enforces the always-no-orphans consistency condition.

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

$$p \ orphan\ of\ \mathcal{F}, \mathcal{E} \stackrel{\mathrm{def}}{=} \begin{pmatrix} \wedge & (p \in \mathcal{N} - \mathcal{F}) \\ \wedge & \exists m : ((p \in \mathit{Depend}(m)) \wedge (\mathit{Log}(m) \subseteq \mathcal{E})) \end{pmatrix} \tag{4.13}$$

A logging site must fail for a process to become an orphan. Negating (4.13) and quantifying over all p gives the following:

$$\forall m : ((\mathit{Log}(m) \subseteq \mathcal{E}) \Rightarrow (\mathit{Depend}(m) \subseteq \mathcal{F})) \tag{4.14}$$

Property (4.14) is necessary and sufficient to guarantee that the failure of $\mathcal{F}$ processes and of $\mathcal{E}$ logging sites creates no orphans. From definitions (4.11) and (4.12), it follows that:

$$\mathcal{A} \subseteq \mathcal{B} \quad \Rightarrow \quad P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B}) \tag{4.15}$$

$$\mathcal{A} \subseteq \mathcal{B} \quad \Rightarrow \quad L^{\cup}(\mathcal{A}) \subseteq L^{\cup}(\mathcal{B}) \tag{4.16}$$

where $\mathcal{A}$ and $\mathcal{B}$ are sets of the appropriate type. Noting that $L^{\cup}(P^{\cup}(\mathcal{S})) = \mathcal{S}$, it is straightforward to show (see Figure 4.2) that $\mathcal{A} \subseteq \mathcal{B} \equiv P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B})$. Hence, we can replace the antecedent of (4.14) with the equivalent expression $P^{\cup}(\mathit{Log}(m)) \subseteq P^{\cup}(\mathcal{E})$:

$$\forall m : ((P^{\cup}(\mathit{Log}(m)) \subseteq P^{\cup}(\mathcal{E})) \Rightarrow (\mathit{Depend}(m) \subseteq \mathcal{F})) \tag{4.17}$$

By assumption, $\mathcal{E}$ and $\mathcal{F}$ are constrained: $P^{\cup}(\mathcal{E}) \subseteq \mathcal{F}$. Hence, the following strengthens (4.17):

$$\forall m : ((P^{\cup}(\mathit{Log}(m)) \subseteq \mathcal{F}) \Rightarrow (\mathit{Depend}(m) \subseteq \mathcal{F})) \tag{4.18}$$

Proof that $\mathcal{A} \subseteq \mathcal{B} \Rightarrow P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B})$:

$\langle\langle$Formula (4.15)$\rangle\rangle$

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

Proof that $P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B}) \Rightarrow \mathcal{A} \subseteq \mathcal{B}$:

$\langle\langle$Hypothesis assumed$\rangle\rangle$

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

   $\langle\langle$Modus ponens, using 1 and (4.16)$\rangle\rangle$

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

   $\langle\langle L^{\cup}(P^{\cup}(\mathcal{S})) = \mathcal{S}$ and 2$\rangle\rangle$

3. $\mathcal{A} \subseteq \mathcal{B}$ $\Box$

Figure 4.2: $\mathcal{A} \subseteq \mathcal{B} \equiv P^{\cup}(\mathcal{A}) \subseteq P^{\cup}(\mathcal{B})$.

Universally quantifying (4.18) over $\mathcal{F}$ we obtain:

$$\forall m : (\mathit{Depend}(m) \subseteq P^{\cup}(\mathit{Log}(m))) \tag{4.19}$$

Since we want (4.19) to hold in every state, we obtain the following property:

$$\forall m : \Box(\mathit{Depend}(m) \subseteq P^{\cup}(\mathit{Log}(m))) \tag{4.20}$$

Property (4.20), like Property (4.4) of Section 4.1, is a safety property. If satisfied, (4.4) guarantees that no orphans will be created during a run.

If it is acceptable to make assumptions about the maximum number of concurrent failures during a run, then Property (4.20) can be weakened. To do so, it suffices to consider the maximum number of concurrent failures of logging sites, because orphan processes are directly created by logging site failures. Assuming that no more than $f_{\mathcal{E}}$ logging sites can fail at any time, once $\#m$ has been logged at more than $f_{\mathcal{E}}$ logging sites, $\#m$ is stable. Hence, Property (4.20) need hold only as long as failures cannot cause $\#m$ to be lost:

$$\forall m : \Box((|Log(m)| \leq f_{\mathcal{E}}) \Rightarrow ((Depend(m) \subseteq P^{\cup}(Log(m))) \tag{4.21}$$

Finally, we can strengthen Property (4.21) as in Section 4.3, to obtain the following property, which characterizes causal message logging when there are shared logging sites:

$$\forall m : \Box \left(\begin{array}{c} (|Log(m)| \leq f_{\mathcal{E}}) \Rightarrow \\ \left(\begin{array}{ll} \wedge & (Depend(m) \subseteq P^{\cup}(Log(m))) \\ \wedge & \Diamond(Depend(m) = P^{\cup}(Log(m))) \end{array} \right) \end{array} \right) \tag{4.22}$$

Chapter 5

Optimal Message Logging Protocols

In this chapter, we begin to explore the performance of causal message-logging protocols. To do so, we propose in Section 5.1 a set of metrics for evaluating the performance of message-logging protocols, and we characterize protocols that are *optimal* with respect to these metrics. In Sections 5.2 and 5.3 we show how to implement an optimal message-logging protocol using causal message logging. Finally, in Section 5.4, we describe existing optimal message-logging protocols.

5.1 Performance Metrics

Let Π be a protocol executed by a set $\mathcal{N}$ of processes, and assume that Π is written to tolerate no process failures and no transient channel failures. Let Π_μ denote Π combined with a message-logging protocol μ that tolerates process failures and transient channel failures. We consider the following four metrics for comparing message-logging protocols:

1. *Number of forced roll-backs*: Consider a run ρ of Π_μ, and suppose that a set $\mathcal{F} \subseteq \mathcal{N}$ of processes fails. We say that μ forces r roll-backs if, for all ρ, μ requires at most r correct processes to roll-back their state. Clearly, a lower bound for r is 0; furthermore, the existence of pessimistic protocols, where no correct process ever rolls-back, establishes that this bound is tight.

2. *k–blocking*: Consider a message m delivered by process $m.dest$, and let e be the first send event of process $m.dest$ that causally follows $deliver_{m.dest}(m)$. We say that a message-logging protocol is *k–blocking* if, in all failure-free runs and for all messages m, process $m.dest$ delivers no less than k messages between $deliver_{m.dest}(m)$ and e. For example, pessimistic sender-based logging [JZ87] is 1–blocking because process $m.dest$ must receive a message acknowledging the logging of $m.rsn$ before sending a message subsequent to the delivery of m. Optimistic protocols are, by design, 0–blocking. For completeness, we define pessimistic protocols that log the determinant of m in local stable storage to be 1–blocking, since there is one event (the acknowledgment of $\#m$ being written to stable storage) that must occur between $deliver_{m.dest}(m)$ and a subsequent send [BBG83]. Clearly, a lower bound for k is 0; furthermore, the existence of optimistic protocols establishes that this bound is tight.

3. *Number of messages*: Protocol Π can be transformed into a protocol Π_ℓ that tolerates transient channel failures by a simple acknowledgment scheme: whenever process $m.dest$ receives a message m generated by Π, it sends an acknowledgment to process $m.source$. Process $m.source$ repeatedly sends m until it receives such an acknowledgment. For any failure-free run ρ, protocol Π_ℓ will send at least twice the number of messages that Π sends.

   Suppose now that ρ is run using Π_μ instead of Π_ℓ. We say that μ sends additional messages in ρ if Π_μ sends more messages than Π_ℓ. For example, in order to tolerate single crash failures, pessimistic sender-based logging [JZ87] potentially requires that one extra acknowledgment be sent for each application message sent. Clearly, a lower bound is to send no additional messages, and the existence of optimistic protocols establishes that this bound is tight.

4. *Size of messages*: Consider any message m sent in a run of Π_ℓ, and let m_μ be the corresponding message sent in the equivalent failure-free run of Π_μ. Let $|m|$ and $|m_\mu|$ be the size of m and m_μ, respectively. We will say that μ sends a additional data if the maximum value of $|m_\mu| - |m|$ is a for all such m and m_μ taken from all pairs of corresponding runs. For example, for optimistic protocols that track direct dependencies, a is a constant [SW89,VJ94]; while for optimistic protocols that track transitive dependencies a is proportional to the number of processes in the system [SY85,SW89]. Clearly, a lower bound for a is 0, and the existence of pessimistic receiver-based logging proves that this bound is tight.

There is a tradeoff between the number of additional messages sent in Π_μ and the sizes of messages — a message-logging protocol μ could include less information in a message by sending additional messages containing the extra data. Since there is, to a point, a performance benefit in sending a few large messages instead of several small messages, we will prefer to keep the number of additional messages small at the expense of message size. Hence, we define an *optimal message-logging protocol* as a message-logging protocol that is 0–blocking, introduces 0 forced roll-backs, and sends no additional messages. Notice, protocols that are optimal according to our definition do not necessarily outperform in practice non-optimal protocols. For instance, a message may become large enough that the underlying network will need to break it into separate packets, thereby significantly increasing the overhead of the protocol. In addition, other issues that are difficult to quantify, such as the cost of output commit, must be taken into consideration in assessing the performance of a message-logging protocol [EZ94]. Nevertheless, the protocols we call optimal are unique in optimally addressing the theoretical *desiderata* of the message-logging approach.

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

We now show how to derive an optimal causal message-logging protocol, i.e. an optimal message-logging protocol that implements Property (4.10). We do so by first presenting, in this section, a simple non-optimal protocol that uses causal delivery order to implement the no-orphans consistency condition. In the following, section, we refine this protocol and obtain an optimal causal message-logging protocol. We begin by defining causal delivery order.

5.2.1 Causal Delivery Order

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

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

$$\begin{aligned} \forall p, r, m, m' : send_p(m, r) \rightarrow send_p(m', r) \Rightarrow \\ deliver_r(m) \rightarrow deliver_r(m') \end{aligned} \tag{5.1}$$

FIFO delivery order constrains the order in which the destination process delivers messages m and m' only when (i) m causally precedes m' and (ii) m and m' are sent by the same process. Causal delivery order [BJ87] strengthens FIFO delivery order by removing the requirement that ordering occurs only when the source of m and m' are the same. It guarantees that if (i) the sending of m causally precedes the sending of m' and (ii) m and m' are directed to the same destination process, then the destination process does not deliver m' unless it has previously delivered m. Formally:

$$\begin{aligned} \forall p, q, r, m, m' : send_p(m, r) \rightarrow send_q(m', r) \Rightarrow \\ deliver_r(m) \rightarrow deliver_r(m') \end{aligned} \tag{5.2}$$

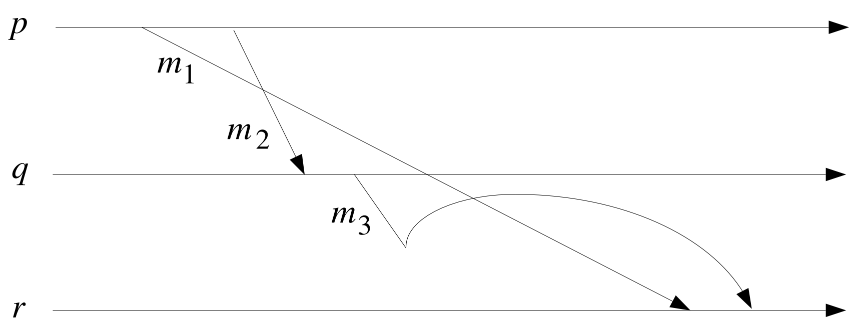

Figure 5.1: An example of causal delivery order

An example of an execution where process r delivers messages according to causal delivery order is shown in Figure 5.1. Note that

$$send_p(m_1, r) \rightarrow send_q(m_3, r)$$

because:

- $send_p(m_1, r) \rightarrow send_p(m_2, q)$.
- $send_p(m_2, q) \rightarrow receive_q(m_2)$.
- $receive_q(m_2) \rightarrow deliver_q(m_2)$.
- $deliver_q(m_2) \rightarrow send_q(m_3, r)$.
- $\rightarrow$ is a transitive relation.

Therefore, r must deliver m_3 only after m_1 has been received and delivered.

There are two fundamental approaches to implementing causal message delivery. The first is to add to each message m additional information [RST91,SS92, BSS91] that m's destination process uses to determine when m can be delivered.

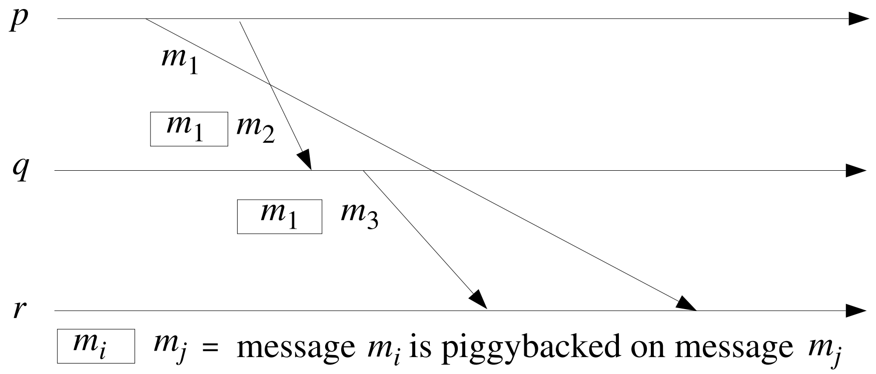

Figure 5.2: Implementing causal delivery order through piggybacking

Using this approach, process r in Figure 5.1 would realize, when it receives m_3, that it must wait to receive m_1, and will delay delivery of m_3 accordingly.

The problem with this approach is that slow messages can significantly affect the performance of the system, since they prevent faster messages from being delivered.

The second approach is for each process p to piggyback, on each message m that p sends, all messages m' sent in the causal history of the event $send_p(m)$ such that p does not know if m' has already been delivered [BJ87]. The piggybacked messages are placed in a total order that extends the partial order imposed by the happens-before relation. Before delivering m, process $m.dest$ first checks if any message m' in m's piggyback has $m.dest$ for destination. If so, p delivers m only after each such message m' has been delivered according to causal delivery order. This is the approach illustrated in Figure 5.2. Message m_1 is piggybacked on m_2 and m_3, since p and q do not know whether m_1 has already been delivered. When r receives m_3, it checks the piggyback to find m_1. Process r therefore delivers m_1

immediately, and then m_3, without waiting for the slow copy of message m_1 to arrive. When the copy of m_1 is received directly from q, r will discard it and will not deliver it.

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

5.2.2 Protocol Outline

We now present a Protocol Π_{cd} that uses causal delivery order to satisfy Property (4.4). Processes in Π_{cd} behave as follows:

(i) Processes exchange two kinds of messages: *application messages* and *determinant messages*. A determinant message contains the determinant of an application message.

(ii) Application and determinant messages are delivered according to causal delivery order.

(iii) Suppose process p receives an application message m. To deliver m, p creates the determinant $\#m$ of message m and logs $\#m$ in its volatile memory. Then, p supplies $m.data$ to the application.

(iv) Suppose process p delivers an application message m. Before sending any subsequent application messages, p sends a determinant message containing $\#m$ to all the other processes.

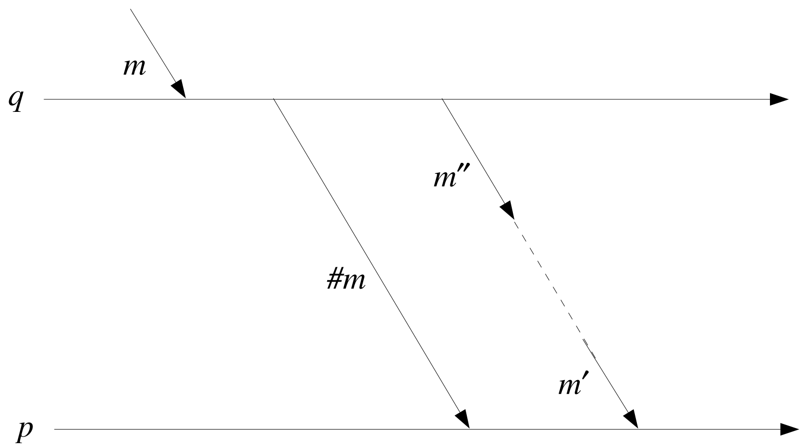

Figure 5.3: *Depend*$(m) \subseteq$ *Log*(m) in Π_{cd}

(v) Suppose process p receives a determinant message containing $\#m$. To deliver the determinant message, p logs $\#m$ in its local volatile memory.

Theorem 5.1 *Protocol Π_{cd} satisfies Property (4.4).*

Proof. Property (4.4) requires that

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

hold throughout execution. According to the definition of *Depend*(m) given in Section 4.1, a process p is a member of *Depend*(m) for an application message m if one of the following two cases holds:

Case 1: p is the destination of m.

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

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

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

Case 2: If p is not the destination of m, then some other process $q \neq p$ delivered m, and p delivered a message m' such that $\mathit{deliver}_q(m) \rightarrow \mathit{deliver}_p(m')$. Furthermore, there must exist an application message m'', not necessarily distinct from m', such that $\mathit{deliver}_q(m) \rightarrow \mathit{send}_q(m'') \rightarrow \mathit{deliver}_p(m')$ (see Figure 5.3). By point (iv) of Π_{cd}, q must have sent a determinant message containing $\#m$ to all processes — including process p — before sending m''. By point (ii) of Π_{cd}, all messages are delivered according to causal delivery: therefore p must have delivered the message containing $\#m$ before delivering m''. It follows that by point (v) of Π_{cd}, p must have logged $\#m$ in its volatile storage before delivering m''. Hence, $p \in \mathit{Log}(m)$ holds. $\Box$

Figure 5.4 shows an execution of Protocol Π_{cd}. Process p_1 sends application message m_1 to process p_3, and then sends application message m_2 to process p_2. After delivering m_2, and before sending application message m_3, p_2 sends to all processes a determinant message containing $\#m_2$. Note that process p_3 first receives m_3, then $\#m_2$, and finally m_1. However, in order to respect causal delivery order, p_3 actually delivers first m_1, then $\#m_2$, and finally m_3.

5.3 An optimal causal protocol

Protocol Π_{cd} has several limitations. First, it is not optimal, since for each deliver event of an application message m, it uses additional determinant messages to send

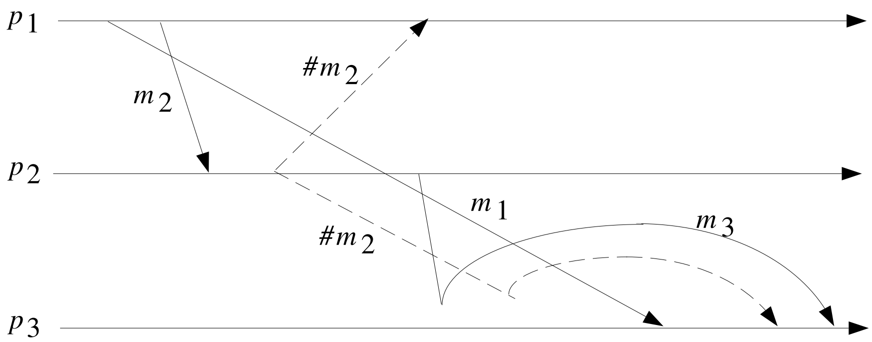

Figure 5.4: An execution of Π_{cd}

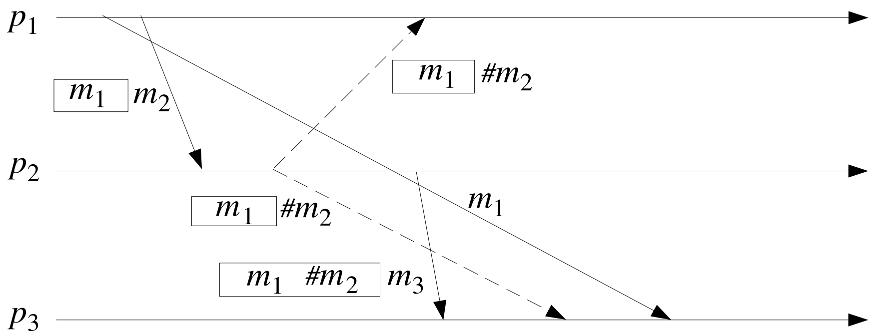

Figure 5.5: Implementing causal delivery order through piggybacking

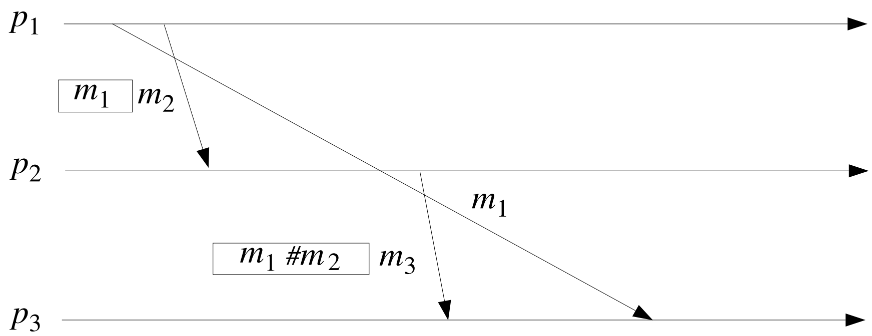

Figure 5.6: Determinants are piggybacked on application messages

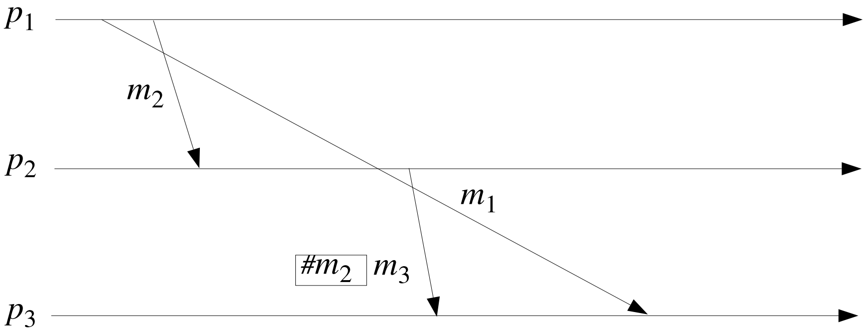

Figure 5.7: An execution of Π_{oc}

$\#m$ to all processes. Second, it does not satisfy Property (4.10), since a process can enter *Log*(m) by receiving a copy of $\#m$ without ever becoming a member of *Depend*(m). Finally, it forces both application and determinant messages to be delivered according to causal delivery order, even though causal delivery order is necessary only to regulate how determinant messages are delivered with respect to application messages.

However, by choosing an appropriate implementation of causal delivery order, protocol Π_{cd} can serve as the starting point for a more efficient protocol.

Since we desire a non-blocking protocol, our first step is to choose an implementation of causal delivery based on the piggybacking scheme described in Section 5.2.1. Figure 5.5 shows the effects of using the piggybacking scheme on the execution in Figure 5.4.

Observe that, since for each application message m the determinant $\#m$ is piggybacked on any application message sent causally after event $deliver_p(m)$, there is no need to explicitly send $\#m$ in a separate determinant message. Instead, we require each process, before delivering a message m, to deliver all the determinants piggybacked on m. The resulting protocol successfully addresses the first two limitations of the original protocol. By piggybacking $\#m$ on application messages, the protocol ensures that no correct processes p will enter $Log(m)$ unless p will eventually join $Depend(m)$. Furthermore, the protocol is optimal in the number of messages it sends because the determinants are piggybacked on existing application messages. Figure 5.6 shows the effect of the new protocol on the execution in Figure 5.5.

Note that, once the piggybacking of the determinants is in place, we are guaranteed that no process will deliver an application message m unless it first delivers the determinants of all the messages delivered in the causal history of $send_{m.source}(m)$. This is precisely the property that we need to guarantee that $\forall m : Depend(m) \subseteq Log(m)$ holds. Therefore, provided that we piggyback the determinants, we can safely relax the requirement that application messages be delivered using causal delivery order. Figure 5.7 shows the effects of the new protocol on the execution in Figure 5.6.

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

(i) Processes exchange only application messages. However, determinants may be piggybacked on application messages.

(ii) Suppose process p sends a message m to process q. p piggybacks on m all the non-stable determinants $\#m'$ such that p has $\#m'$ its volatile memory and $p \in \mathit{Depend}(m')$. Note that, in order to determine which determinants are non-stable, p needs to have instantaneous access to $\mathit{Log}(m)$. This assumption is not realistic in a distributed system, but we use it for now, so that we can concentrate on the remaining aspects of the piggybacking scheme used by Π_{oc}. We abandon this assumption in Chapter 6, when we describe a new class of message-logging protocols derived from Π_{oc}.

(iii) Suppose process p receives a message m. Before delivering m, process p first logs all the determinants piggybacked on m in its determinant log, which is implemented in p's volatile memory. Determinants of events with the same value in the *dest* field are logged in ascending receive sequence number order. Then, p creates the determinant for message m, and logs it in the determinant log. Finally, p delivers m by supplying $m.data$ to the application.

Theorem 5.2 *Protocol Π_{oc} satisfies Property (4.10). Furthermore, Protocol Π_{oc} is optimal.*

Proof. We first prove that Π_{oc} satisfies Property (4.10). In particular, we prove that:

(a) $(|\mathit{Log}(m)| \leq f) \Rightarrow (\mathit{Depend}(m) \subseteq \mathit{Log}(m))$

(b) $(|\mathit{Log}(m)| \leq f) \Rightarrow \Diamond(\mathit{Depend}(m) = \mathit{Log}(m))$

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

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

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

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

Base Case : $\ell = 1$.

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

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

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

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

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

We now prove part (b), assuming $|Log(m)| \leq f$.

If process p is a member of $Log(m)$, then it must have stored a copy of $\#m$ in its determinant log. In order for this to happen, p must have either received m, or it must have received a message m', onto which $\#m$ was piggybacked, such that $deliver_{m.dest}(m) \rightarrow send_{m'.source}(m')$. In the first case, unless p fails, it will eventually deliver m and become a member of $Depend(m)$. The second case is similar: unless p fails, it will eventually deliver m' and become a member of $Depend(m)$. Furthermore, since there is only a finite number of processes in $\mathcal{N}$ and $Log(m) \subseteq \mathcal{N}$, eventually all non-faulty members of $Log(m)$ will join $Depend(m)$ and (b) will hold. Note that, if p fails, then p will no longer be a member of

Depend(m). However, in this case, p will lose all the data in its determinant log, which is implemented in volatile memory, and will therefore leave *Log*(m). Hence, (b) holds whether p is a correct process or not. This concludes the proof that Π_{oc} satisfies Property (4.10).

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

Protocol Π_{oc}'s piggybacking scheme guarantees that all the determinants needed to recover the system to a consistent global state from up to f concurrent failures will be available during recovery. However, the scheme leaves several open questions. First, Π_{oc} does not specify how to collect and use the logged determinants to perform recovery. Furthermore, Π_{oc} assumes that processes have knowledge of the current value of $|Log(m)|$ when they determine whether or not to piggyback the determinant $\#m$ on an application message. This knowledge is not realistic in an asynchronous distributed system, since — as we have observed before — it requires processes to have instantaneous access to *Log*(m), which is defined over the entire distributed system.

5.4 Existing Optimal Message-Logging Protocols

The piggybacking scheme used by Π_{oc} is the core of the logging components of the only two known optimal and causal message-logging protocols: family-based logging and Manetho [Eln93]. Family-based logging is described in detail in Chapter 6. We briefly describe the aspects of Manetho that concern the logging of determinants in the following section.

5.4.1 Manetho

Manetho is an optimal causal message-logging protocol designed to tolerate up to n concurrent failures. In Manetho, each process maintains an *antecedence graph* AG that records the causal relationship between non-deterministic events of an execution. AG contains a node for each determinant. Assuming that only deliver events are nondeterministic, an directed edge is drawn from $\#m'$ to $\#m$ when either m was sent in a state interval initiated by the delivery of message m' or when some process p delivered both m‘ and m and $m'.rsn = m.rsn - 1$.

The operating systems on which Manetho executes allows for the deterministic re-execution of non-deterministic internal events. Therefore, in addition to all of the message deliver events, the nondeterministic events recorded in the antecedence graph include other non-deterministic internal events, such as synchronization operation between threads. The nodes of the antecedence graph represent the state intervals started by each deliver event and contain the determinants of the corresponding deliver events. In particular, if process p is executing in state interval

$p[i]$, then the antecedence graph associated with $p[i]$ contains the determinant of the non-deterministic event e that started $p[i]$, plus the determinants of all the non-deterministic events that are in e's causal past. When process p sends a message m to process q, p piggybacks on m the current value of its antecedence graph. Optimizations are used to limit the amount of piggybacked data, though. When q receives m, it uses the antecedence graph piggybacked by p to update its own antecedence graph.

Manetho is unique among message-logging protocols because processes periodically take coordinated — as opposed to independent — checkpoints. This allows Manetho to simplify garbage collection of the determinants maintained in volatile memory when a checkpoint is taken.

A major difference between Manetho and the family-based logging protocols described in this dissertation is that Manetho implements Property (4.10) only for the special case $f = n$. Therefore, applications for which a smaller value of f would suffice must nonetheless pay the full cost of ensuring resilience from total failure. In contrast, as we will see, family-based logging provides a series of protocols, parameterized by f. The determinant of each message m need to be piggybacked only until $|Log(m)| > f$ holds rather than until $|Log(m)| = n$ holds. Consequently, fewer determinants need to be piggybacked and less volatile memory needs to be devoted to saving determinants.

Chapter 6

Family-Based Logging

This chapter presents family-based logging, a new message-logging protocol that is both causal and optimal. Section 6.1 provides an overview of the logging component of family-based logging, which is based on a piggybacking scheme similar to the one used by Π_{oc} of Chapter 5. Section 6.2 assesses the piggybacking overhead and presents several techniques that can be used to limit it. A central problem that family-based logging protocols must confront is how processes in an asynchronous distributed system can effectively compute $Log(m)$ and $|Log(m)|$ in order to satisfy Property 4.10. Section 6.3 presents several approaches to this problem. Section 6.4 defines five protocols that use these approaches in different ways, and Sections 6.5 and 6.6 compare their performance. Section 6.7 describes the recovery component of family-based logging. Section 6.8 discusses checkpointing, and Section 6.9 discusses communication with the environment.

6.1 The Logging Component

Family-based logging (FBL) protocols [AHM93, AM95, AM94b] disseminate and log determinants using a slightly modified version of the piggybacking scheme of protocol Π_{oc} in Section 5.3:

1. FBL does not assume that a process knows $|Log(m)|$ when determining whether to piggyback $\#m$ on an outgoing message m.

2. FBL requires each process to save in a message log, kept in volatile memory, the content of every message it has sent. Hence, in FBL each process p records the following data:

   - In the message log:
     - The content of every sent message.
   - In the determinant log:
     - The determinant of every delivered message.
     - A copy of any determinant $\#m$ piggybacked on any received message, for which, at the time of receipt, p estimates that $|Log(m)| \leq f$.

3. FBL requires the sender of a message m' to piggyback on m' a non-stable determinant $\#m$ only if the sender does not estimate that the destination of m' is a member of $Log(m)$. Furthermore, since channels are FIFO, a process p holding a non-stable determinant in its determinant log needs to piggyback it to the same process q no more than once.

Despite these differences, it is easy to see that, if FBL protocols never conclude that a determinant is stable unless $|Log(m)|$ has become larger than f, then the arguments used in Theorem 5.2 to prove that Π_{oc} is an optimal causal protocol apply also to FBL protocols. Therefore, the following theorem, that we present without further proof, holds:

Theorem 6.1 *FBL protocols satisfy Property 4.10. Furthermore, FBL protocols are optimal.*

Conceptually, in FBL each process p maintains for each message m estimated values for $Log(m)$ (which we denote as $Log(m)_p$) and $|Log(m)|$. Note that p's estimate of the size of $Log(m)$ need not be equal to the size of p's estimate of $Log(m)$. To distinguish between the two, we denote them as $|Log(m)|_p$ and $|Log(m)_p|$ respectively.

$Log(m)_p$ is updated so that if a process q is a member of $Log(m)_p$, then either

- q is a member of $Log(m)$, or
- q has failed, q was a member of $Log(m)$ before failing, and p has not yet detected q's failure.

Note that this update rule for $Log(m)_p$ guarantees that $|Log(m)|_p > f$ only if $\#m$ is stable, and therefore Property 4.10 is satisfied.

Furthermore, p maintains a set $\mathcal{DS}_p$ that contains the non-stable determinants for all the delivery events in the causal history of the last delivery event executed by p. $\mathcal{DS}_p$ is a subset of all of the determinants that p has in its determinant log. Whenever process p sends a message m to some process q, p piggybacks onto m all the determinants $\#m'$ in $\mathcal{DS}_p$ that p believes q may not have yet logged — i.e. the subset of $\mathcal{DS}_p$ that contains all the determinants $\#m'$ for which q is not a member of $Log(m')_p$ and such that $\#m'$ was not piggybacked on a previous message from p to q. We denote this subset with $\mathcal{DS}_p(q)$. Membership in $\mathcal{DS}_p$ is managed as follows: when p delivers a new message m, it adds to $\mathcal{DS}_p$ the determinant of m, plus all the non-stable determinants piggybacked onto m. As soon as p determines that a determinant in $\mathcal{DS}_p$ has become stable, it removes the determinant from $\mathcal{DS}_p$.

Notice that information needed by p to replay a message m during recovery is maintained partly in the message log of the sender of m, and partly in the determinant logs of processes that have delivered messages sent causally after the m's delivery. We call the sender of m the *parent* of p with respect to m; we call any process that delivers a message sent causally after m's delivery a *descendant* of p with respect to m. In particular, we call a process q a *child* of p if q delivers a message sent by p after m's delivery. Family-based logging owes its name to the fact that, if $f = 1$, then the information necessary to recover a process is distributed among its parents and its children. If $f > 1$, such information is distributed among the process' parents and descendants.

6.2 Assessing the Cost of Piggybacking

Family based logging protocols avoid creating orphan processes by piggybacking non-stable determinants on existing application messages. In this section we analyze the overhead in message size caused by such piggybacking. Note that FBL protocols may elect to piggyback other data in addition to the determinants, in order to achieve better estimates of $Log(m)$ and $|Log(m)|$. We discuss this possibility in Section 6.3.

Given a run ρ of a system of n processes and an FBL protocol that tolerates f process failures, we consider two metrics: the number of determinants that can be piggybacked on any single message and the number of times a single determinant can be piggybacked in ρ. We assume that during ρ no process fails and that processes rely solely on piggybacking to make determinants stable — i.e. processes do not use stable memory to log determinants. We define the following variables and functions:

- s is the number of send events in ρ.
- r is the number of receive events in ρ.
- d is the number of deliver events in ρ.
- k is the number of transient channel failures suffered in ρ.
- $pd_{max}(f, d)$ is the maximum number of determinants that are piggybacked on any of the messages sent in ρ.
- $\overline{pd}(f, n, k)$ is the average number of determinants piggybacked on all messages sent in ρ.
- $t_{max}(f, n, k)$ is the maximum number of times a determinant is piggybacked in ρ.

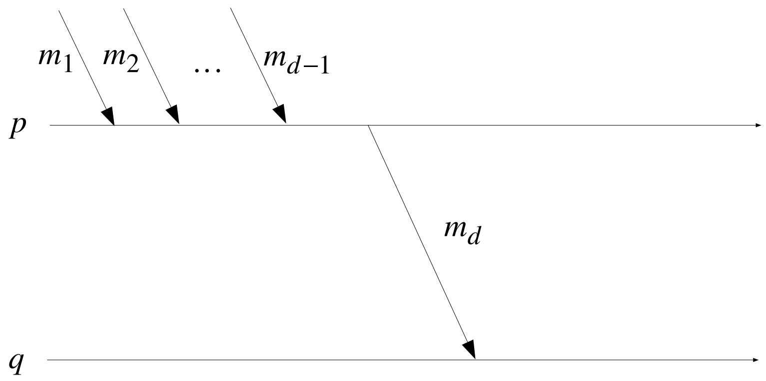

Figure 6.1: An execution for which $pd_{max}(f, d) = d - 1$.

Given these definitions, we make the following two observations:

1. Since ρ contains d delivery events, the total number of determinants generated during ρ is d.

2. The number k of transient channel failures suffered in ρ is at most equal to the difference between the number of send events and the number of receive events — i.e. $k \leq s - r$.

Theorem 6.2 *$pd_{max}(f, d)$ is $\Theta(d)$.*

Proof. Since a determinant needs to be piggybacked only once on a single message, it immediately follows that $pd_{max}(f, d)$ is bound by the total number of determinants d generated during ρ. To see that this bound is tight and does not depend on f, consider the run depicted in Figure 6.1. Process p delivers $d - 1$ messages from the environment before sending message m_d. Hence, p must piggyback $d - 1$, or $O(d)$, determinants on m. $\square$

Theorem 6.3 *$t_{max}(f, n, k)$ is $\Theta(n^{min(f,2)} + k)$.*

Proof. We first derive an upper bound for $t_{max}(f, n, k)$ that is independent of f, and we show that this bound is tight for $f > 1$. We then show that if $f = 1$, then we can derive a tighter upper bound. Finally, we combine these results to obtain the bound stated in the theorem.

Assume that during ρ no transient channel failures occur. Since by assumption channels are FIFO, any process $m.dest$ that delivers a message m must piggyback determinant $\#m$ at most once to every other process. Furthermore, no process needs to piggyback $\#m$ back to process $m.dest$. Hence, since there are n processes, $\#m$ can be piggybacked at most $n-1$ times by process $m.dest$, and at most $n-2$ times by any other process. Therefore it follows that:

$$t_{max}(f, n, k) \leq (n-1) + (n-1)(n-2) = (n-1)^2$$

Assuming k transient channel failures, the bound becomes:

$$t_{max}(f, n, k) \leq (n-1)^2 + k \tag{6.1}$$

We now show that the upper bound we have derived is tight for all $f > 1$. Consider the execution in Figure 6.2, in which $n = 4$. Assume that $f > 1$, and that no transient channel failures occur. In order to avoid creating orphan processes, once process p_2 delivers m it must piggyback $\#m$ onto each subsequent message it sends — though no more than once to any process, since channels are FIFO — until it can determine that $|Log(m)| > f$. Process p_2 can improve its estimates of $|Log(m)|$ whenever it receives an acknowledgment for a message onto which $\#m$ was piggybacked. For instance, in Figure 6.2 if p_2 receives an acknowledgment for the message sent to p_4, then p_2 can conclude that $|Log(m)| \geq 2$, since $Log(m)$

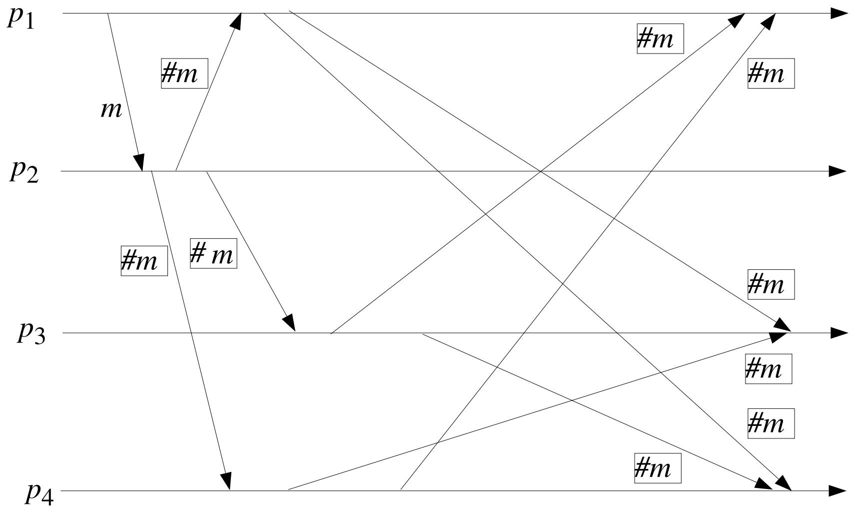

Figure 6.2: An execution in which $\#m$ is piggybacked $(n-1)^2$ times.

must contain at least p_4 and p_2 itself. Thus, the worst case scenario arises when acknowledgments are slow and p_2 must piggyback $\#m$ $n-1$ times before receiving any acknowledgment. Consider now any of the $n-1$ processes q that have received $\#m$ piggybacked on a message received from p_2. The best estimate of $|Log(m)|$ that can be computed by such a process is that $|Log(m)|_q \geq 2$, since $Log(m)_q$ certainly contains p_2. Therefore, if $f \geq 2$, each process other than p_2 must in turn piggyback $\#m$ onto each subsequent message it sends, until it can determine that $|Log(m)| > f$. The worst case scenario is again the one in which acknowledgments are slow, and each process piggybacks $\#m$ to the remaining $n-2$ processes before receiving any acknowledgments (see Figure 6.2). It immediately follows that in this scenario $\#m$ is piggybacked $(n-1)^2$ times.

If we assume that the run containing the execution shown in Figure 6.2 suffers k transient channel failures and that all such failures affect messages carrying $\#m$, then we have that in the worst case scenario $\#m$ can be piggybacked $(n-1)^2+k$ times, which matches the bound given in Equation 6.1.

If $f=1$, then we calculate the following tighter bound on $t_{max}(f,n,k)$:

$$t_{max}(f,n,k) \leq n-1+k \tag{6.2}$$

To see why, consider again the execution of Figure 6.2, and assume no transient channels failures occur. Whenever a process $p_i : i=1,3,4$ receives $\#m$ piggybacked on a message coming from p_2, p_i can conclude that $\#m$ has become stable, and that therefore $\#m$ does not need to be piggybacked any further. This is because $Log(m)$ must contain at least p_2 and p_i itself, and so $|Log(m)|>1$. Furthermore, as soon as p_2 receives an acknowledgment for any of the messages on which $\#m$ was piggybacked, it can also conclude that $\#m$ is stable, and stop piggybacking it. Therefore, the worst case scenario arises when p_2 piggybacks $\#m$ $n-1$ times, once to every process, before receiving any acknowledgments. Hence:

$$t_{max}(f,n,k) \leq n-1$$

If we assume that no more that k transient channel failures occur, then we obtain the bound given in Equation 6.2.

We have proved the following bounds:

1. If $f=1$, then $t_{max}(f,n,k) \leq n-1+k$.

2. If $f>1$, then $t_{max}(f,n,k) \leq (n-1)^2+k$.

We can combine them as follows:

$$t_{max}(f,n,k) \leq (n-1)^{min(f,2)}+k. \tag{6.3}$$

Since $(n-1)^{min(f,2)} + k$ is $\Theta(n^{min(f,2)} + k)$, the theorem is proved. $\Box$

Theorem 6.4 *$\overline{pd}(f,n,k)$ is $\Theta(n^{min(f,2)} + k)$.*

Proof. By definition, $\overline{pd}(f,n,k)$ equals the total number of piggybacked determinants divided by the total number of send events. By the proof of Theorem 6.3 we know that no determinants can be piggybacked more than $(n-1)^{min(f,2)} + k$ times. Since the number of determinants is d, the total number of piggybacked determinants is at most $d \cdot ((n-1)^{min(f,2)} + k)$. Dividing by the total number $r+k$ of send events, we obtain the following bound:

$$\overline{pd}(f,n,k) \leq d \cdot \frac{(n-1)^{min(f,2)} + k}{r+k}$$

Since $d \leq r$ and $k \geq 0$, this bound simplifies to:

$$\overline{pd}(f,n,k) \leq (n-1)^{min(f,2)} + k \tag{6.4}$$

Hence, $\overline{pd}$ is $O(n^{min(f,2)} + k)$. The bound given in Equation 6.4 is achieved if during ρ each process $p_i : 1 \leq i \leq n$ executes the following program:

do d/n **times**
  for all $p_j \in \mathcal{N}$ **st** $j \neq i$
    send message to p_j
  for $l := 1$ **to** $n-1$
    receive message

The second time the do-loop iterates, each process must piggyback $n-1$ determinants on every outgoing message. If $f = 1$, each process continues in subsequent iterations to piggyback $n-1$ determinants. If $f > 1$, then by the third time the do-loop iterates, each process must piggyback $(n-1)^2$ determinants on every outgoing message. Assuming no channel failures, the average number of determinants piggybacked will quickly approach $(n-1)^2$ as the loop repeats. $\Box$

6.2.1 Discussion

The theoretical bounds derived above suggest a few remarks.

First, the frequency of acknowledgments affect the performance of family-based logging. The number of piggybacked determinants carried by each message will decrease if acknowledgments arrive promptly. Thus, a positive acknowledgment protocol is ideal for family-based logging. Negative acknowledgment protocols may delay acknowledgments and increase piggyback size compared to positive acknowledgment schemes. However, the $n^{min(2,f)} + k$ bound on $\overline{pd}(f, n, k)$ applies regardless of using positive or negative acknowledgments

Second, the application's communication pattern strongly affects the performance of family-based logging. As the worst-case scenario shows, the size of the piggyback overhead becomes larger when each process exchanges a burst of messages with a large family of parents and children processes. Note that in this case sets of parents and children will intersect: each process will tend to send messages to and receive messages from some common set of processes. In such an environment, a negative acknowledgment protocol should be very effective, since extra acknowledgment packets will rarely be sent. Thus, in worst-case scenarios, family-based logging should actually perform better using a negative acknowledgment protocol.

Finally, Theorem 6.2 asserts that the number of determinants piggybacked to a specific message m can be bound only by the total number of delivery events that precede the sending of m (see Figure 6.1). Even if this worst case scenario is unlikely to occur in practice, there may be circumstances in which the number of non-stable determinants may become large enough that piggybacking them on a single message is costly. To avoid this problem, a process p can use one of the following two techniques whenever the number of non-stable determinants in $\mathcal{DS}_p$ exceeds a pre-determined threshold:

1. Flush $\mathcal{DS}_p$ to stable storage. For example, in Manetho a process asynchronously flushes $\mathcal{DS}_p$ to stable storage whenever the size of $\mathcal{DS}_p$ exceeds 64 bytes [Eln93]. Note that if this solution is used then the threshold must be chosen conservatively, in order to account for the possible increase of the size of $\mathcal{DS}_p$ while the asynchronous flushing completes.

2. Send extra messages carrying the non-stable determinants. Note that this solution contradicts one of the optimality criteria introduced in Section 5.1. Nevertheless, sending extra messages may be preferable in practice to piggybacking a large number of determinants on existing application messages.

6.2.2 Reducing the Size of the Piggyback

Consider a set of processes $p_i : 1 \leq i \leq n$. Whenever p_i sends a message m to p_j, it piggybacks on m all the determinants $\#m$ in $\mathcal{DS}_{p_i}$ for which $p_j \notin \mathit{Log}(m)_{p_i}$.

Hence, there are two approaches process p_i can use to reduce the amount of piggybacked data:

$$\begin{array}{lll} \{ & p_1 & \langle m_1.source, m_1.ssn, m_1.rsn \rangle \\ & & \langle m_2.source, m_2.ssn, m_2.rsn \rangle \\ & & \quad\vdots \qquad\quad \vdots \qquad\quad \vdots \\ & & \langle m_D.source, m_D.ssn, m_D.rsn \rangle \quad \} \end{array}$$

Figure 6.3: $\mathcal{DS}_{p_i}^{p_1}$ before compression.

1. Computing accurate estimates of $Log(m)$ and $|Log(m)|$ for all messages m, in order to avoid unnecessary piggybacking of stable determinants.

2. Using compression techniques to decrease the amount of data necessary to represent the information contained in $\mathcal{DS}_{p_i}$.

The second approach is the subject of this section. We discuss in detail the first approach in Section 6.3.

A preliminary step in reducing the space necessary to represent $\mathcal{DS}_{p_i}$ is to group determinants that share the same value for $m.dest$ and to factor the value of $m.dest$ out of each determinants. Thus, we partition $\mathcal{DS}_{p_i}$ into n subsets $\mathcal{DS}_{p_i}^{p_1} \ldots \mathcal{DS}_{p_i}^{p_n}$. Figure 6.3 shows a non-compressed representation of $\mathcal{DS}_{p_i}^{p_1}$.

In the following, we consider three techniques that can be used to compress $\mathcal{DS}_{p_i}^{p_1}$. Of course, the same techniques can be used to compress all the other subsets of $\mathcal{DS}_{p_i}$.

Exploiting FIFO Channels

Consider a determinant $\#m = \langle m.source, m.ssn, m.rsn \rangle$ in $\mathcal{DS}_{p_i}^{p_1}$. The fields $m.source$ and $m.ssn$ uniquely identify m in the message log of process $m.source$.

If process p_1 fails, then this information can be used during recovery to replay the content of m to p_1 according to the order established by $m.rsn$. If channels are FIFO, however, there is no need to explicitly keep $m.ssn$ in the determinant $\#m$. To see why, note that with FIFO channels $m.rsn$ and $m.source$ uniquely identify the relative send sequence number of m among the messages that process $m.source$ sends to p_1. Hence, given $m.dest$ and $m.rsn$, in order to identify m in $m.source$'s message log it is sufficient to structure the message log so that all messages sent to the same destination are maintained in sequences sorted by send sequence number.

Exploiting Causal Logging

Not only can we eliminate send sequence numbers from $\mathcal{DS}_{p_i}^{p_1}$, but we also can eliminate most of the receive sequence numbers. To do so, we first need to prove the following theorem and the subsequent corollary.

Theorem 6.5 *Let $\#m$ and $\#m'$ be such that:*

$$m'.dest = m.dest \wedge m'.rsn < m.rsn.$$

If $\#m'$ is not stable, then $Log(m) \subseteq Log(m')$

Proof. Assume $\#m'$ is not stable. If process p is a member of $Log(m)$, then either p is the destination of m, or p is the destination of a message m'' onto which $\#m$ was piggybacked. We consider these two cases separately.

Case 1 p is the destination of m: Since by assumption $m.dest = m'dest$, p is the destination of m'. Since $m'.rsn < m.rsn$, p logged $\#m'$ before logging $\#m$. Hence, $p \in Log(m) \Rightarrow p \in Log(m')$.

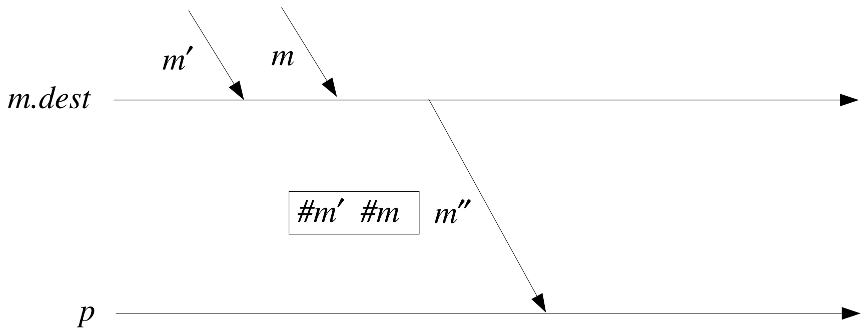

Figure 6.4: Proof of Theorem 6.5: base case.

Case 2 p is not the destination of m: We use an inductive argument. Let m''' be the first message sent by process $m.dest$ after adding $\#m$ to its determinant log, and let ℓ be the length of the causal chain of processes associated with the causal path that starts with $send_{m.dest}(m''')$ and ends with process p receiving m'' and thereby adding $\#m$ to its determinant log. We proceed by induction on ℓ.

Before proving the base case, though, we make the following observation: if we show that m'' piggybacks both $\#m$ and $\#m'$, then the theorem holds. This is because FBL guarantees that when determinants with the same value in their *dest* fields are piggybacked on the same message, these determinants are logged by the receiver in increasing receive sequence number order. So, since $m'.rsn < m.rsn$ and the *dest* fields in $\#m$ and $\#m'$ hold the same value, p must add $\#m'$ to its determinant log before $\#m$.

Base Case : $\ell = 1$.

Figure 6.4 illustrates the base case. If $\ell = 1$, then process $m.dest$ is the sender of m''. Since, by assumption, $\#m'$ is not stable when p adds $\#m$ to its determinant log, then $\#m'$ must be non-stable when $m.dest$ sends

m'', and since $m'.rsn < m.rsn$, process $m.dest$ must have saved $\#m'$ in its determinant log before saving $\#m$. Hence, when $m.dest$ sends m'' to p, both $\#m'$ and $\#m$ are members of $\mathcal{DS}_{m.dest}$. The FBL piggybacking scheme guarantees that $m.dest$ piggybacks on m'' all the determinants in $\mathcal{DS}_{m.dest}(p)$. Since $\#m$ is by assumption in $\mathcal{DS}_{m.dest}(p)$, if also $\#m'$ is in $\mathcal{DS}_{m.dest}(p)$ then m'' carries both $\#m$ and $\#m'$, and by the above observation the theorem is proved. If $\#m' \notin \mathcal{DS}_{m.dest}(p)$, then from the definition of $\mathcal{DS}_{m.dest}(p)$ and the assumption that $|Log(m')| \leq f$, it follows that $p \in Log(m')_{m.dest}$. Since we assume that p is in $Log(m)$, p has not failed. Hence, p in $Log(m')_{m.dest}$ implies that p must already be a member of $Log(m')$, and the theorem is proved.

Inductive Step: We assume that the theorem holds for all causal paths whose associated causal chain of process has length $\ell \leq k$. We prove that the theorem holds for any causal path whose associated causal chain of process has length $\ell = k + 1$.

Figure 6.5 illustrates the inductive step. Consider a causal path that starts with event $send_{m.dest}(m''')$ and ends when process p receives m'' and adds $\#m$ to its determinant log, and assume that the associated causal chain of processes has length $\ell = k + 1$. Let q be the process that sent message m'' to p. Since we assume that a process never sends a message to itself, it follows that $p \neq q$. Therefore, the causal chain associated with the causal path that starts with $send_{m.dest}(m''')$ and ends when process p receives m'' and adds

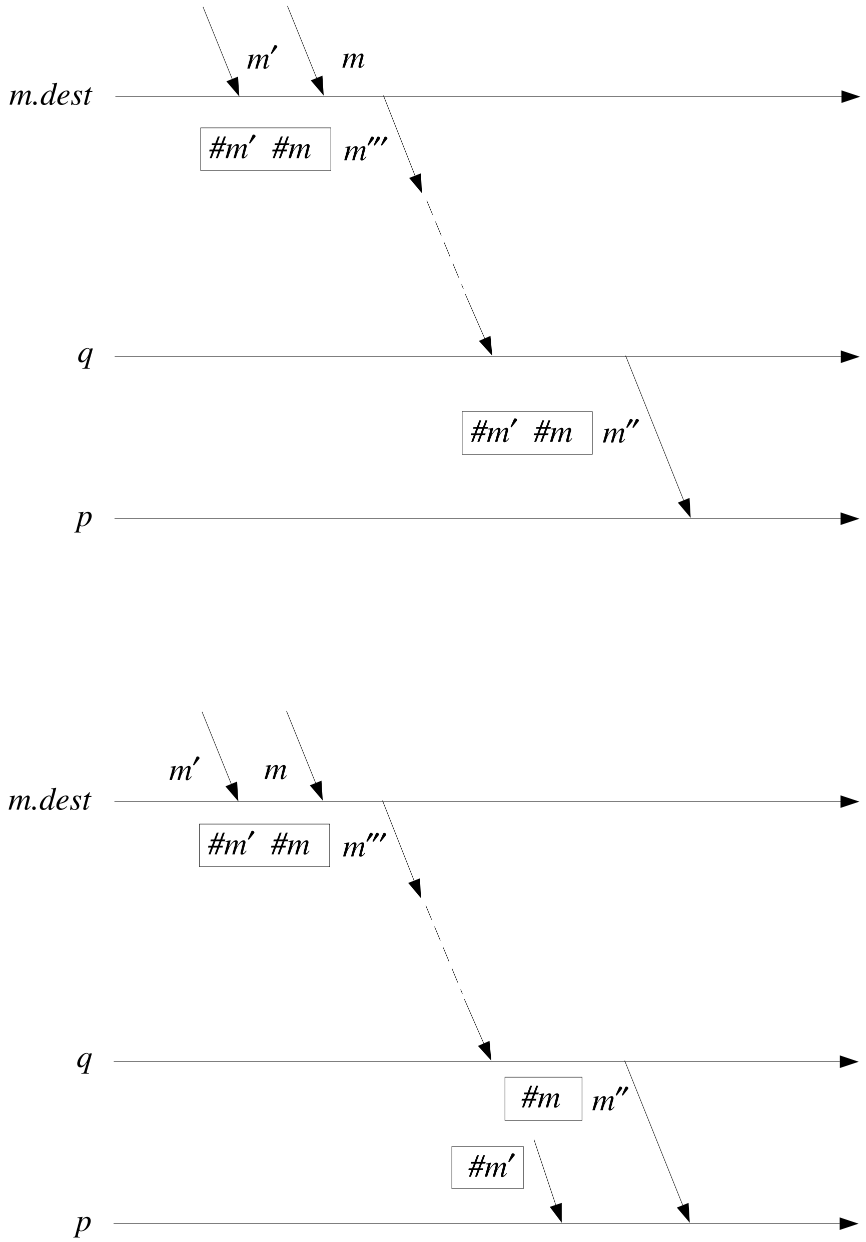

Figure 6.5: Proof of Theorem 6.5: the two cases in the inductive step.

$\#m$ to its determinant log must be of length k. Since, by assumption, $\#m'$ is not stable when p adds $Log(m)$ to its determinant log, then $\#m'$ is also not stable when q sends m'' to p. Hence, by the inductive step, when q sends m'' it must have both $\#m'$ and $\#m$ in its determinant log. Therefore, one of the following is true:

1. $\#m$ and $\#m'$ are both in $\mathcal{DS}_q(p)$.
2. $\#m$ is in $\mathcal{DS}_q(p)$, and $\#m'$ is in $\mathcal{DS}_q$, but not in $\mathcal{DS}_q(p)$.

We consider these two cases separately. The first case is illustrated in the top half of Figure 6.5. The FBL piggybacking scheme guarantees that q piggybacks on m'' all the determinants in $\mathcal{DS}_q(p)$. Since $\#m$ is by assumption in $\mathcal{DS}_q(p)$, if $\#m'$ is in $\mathcal{DS}_q(p)$ then m'' carries both $\#m$ and $\#m'$, and by the observation we made at the beginning of the proof the theorem holds. The second case is illustrated in the bottom half of Figure 6.5. If $\#m' \notin \mathcal{DS}_q(p)$, from the definition of $\mathcal{DS}_q(p)$ and the assumption that $|Log(m')| \leq f$ it follows that $p \in Log(m')_q$. Since we assume that p is in $Log(m)$, p has not failed. Hence, p in $Log(m')_q$ implies that p must already be a member of $Log(m')$, and the theorem is proved. $\square$

Corollary 6.1 *Let $\#m$ and $\#m'$ be such that:*

$$m'.dest = m.dest \wedge m'.rsn < m.rsn.$$

If $\#m'$ is not stable, then $\#m$ is not stable.

$$\{ \quad p_1 \quad m_l.rsn \quad \{ \quad m_l.source \\ \vdots \\ m_h.source \quad \} \quad \}$$

Figure 6.6: $\mathcal{DS}_{p_i}^{p_1}$ compressed using FIFOness and causal logging.

Proof. Assume that $\#m'$ is not stable. By Theorem 6.5, we know that $Log(m) \subseteq Log(m')$ at all times. Hence, from the definition of being stable and from the fact that $\#m'$ is not stable, $\#m$ is not stable. $\Box$

We now can compress the representation of $\mathcal{DS}_{p_i}^{p_1}$ as follows. Suppose that we sort the determinants in $\mathcal{DS}_{p_i}^{p_1}$ in ascending receive sequence number order. Let rsn_l and rsn_h be the lowest and highest receive sequence numbers in $\mathcal{DS}_{p_i}^{p_1}$. We denote the corresponding determinants with $\#m_l$ and $\#m_h$. Theorem 6.5 guarantees that all the non-stable determinants with receive sequence number lower than rsn_h are in $\mathcal{DS}_{p_i}^{p_1}$. Corollary 6.1 guarantees that all the determinants with a receive sequence number higher than rsn_l are non-stable. Taken together, these results guarantee that all the receive sequence number between rsn_l and rsn_h are represented in $\mathcal{DS}_{p_i}^{p_1}$.

Then, it is sufficient to keep in $\mathcal{DS}_{p_i}^{p_1}$ only the lowest receive sequence number value, followed by the list of the processes that sent the determinants in $\mathcal{DS}_{p_i}^{p_1}$ to be able to infer the receive sequence number of all the determinants in $\mathcal{DS}_{p_i}^{p_1}$.

Using Run Length Encoding

Using the two previous techniques, we have transformed $\mathcal{DS}_{p_i}^{p_1}$ from the representation of Figure 6.3 to the representation of Figure 6.6. We now consider one method of further compressing the piggybacked sequence of parents.

Suppose it is known that some process p has only a single parent q. Thus, the send sequence numbers assigned by q define a total ordering of the messages delivered by p, and p need not save any determinant for all its delivery events. Should p fail, q's message log contains sufficient information to recover p. This optimization can be generalized to the case where p has multiple parents. The result is similar to the *null-logging* technique described in [SBY88].

Process p logs receive sequence numbers in order to record the nondeterministic choices made during a run. Since channels are FIFO, however, p need only log the order in which p interleaves messages from different parents. If a message-logging protocol records the interleaving of messages from different parents for a process p, then this information together with the message logs of p's parents is sufficient to recover p from failure.

Implementing such a scheme requires each process p to maintain an *interleave sequence number* isn_p. isn_p is initially zero, and is incremented in state $p[\ell]$ if $\ell = 1$ or if the source of the ℓ^{th} message is different from the source of the $(\ell - 1)^{th}$ message. Furthermore, the notion of determinant and the rules followed by the logging component must be modified. A determinant $\#m$ must now contain two new fields:

- $m.isn$, which equals the value of $isn_{m.dest}$ in state $m.dest[m.rsn]$.

- $m.runlength$, which equals the number of consecutive messages delivered by $m.dest$ from $m.source$ since state $m.dest[m.rsn]$.

The logging component is then modified so that a process p generates and logs a new determinant only when p increments its interleave sequence number. If p delivers a message m' but does not increment its interleave sequence number, then m' must have been sent by the same process that sent the last message delivered by p. Hence, p must increment the *runlength* field in the latest determinant it generated. Note that in this scheme several delivery events may be encoded in a single determinant: together with $m.rsn$, $m.runlength$ determines the receive sequence numbers of all the delivery events encoded in $\#m$.

The details of how this optimization can be implemented in an FBL protocol for $f = 1$ are presented in [AHM93].

A similar compression of the representation of $\mathcal{DS}_{p_i}^{p_1}$ can be achieved by applying directly run length encoding to the sequence of parents $m_l.source, \ldots m_h.source$ in Figure 6.6. Note however that while this simpler solution successfully decreases the size of the piggybacked information, it still generates a distinct determinant every time the receive sequence number is incremented, as opposed to every time the interleave sequence number is incremented. By using a version of FBL that explicitly records interleave sequence numbers, applications for which run length encoding significantly reduces the size of the piggyback can also achieve a parallel significant reduction in the size of their determinant logs.

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

A fundamental issue in implementing FBL is how a process p determines its estimates of $Log(m)$ and $|Log(m)|$ for any message m whose determinant that p has received. In order for the protocol to satisfy Property (4.10), p must never conclude that a determinant is stable unless $|Log(m)| > f$ holds. As noted earlier, this is guaranteed if, when p estimates that a process q is a member of $Log(m)$, then either q is currently a member of $Log(m)$ or q has failed and was a member of $Log(m)$ before failing. In either case we say that p *can safely infer* that q is a member of $Log(m)$. However, if p underestimates $|Log(m)|$, it may needlessly piggyback determinants that are already stable, making messages significantly larger on average.

Process p computes its estimates on the basis of acknowledgments it receives, and on the basis of information piggybacked on the messages it receives. Hence, by piggybacking more information on messages they exchange, processes can improve the accuracy of their estimates and avoid piggybacking useless data; however, information piggybacked to maintain more accurate estimates can make the messages significantly larger. There is a tradeoff that must be taken into account when designing FBL protocols.

The most basic piece of information about $Log(m)$ and $|Log(m)|$ is gained when a process q delivers a message m: once q delivers m, q knows that $q \in Log(m)$. Also, when q receives an acknowledgment from p for a message onto which q had piggybacked $\#m$, q can safely infer that p is a member of $Log(m)$. Further pieces of information can be piggybacked on messages. Three natural pieces of information are:

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

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

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

6.4 Five FBL Protocols

In this section we define five FBL protocols that use in different ways the pieces of information we identified in Section 6.3 to compute their estimates of $|Log(m)|$ and $Log(m)$. We use these protocols throughout the chapter to illustrate trade-offs that arise in designing FBL protocols.

Recall, $\mathcal{DS}_p(q)$ denotes the subset of $\mathcal{DS}_p$ that contains the determinants $\#m$ such that $q \notin Log(m)$. Based on the information that process p piggybacks on its messages to process q, we define the following five FBL protocols:

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

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

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

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

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

6.5 Accuracy of the Five Protocols in Estimating $|Log(m)|$ and $Log(m)$

The execution shown in Figure 6.7 illustrates the differences between Π_{Det}, $\Pi_{|Log|}$ and Π_{Log} with respect to how accurately they estimate $Log(m)$ and $|Log(m)|$. For

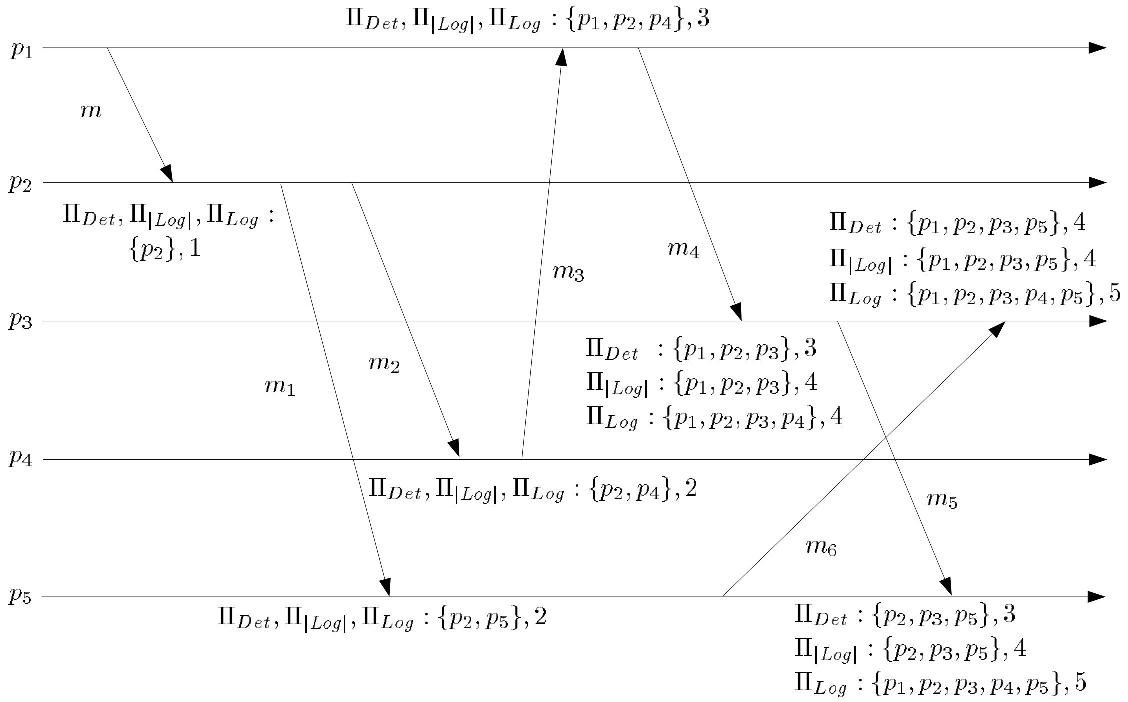

Figure 6.7: $Log(m)_{p_i}$ and $|Log(m)|_{p_i}$ for Π_{Det}, $\Pi_{|Log|}$ and Π_{Log}.

each deliver event executed by process p_i and for each of the three protocols, we show $Log(m)_{p_i}$ and $|Log(m)|_{p_i}$.

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

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

$\Pi_{|Log|}$: As in the previous case, process p_3 sets $Log(m)_{p_3}$ to $\{p_1, p_2, p_3\}$. However,

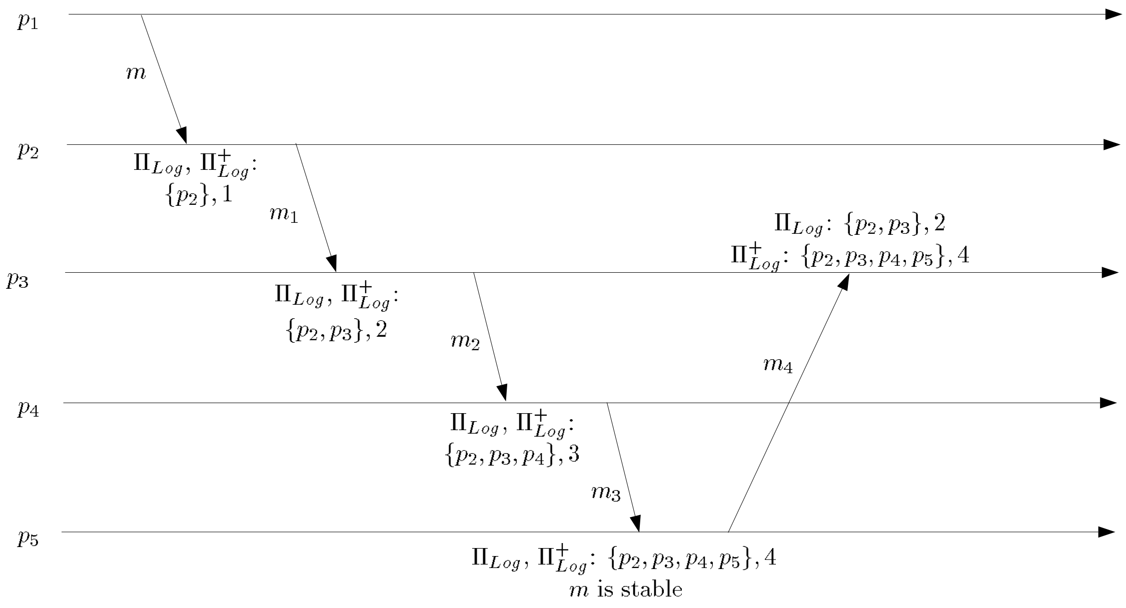

Figure 6.8: Comparison of Π_{Log} and Π^+_{Log} for $f = 3$.

since this is the first time that p_3 receives $\#m$, p_3 was not in $Log(m)$ when p_1 sent m_4. Since $|Log(m)|_{p_1} = 3$, p_3 can infer that $|Log(m)|$ must be at least 4.

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

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

Protocols $\Pi^+_{|Log|}$ and Π^+_{Log} are similar to $\Pi_{|Log|}$ and Π_{Log}, but can provide better estimates of $Log(m)$ and $|Log(m)|$. An example illustrating the difference between Π_{Log} and Π^{+}_{Log} is given in Figure 6.8. 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 subsequent message sent by p_3 will contain a piggybacked value of $\#m$, while using Protocol Π^{+}_{Log} it will not. A similar scenario can be constructed with Protocols $\Pi_{|Log|}$ and $\Pi^{+}_{|Log|}$. Note that $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} can provide better estimates of $\Pi_{|Log|}$ and Π_{Log} even when $f = n$. This is somewhat surprising, since when $f = n$ the $\mathcal{DS}$ set of a process contains all the determinants in that process determinant log, and $\Pi_{|Log|}$ and Π_{Log} would appear to become identical to $\Pi^{+}_{|Log|}$ and Π^{+}_{Log}, respectively. To see why the differ, consider $\Pi_{Log}$and Π^{+}_{Log}. A process p using Π_{Log} piggybacks $Log(m)_p$ on a message to a process q only if $\#m$ is also piggybacked on the same message. Since channels are FIFO, p does not piggyback $\#m$ to q more than once, and therefore does not piggyback $Log(m)_p$ to q more than once. If p instead uses Π^{+}_{Log}, then, once $\#m$ is in p's determinant log, $Log(m)_p$ is piggybacked on every message p sends to q, and q can use $Log(m)_p$ repeatedly to update its own estimate of $Log(m)$.

Consider again the execution shown in Figure 6.7. As long as a process estimates that $|Log(m)|$ is less than three, the three protocols produce identical estimates. This is true in general: the estimates given by any FBL protocol will be identical as long as for all messages m, a sender's estimate of $|Log(m)|$ is less than three. The reason why this holds is that whenever a process q receives $\#m$ from process p, it can always conclude that $\{m.dest, p, q\} \subseteq Log(m)$. Hence, whenever

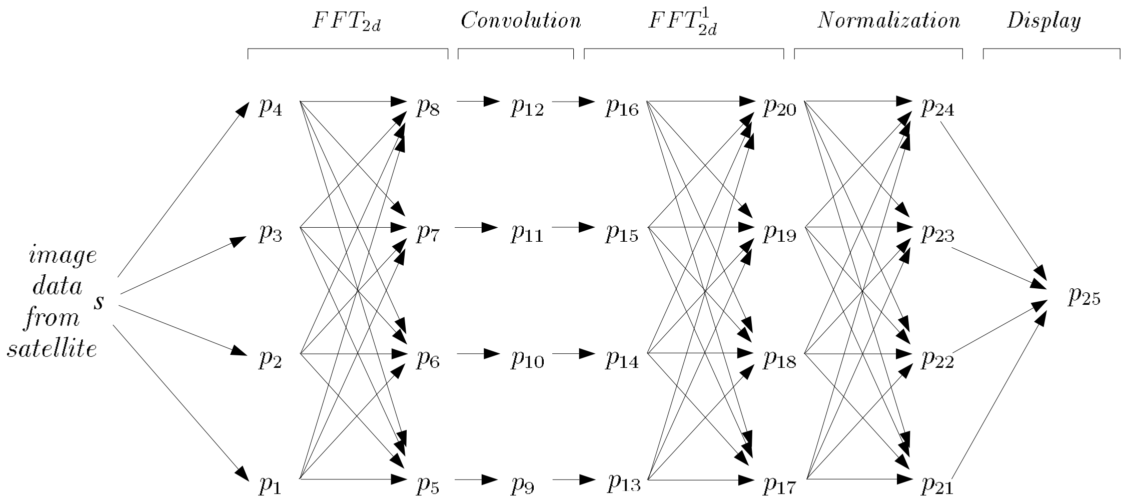

Figure 6.9: A parallel solution to the Synthetic Aperture Radar problem.

$f < 3$ the most efficient FBL protocol is the one that piggybacks on each message the minimum amount of data needed to satisfy Property 4.10. We can therefore formulate the following general guideline:

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

There are applications, however, for which Π_{Det} performs as well as Π_{Log} even for large values of f. For example, Figure 6.9 shows an application for which Π_{Det} does as well as Π^{+}_{Log} when $f = n$ [BA$^+$92]. The application is a parallel solution to the Synthetic Aperture Radar problem (SAR) in which radar echoes, collected by aircraft or spacecraft, are used to construct terrain contours. The steps necessary for producing high-quality images from SAR data consist of the following sequence of computations: two-dimensional discrete Fourier transform, binary convolution, two-dimensional inverse discrete Fourier transform, and intensity level normalization for visualization. For our purposes, however, the important property to note is that data flows in a particular manner.

To characterize a set of applications for which Π_{Det} performs as well as Π^{+}_{Log}, we represent an application's pattern of communication with a *communication graph.* For a given application, its associated communication graph is a directed graph that contains nodes, solid edges, and dashed edges. 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; solid edges are used to represent the direction of application messages; dashed edges are used to represent the direction of acknowledgments. Hence, communication graphs are built as follows:

- The communication graph contains a node for each application process, and each source and destination of application messages respectively received from, or sent to, the environment.

- The communication graph contains a solid edge from node i to node j, and a dashed edge from node j to node i, if one of the following holds:

  1. the process corresponding to node i is a parent of the process corresponding to node j.

  2. the process corresponding to node j delivers messages received from the source in the environment corresponding to node i

  3. the process corresponding to node i sends messages to the destination in the environment corresponding to node j

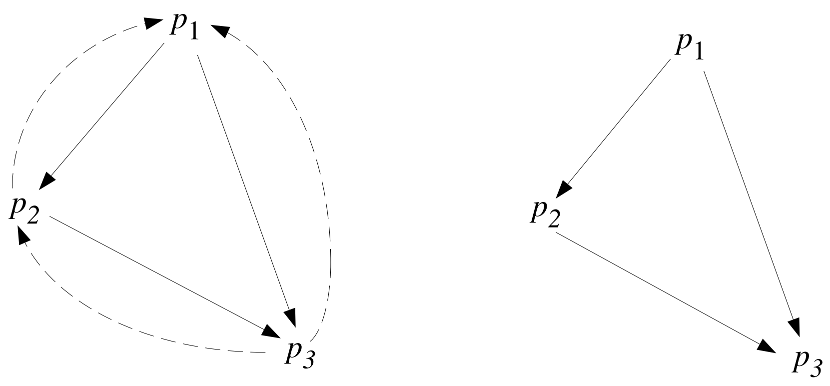

Figure 6.10: A simple communication graph and its associated channel graph.

For simplicity, we will denote with p_i the node in the communication graph corresponding to process p_i.

Given a communication graph $\mathcal{C}$, we define its associated *channel graph* to be the graph obtained from $\mathcal{C}$ by removing all the dashed edges.

Figure 6.10 shows the communication and channel graphs for an application in which process p_1 sends messages to p_2 and p_3, and process p_2 sends messages to p_3.

The communication and channel graphs of an application provide some indication of how a process builds its estimates of $Log(m)$ and $|Log(m)|$ for a determinant $\#m$. Recall from Section 6.3 that a process p_i may be able to add a process p_j such that $i \neq j$ to $Log(m)_{p_i}$ if either:

- p_i receives an acknowledgment from p_j of a message onto which p_i had piggybacked $\#m$, or

- p_i receives an application message from a process p_k for which $p_j \in Log(m)_{p_k}$.

We can express these conditions in terms of the communication graph by saying that in order for p_i to be able to add p_j to $Log(m)_{p_i}$, there must exist a path in the communication graph from node p_i to node p_j. Furthermore, all the edges along the path, with the possible exception of the first edge, must be solid edges, since neither $\#m$ nor estimates of $Log(m)$ and $|Log(m)|$ are piggybacked on acknowledgments.

Definition 1 *A channel graph is* tree-like *if there exists a function c that assigns to each node in the graph a (not necessarily unique) integer such that if there is an edge from node i to node j, then the integer assigned to i is one less than the integer assigned to j:* $c(i) = c(j) - 1$

It follows immediately from the definition that tree-like channel graphs are acyclic. However, not all acyclic graph are tree-like. An example of an acyclic channel graph that is not tree-like is given in Figure 6.10. Note that an application in which processes communicate in a tree results in a tree-like channel graph, and that the channel graph in Figure 6.9 is not a tree, but is tree-like.

Theorem 6.6 *Let* $f = n$. *Given a communication graph such that its associated channel graph is tree-like, for any run* ρ, *Protocol* Π_{Det} *piggybacks on each message the same determinants as Protocol* Π^{+}_{Log}.

Proof. A determinant $\#m$ is piggybacked on a message m' from p_i to p_j if the following four conditions hold:

(i) $p_i \in Log(m)$

(ii) $|Log(m)|_{p_i} \leq f$

(iii) $p_i \in \mathit{Depend}(m)$

(iv) $p_j \notin \mathit{Log}(m)_{p_i}$

To prove the theorem, we show that, for any ρ, if these conditions are satisfied for a tree-like set of processes by Π_{Det}, then they are also satisfied by Π^{+}_{Log}, and vice versa.

Satisfying (ii) and (iii) does not depend on the kind of FBL protocol used during ρ. Since $f = n$, (ii) trivially holds.[1] Furthermore, since FBL protocols don't send any message in addition to those required by the application, they don't affect the membership in $\mathit{Depend}(m)$. Hence, (ii) and (iii) hold using Π_{Det} if and only if they hold using Π^{+}_{Log}.

Π_{Det} and Π^{+}_{Log} are FBL protocols, and so satisfy Property 4.10. From Property 4.10 and the fact that (ii) and (iii) hold using Π_{Det} if and only if they hold using Π^{+}_{Log}, it follows that (i) holds using Π_{Det} if and only if it holds using Π^{+}_{Log}

The last obligation left to prove the theorem is to show that (iv) holds using Π_{Det} if and only if it holds using Π^{+}_{Log}. In order to do so, we first show that, if (iv) holds using Π^{+}_{Log}, then it holds using Π_{Det}. We then show if (iv) holds using Π_{Det}, then it holds using Π^{+}_{Log}.

Assume that, using Π^{+}_{Log}, $p_j \notin \mathit{Log}(m)_{p_i}$ holds. Since Π^{+}_{Log} carries more information than Π_{Det}, any estimate of $\mathit{Log}(m)$ that p_i computes using Π_{Det} is at least as accurate as the one obtained using Π^{+}_{Log}. Thus, we conclude that if $p_j \notin \mathit{Log}(m)_{p_i}$ using Π^{+}_{Log}, then $p_j \notin \mathit{Log}(m)_{p_i}$ using Π_{Det} .

[1]Without loss of generality, we assume that processes do not checkpoint nor save their volatile determinant log to stable storage during ρ.

Assume now that, using Π_{Det}, $p_j \notin \mathit{Log}(m)_{p_i}$. In order for Π^{+}_{Log} to estimate that $p_j \in \mathit{Log}(m)_{p_i}$, there must exist a path in the communication graph from node p_j to node p_i. This path cannot contain only solid edges, or the associated channel graph would contain a cycle and therefore would not be tree-like. Hence, the first edge of the path must be a dashed edge, corresponding to an acknowledgment from process p_j to a third process p_k. Note that p_k must be different from p_i, since Π_{Det} and Π^{+}_{Log} do not differ in how they use acknowledgments to estimate $\mathit{Log}(m)$, and we know that by using Π_{Det} $p_j \notin \mathit{Log}(m)_{p_i}$. Furthermore, since there is a dashed edge from p_j to p_k, there must be a solid edge from p_k to p_j. We therefore conclude that in order for p_j to be a member of $\mathit{Log}(m)_{p_i}$ using Π^{+}_{Log}, but not using Π_{Det}, the tree like channel graph must contain:

- An edge from p_k to p_j
- A path from p_k to p_i
- An edge from p_i to p_j

Figure 6.11 shows such a channel graph. We now show by contradiction that no such channel graph can be tree-like.

Assume the graph to be tree-like. Then, since there is an edge from p_k to p_j, and from p_i to p_j, we can conclude that $c(p_k) = c(p_i) = c(p_j) - 1$. However, since there is a path from p_k to p_i, we can also conclude that $c(p_k) \leq c(p_i) - 1$, and therefore that $c(p_k) < c(p_i)$. Hence, we have obtained that $c(p_k)$ must be at the same time equal to and less than $c(p_i)$, which is a contradiction.

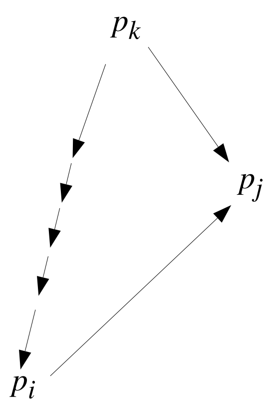

Figure 6.11: The channel graph obtained assuming that Π^+_{Log} estimates $Log(m)_{p_i}$ better than Π_{Det}.

We can therefore conclude that for all tree-like channel graphs, if Π_{Det} estimates that $p_j \notin Log(m)_{p_i}$, then so does Π^+_{Log}.

Since Conditions (i) to (iv) hold identically using Π_{Det} or Π^+_{Log}, the two protocols piggyback the same determinants on each message, and the theorem is proved.□

There exist communication graphs for which Π_{Det} sends the same determinants as Π^+_{Log} even when $f < n$:

Theorem 6.7 *Let $f \leq n$. Given a communication graph such that its associated channel graph forms a tree (as opposed to a tree-like channel graph), for any run ρ Protocol Π_{Det} piggybacks on each message the same determinants as Protocol Π^+_{Log}.*

Proof. As we have noted in Section 6.2.2, a process needs only log the order in which it interleaves messages from different parents. If the application's communi-

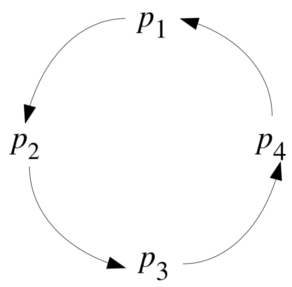

Figure 6.12: A non tree-like communication graph where Π_{Det} performs as well as Π^+_{Log}.

cation graph is a tree, then each process has a single parent. Therefore, if channels are FIFO, no determinant needs to be logged, independent of the FBL protocol used. □

These two characterizations are not the weakest possible: there are non tree-like communication graph in which Π_{Det} piggybacks determinants no more often that Π^+_{Log} when $f < n$. Figure 6.12 shows an example of such a graph, in which again, since processes do not interleave messages from different sources, any FBL protocol only needs to log the determinant of the first deliver event.

6.6 Piggyback Overhead for the Five Protocols

In Section 6.5 we have shown that if $f < 3$, then the additional data piggybacked by $\Pi_{|Log|}$ $\Pi^+_{|Log|}$, Π_{Log} and Π^+_{Log} add no value over the information carried by Π_{Det}.

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

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

As we saw in Section 6.2.1, how this trade-off works in practice for a particular application is largely a function of the application's pattern of communication and of the network's responsiveness in delivering acknowledgments. In order to understand the parameters of this trade-off, however, it is instructive to compare the amount of data that each protocol piggybacks on a message carrying a fixed number of determinants. For simplicity, in our calculations we don't consider optimizations achievable by applying the compression techniques described in Section 6.2.2.

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

1. Π_{Det}: Dw words.

2. $\Pi_{|Log|}$: $D(w+1)$ words, or D words more than Π_{Det}.

3. $\Pi^{+}_{|Log|}$: $Dw+N$ words, or N words more than Π_{Det}.

4. Π_{Log}: Up to $D(w + f)$ words, or Df words more than Π_{Det}.

5. Π^{+}_{Log}: Up to $Dw + Nf$ words, or Nf words more than Π_{Det}.

From our discussion in Section 6.2, we know that in the worst case D and N can only be bound by the total number of delivery events d that causally precede the sending of m. Thus, the extra information sent by $\Pi_{|Log|}$, $\Pi^{+}_{|Log|}$, Π_{Log} and Π^{+}_{Log} does not worsen the theoretical asymptotically worst case behavior of FBL protocols. In practice, however, when D is large, adding an extra piggyback proportional to D, as $\Pi_{|Log|}$ and Π_{Log} do, can result in significant extra overhead. Furthermore, even when D is small, N is most likely large, making $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} appear even less practical.

Hence, it could be advantageous to represent the extra information carried by $\Pi_{|Log|}$, $\Pi^{+}_{|Log|}$, Π_{Log} and Π^{+}_{Log} using a data structure whose size is independent of D or N.

Protocol $\Pi_{|Log|}$ can be easily modified to achieve this goal by sorting the determinants $\#m'$ piggybacked on m according to $|Log(m')|$. The resulting version of $\Pi_{|Log|}$ piggybacks no more than f additional words than Π_{Det}, an amount which is independent of D. A drawback of this approach, however, is that determinants sorted in this manner are not suitable for some of the compression techniques described in Section 6.2.2, which can dramatically reduce the size of the piggyback. Furthermore, this approach can not be applied to $\Pi^{+}_{|Log|}$, Π_{Log} or Π^{+}_{Log}.

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

6.6.1 The Dependency Matrix

In order to represent the information carried by $\Pi_{|Log|}$, $\Pi^{+}_{|Log|}$, Π_{Log} and Π^{+}_{Log} efficiently, we exploit the relationship that exists in Property (4.10) between $Log(m)$ and $Depend(m)$. Observe that the following definition of $Log(m)$ satisfies Property 4.10:

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

A process can use $|Depend(m)|$ to evaluate $|Log(m)|$, and take advantage of *vector clocks* [Mat89] to detect causal dependencies.

A vector clock is a generalization of Lamport's logical clocks [Lam78]. Given a set $\mathcal{N}$ of n processes, a vector clock is a vector of size n of natural numbers. Each process p in the set maintains its local vector clock and assigns a (vector) time-stamp to each event it executes. Let $VC_p(e_p)$ denote the time-stamp associated by process p to event e_p. The update rules for vector clocks support the following interpretation for the entries of $VC_p(e_p)$:

- $VC_p(e_p)[p]$ is the number of events process p executed up to and including e_p;
- $VC_p(e_p)[q], q \neq p$ is the number of events of q that causally precede event e_p.

The ordering relation $VC(e) \leq VC(e')$ is defined to be

$$\forall i \in \mathcal{N} : VC(e)[i] \leq VC(e')[i]$$

Furthermore,

$$VC(e) < VC(e') \text{ iff } (VC(e) \leq VC(e')) \wedge (\exists i : VC(e)[i] < VC(e')[i])$$

The vector clock update rules ensure that the following condition holds:

$$e \rightarrow e' \equiv VC(e) < VC(e'). \tag{6.5}$$

Note that for the above test it is not necessary to know on which processes the two events occurred. If this information is available, then causal precedence between two events can be verified through a single scalar comparison. In particular, given event e_p of process p and event e_q of process $q \neq p$, the following condition holds:

$$e_p \rightarrow e_q \equiv VC(e_p)[p] < VC(e_q)[p]. \tag{6.6}$$

Strom and Yemini [SY85] were the first to use vector clocks in conjunction with message logging when they introduced the notion of *dependency vector*. A dependency vector is a vector clock that is specialized to determine causal dependencies between delivery events occurring at different processes. Since in the piecewise deterministic model there is a one-to-one correspondence between delivery events and state intervals, dependency vectors can be used to determine dependencies among state intervals of different processes.

Let $DV_p(deliver_p(m))$ denote the value of the dependency vector of process p corresponding to the delivery of message m by p:

> $DV_p(deliver_p(m))[p]$ is the index of the state interval initiated in p by event $deliver_p(m)$, which is identical to the receive sequence number of message m.

$DV_p(deliver_p(m))[q]$ is the highest index of any state interval of process q that process p depends upon, which is identical to the highest receive sequence number of any message delivered by process q that process p depends upon.

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

$$deliver_p(m) \rightarrow deliver_q(m') \equiv DV_p(deliver_p(m))[p] \leq DV_q(deliver_q(m'))[p] \quad (6.7)$$

Dependency vectors are designed to track arbitrary dependencies between delivery events. In the context of FBL, we are interested in determining which processes depend on event $deliver_p(m)$ only when $|Depend(m)| \leq f$. We therefore introduce *weak dependency vectors* that satisfy the following weaker version of (6.7):

$$\begin{aligned} & deliver_p(m) \rightarrow deliver_q(m') \wedge |Depend(m)| \leq f \Rightarrow \\ & \quad WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p] \end{aligned} \quad (6.8.a)$$

$$\begin{aligned} & (WDV_p(deliver_p(m))[p] \leq WDV_q(deliver_q(m'))[p]) \Rightarrow \\ & \quad deliver_p(m) \rightarrow deliver_q(m') \end{aligned} \quad (6.8.b)$$

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

Notice that from (6.8.a), (6.8.b), and from the definition of $Depend(m)$ it follows that, for any given message m for which $|Depend(m)| \leq f$, the membership of a process p in $Depend(m)$ can be determined by reading process p's current weak dependency vector. In particular, the following conditions hold:

$$p \in \mathit{Depend}(m) \wedge |\mathit{Depend}(m)| \leq f \Rightarrow \mathit{WDV}_p[m.\mathit{dest}] \geq m.rsn \qquad (6.9.a)$$

$$\mathit{WDV}_p[m.\mathit{dest}] \geq m.rsn \Rightarrow p \in \mathit{Depend}(m). \qquad (6.9.b)$$

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

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

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

Note that the estimate of the weak dependency vector of a generic process q maintained by process p in $DM_p[q, *]$ will not in general be able to satisfy Condition (6.9.a), since the distributed and asynchronous nature of our system will not in general allow process p's estimate to be accurate.

However, it is straightforward to design update rules ensuring that Condition (6.9.b) holds. This provides process p with a simple method to estimate $|\mathit{Depend}(m)|$ and, therefore, $|\mathit{Log}(m)|$, for a particular message m: process p can just check how many entries of $DM_p[*, m.\text{dest}]$, the column corresponding to process $m.\mathit{dest}$, are greater than or equal to $m.rsn$. In particular, process p will consider $\#m$ to be stable if more than f entries of $DM_p[*, m.\text{dest}]$ are greater than or equal to $m.rsn$.

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

Since the dependency matrix of process p can be used to compute $Log(m)_p$ for all the messages for which p is a member of $Depend(m)$, when p sends a message m' to q it can simply piggyback DM_p on m'. Process q can then use the piggybacked dependency matrix, the piggybacked determinants, and its own dependency matrix to compute new values of $Log(m)_q$ and $|Log(m)|_q$ for all messages m whose determinants are logged in q's determinant log. Using this technique, we can implement Protocols Π_{Log} and Π^+_{Log} by piggybacking $n_P{}^2$ additional data over Π_{Det}, independent of the number of determinants D in $\mathcal{DS}_p(q)$ and N in p's determinant log.

Similarly, we can implement a version of $\Pi_{|Log|}$ and a protocol $\Pi^+_{|Log|}$ that is analogous to $\Pi_{|Log|}$ and that piggybacks $n_P(f+1)$ additional data can be derived by extracting the following data structure from the dependency matrix.

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

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

Figure 6.13 shows the dependency matrices of the processes in Figure 6.7, and the stability matrix of processes p_5 and p_3 after having delivered, respectively, messages m_5 and m_3.

6.7 The Recovery Component

Recovery in FBL is accomplished by replaying to the recovering process the messages delivered before failing, in the order established by the corresponding determinants. When a process p fails, a new instance of process p is generated. In order to distinguish between the instance of p that failed and the new instance of p, we denote the first one with p_{old}, and the second one with p_{new}.

Upon restarting after a failure, p_{new} executes a recovery procedure consisting of two phases. During the first phase, p_{new} determines the latest state interval to which it can recover by gathering determinants of delivery events that p_{old} executed before failing. During the second phase, p re-executes the original run that produced the state interval determined in Phase 1. Note that, since the system does not guarantee an upper bound on message delivery time, it is possible that a message m sent by p_{old} is received by a process q after q has detected p_{old}'s failure. We say that in this case m has become *stale*. For most applications, the

$$\begin{bmatrix} 0&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix}$$

DM_{p_2} after the delivery of m

$$\begin{bmatrix} 0&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&1&0&0&1 \end{bmatrix}$$

DM_{p_5} after the delivery of m_1

$$\begin{bmatrix} 0&0&0&0&0\\ 0&1&0&0&0\\ 0&0&0&0&0\\ 0&1&0&1&0\\ 0&0&0&0&0 \end{bmatrix}$$

DM_{p_4} after the delivery of m_2

$$\begin{bmatrix} 1&1&0&1&0\\ 0&1&0&0&0\\ 0&0&0&0&0\\ 0&1&0&1&0\\ 0&0&0&0&0 \end{bmatrix}$$

DM_{p_1} after the delivery of m_3

$$\begin{bmatrix} 1&1&0&1&0\\ 0&1&0&0&0\\ 1&1&1&1&0\\ 0&1&0&1&0\\ 0&0&0&0&0 \end{bmatrix}$$

DM_{p_3} after the delivery of m_4

$$\begin{bmatrix} 1&1&0&1&0\\ 0&1&0&0&0\\ 1&1&1&1&0\\ 0&1&0&1&0\\ 1&1&1&1&2 \end{bmatrix}$$

DM_{p_5} after the delivery of m_5

$$\begin{bmatrix} 1&1&0&1&0\\ 0&1&0&0&0\\ 1&1&2&1&1\\ 0&1&0&1&0\\ 0&1&0&0&1 \end{bmatrix}$$

DM_{p_3} after the delivery of m_6

$$\begin{bmatrix} 1&1&1&1&2\\ 1&1&1&1&0\\ 1&1&0&1&0\\ 0&1&0&1&0\\ 0&1&0&0&0 \end{bmatrix}$$

SM_{p_5} after the delivery of m_5

$$\begin{bmatrix} 1&1&2&1&1\\ 1&1&0&1&1\\ 0&1&0&1&0\\ 0&1&0&0&0\\ 0&1&0&0&0 \end{bmatrix}$$

SM_{p_3} after the delivery of m_6

Figure 6.13: Dependency and stability matrices for Figure 6.7.

most sensible action is for q to ignore stale messages. Hence, we assume that the recovery component implements a mechanism to recognize stale messages: this can be accomplished, for instance, by assigning to each instance of a process a unique *incarnation number* [EZ92], and by requiring each message to be tagged with the current incarnation number of the sender.

In the following, we describe the recovery procedure assuming that no checkpoints are taken by p_{old} during the execution preceding the failure. The changes to this procedure caused by the use of checkpoints are discussed in Section 6.8.

Recovery procedure for process p: p_{new} is given p's initial state and then executes the following two phases:

Phase 1: p_{new} queries the other processes in the system in order to rebuild the determinant log kept by p_{old} before failing. In particular, p_{new} collects in set $\mathcal{M}$ a (not necessarily proper) subset of the determinants of the delivery events executed by p_{old}. The recovery procedure guarantees that if $\#m \notin \mathcal{M}$ by the end of Phase 1, then no process will ever become a member of $Depend(m)$ unless p_{new} joins $Depend(m)$ first:

$$\#m \notin \mathcal{M} \Rightarrow (\Box(Depend(m) = \emptyset)\ \mathcal{U}\ (p_{new} \in Depend(m))) \quad (6.10)$$

where $\mathcal{U}$ is the temporal "unless" operator. In addition to the determinants in $\mathcal{M}$, p collects the determinants necessary to guarantee that p_{new} satisfies Property 4.10. Let $\#m'$ be the determinant in $\mathcal{M}$ with the highest receive sequence number value. To satisfy Property 4.10, p_{new} must collect a copy of all the determinants $\#m$ such that $|Log(m)| \leq f$

that correspond to delivery events that causally preceded the delivery of message m' by p_{old}.

Phase 2: Process p collects from the message logs of its parents the messages associated with each determinant $\#m$ in $\mathcal{M}$. Process p delivers each of these messages according to their receive sequence number. If $\mathcal{Q}$ denotes the set containing those processes that are operational during the time p executes the recovery procedure, then Phase 2 is concluded when each message in $\mathcal{M}$ has been delivered, and p has re-sent all messages that it previously sent to the processes in $\mathcal{Q}$ before failing.

Proving termination of the recovery procedure requires to prove that both Phase 1 and Phase 2 terminate. In order to do so, we need to make the following assumptions:

(i) Once a process crashes it eventually starts executing the recovery procedure.

(ii) Between the time a process crashes and it finishes executing the recovery procedure only a finite number of process crashes can occur.

(iii) If a process p sends a message m to a process q an unbounded number of times and process q does not fail an unbounded number of times, then q will eventually deliver m.

We discuss Phase 1 in Section 6.7.1. Assuming termination of Phase 1, we prove termination of Phase 2 with the following theorem.

Theorem 6.8 *If a failed process p terminates Phase 1 of its recovery procedure, then p will also eventually terminate Phase 2.*

Proof: Assume that process p terminates Phase 1 of its recovery procedure. The only scenario in which process p can block during Phase 2 of its recovery procedure occurs if p is waiting to receive a message m such that $\#m$ is in $\mathcal{M}$ by the end of Phase 1. We argue that m will eventually be sent to p.

Let *Past*(m) be the partially ordered set (POSET) consisting of the send event for m and the send events that causally precede the send event of m. We show that m will be sent by induction on the depth of *Past*(m).

If the depth of *Past*(m) is 1, then it consists of the single send event $send_q(m, p)$. If q has not crashed since it previously sent m, then m is in the send log of q and so q will resend m. If q has crashed since it sent m, then by Assumption (i) q will begin executing its recovery procedure. Note that since *Past*(m) consists of only a single send event, m was sent in the initial state interval of q. Hence by piecewise determinism and the finite number of failures assumption, m eventually will be in q's send log and so p will receive m.

Assume that the theorem holds for any POSET *Past*(m') with a depth less than d, and assume that *Past*(m) has depth d. Let $\{send_{q_1}(m_1, q), send_{q_2}(m_2, q), \ldots, send_{q_k}(m_k, q)\}$ be the send events that immediately precede $send_q(m, p)$ in *Past*(m). By the induction hypothesis and Assumptions (i) and (ii), each distinct process in $\{q_1, q_2, \ldots, q_k\}$ will recover to a point where $\{m_1, m_2, \ldots, m_k\}$ are in the send logs. By Assumptions (ii) and (iii) q will eventually receive the the messages $\{m_1, m_2, \ldots, m_k\}$ and by piecewise determinism will re-send m and re-enter m in its send log. Hence, by Assumptions (ii) and (iii) p will eventually receive m. □

If the recovery procedure terminates, then it does produce a consistent global state, as we show below.

Lemma 6.1 *For all messages m such that Depend(m) $\neq \emptyset$ and $m.source = p$, if p is operational and not executing the recovery protocol, then p has previously sent m.*

Proof: If p has never failed, then since *Depend*(m) $\neq \emptyset$, p has previously sent m. Otherwise, p has failed at least once, in which case m must have been sent either by the current instance p_{new} of process p or by an earlier instance p_{old} of p that later failed. Since the lemma obviously holds when m is sent by p_{new}, consider the case in which m was sent by p_{old} in state interval $p_{old}[i]$. We argue that p_{new} also sends m in $p_{new}[i]$ while executing the recovery procedure.

If $i = 0$, then, by piecewise determinism, p_{new} resends m. If $i > 0$, let $q \in$ *Depend*(m), and let m_j denote the message that starts the state interval $p[j], j \leq i$. If $\#m_j$ is stable, then by definition $\#m$ is available. Otherwise, from the definition of *Depend*(m), *Depend*(m) $\subseteq$ *Depend*(m_j). Hence, $q \in$ *Depend*(m_j), and, from Theorem 6.1 and the fact that $|Log(m_j)| \leq f$, it follows that $q \in Log(m_j)$. Then, $\forall j : 1 \leq j \leq i : \#m_j$ is available. Since *Depend*(m_j) is not empty, the recovery procedure guarantees that, by the end of Phase 1, p_{new} collects in $\mathcal{M}$ each determinant $\#m_j$. By the end of Phase 2 of the recovery procedure all the messages with determinant in $\mathcal{M}$ are delivered by p_{new} in increasing receive sequence number order. Hence, by determinism p_{new} reaches again state interval $p[i]$ and sends message m. □

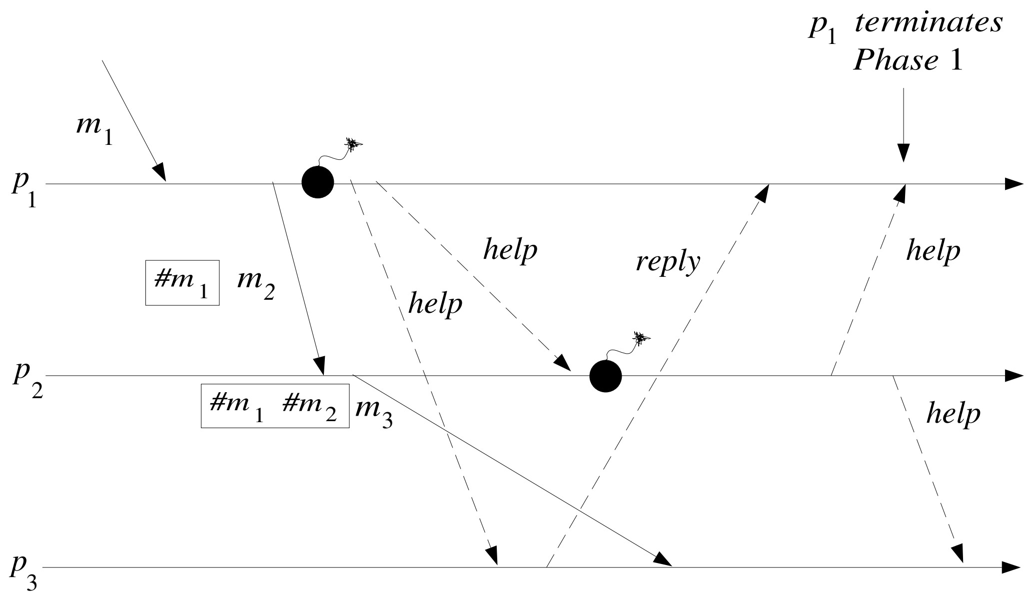

Figure 6.14: Precess p_2 fails while p_1 executes its recovery procedure.

Theorem 6.9 *Consider any run of an FBL protocol. If at some point no process is faulty and those that have previously failed have terminated the recovery procedure, then the global state is consistent.*

Proof: We show that for all pairs of non-faulty processes p and q, p and q are mutually consistent. Consider a message m delivered by q. By definition $q \in$ *Depend*(m), and by Lemma 6.1 p has previously sent m. Hence, by definition, p and q are mutually consistent.□

6.7.1 Discussion

In order to recover the system to a consistent global state, it is critical that the recovery procedure guarantees that, for each recovering process, Property 6.10 holds at the end of Phase 1. Designing a recovery procedure that provides this guarantee, however, is not trivial. This is because, while a recovering process is trying to collect its determinants, other processes may concurrently fail, causing the system to enter an inconsistent state. Consider for instance the following naîve implementation of Phase 1 of the recovery procedure:

1. When a process recovers, it sends a "help" message to all the other processes in the system.

2. When a process receives a "help" message:

   (a) Finds the determinant $\#m$ in its determinant log with the highest receive sequence number.

   (b) Forwards to the recovering process in a "reply" message all the non-stable determinants in its determinant log that correspond to delivery events causally preceding $\#m$.

3. Phase 1 terminates when the recovering process has received from each process in the system either a "reply" message, or a "help" message.

Figure 6.14 shows an execution for which, when $f = 2$, the above implementation of Phase 1 violates Property 6.10, and leaves the system in an inconsistent global state, even if determinants are piggybacked on application messages in order to satisfy Property 4.10.

Note that if process p_3 delivers m_3, then p_3 becomes a member of $Depend(m_1)$. Thus, in order to constitute a consistent global state, p_1 needs to be recovered to the state interval that begins with the delivery of m_1; therefore, by the end of Phase 1, p_1 must have received a copy of $\#m_1$ from either p_2 or p_3.

If we use the above implementation of the recovery procedure, however, p_1 will conclude Phase 1 of its recovery without receiving $\#m_1$. To see why, note that p_3 receives the "help" message from p_1 before receiving $\#m_1$ piggybacked on m_3, and therefore does not forward $\#m_1$ to p_1 in its "reply" message. Furthermore, process p_2 fails before sending a "reply" message to p_1, and so the copy of $\#m_1$ kept in p_2's volatile memory is lost.

Hence, since p_1 cannot be recovered to the state interval that begins with the delivery of m_1, by delivering m_3 p_3 will become an orphan process.

Several schemes have been proposed in order to guarantee that Phase 1 terminates and that Property 6.10 holds for each recovering process at the end of Phase 1 of its recovery procedure. In [JZ90], no correct process delivers any application message while a process is executing Phase 1 of its recovery procedure. At the end of Phase 1, the recovering process sends to all processes the value of the highest receive sequence number among all the determinants in $\mathcal{M}$. This receive sequence number corresponds to the latest delivery event to which the failed process can be recovered. Knowing this value, non-faulty processes can discard all the received application messages that depend on an unrecoverable delivery event. Using this scheme in the execution in Figure 6.14, process p_3 would not deliver m_3, since it would learn that p_1 can not be recovered to deliver m_1. This scheme is refined in [Eln93] using the fact that a process q, when it forwards a "reply" message and receives the corresponding acknowledgment, knows that the recovering process has received all the determinants in q's "reply" message. Let p denote the recovering process, and let rsn_h denote the highest receive sequence number of any determinant $\#m$ of a delivery event of p such that q forwards $\#m$ to p in its "reply" message. Process q can safely deliver all application messages that depend only on delivery events of p whose determinant has a receive sequence number rsn such that $rsn \leq rsn_h$. However, q must still wait for the conclusion of Phase 1 of p's recovery before delivering any other application message.

Elnozahy [Eln95] presents a scheme that does not require correct processes to delay the delivery of any application message while waiting for a process to complete Phase 1 of its recovery procedure. This solution requires a recovering process executing Phase 1 to restart Phase 1 whenever it detects the failure of another process.

In Section 7.2 we sketch a new scheme, based on Chandy and Lamport's snapshot protocol [CL85], that also does not delay the delivery of application messages. In addition to guaranteeing the enforcement of Property 6.10, this scheme integrates in a novel way failure detection with checkpointing in order to expedite recovery.

6.8 Checkpointing

In order to expedite recovery, processes can periodically checkpoint their state to stable storage. FBL protocols can be used with both independent and coordinated checkpointing. We discuss these options in the following.

6.8.1 Independent Checkpointing

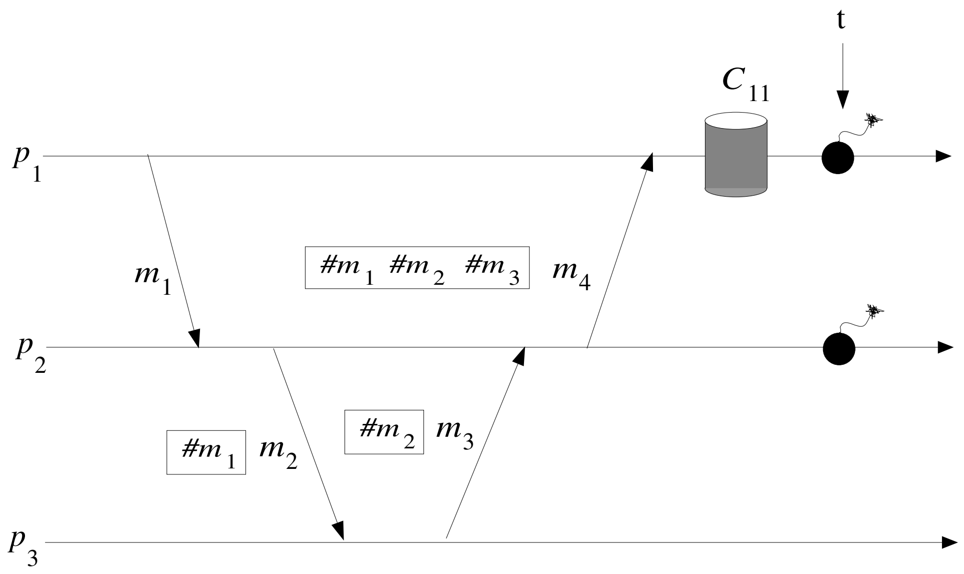

Figure 6.15: Independent checkpoints in FBL.

When taking an independent checkpoint in FBL, a process p needs to save on stable storage, in addition to its local state, the following data:

1. the contents of its message log;

2. the contents of its determinant log;

3. the information necessary to determine when to garbage collect the message and determinant log. Garbage collection can be achieved using techniques similar to those described in [EZ92].

Saving the message and determinant logs is necessary in order to guarantee that no process will ever need to roll-back past its latest checkpoint in order for the system to reach a consistent global state after a failure.

To illustrate this point, consider how an FBL protocol for $f = 2$ operates in the case of the execution shown in Figure 6.15. Suppose process p_1 does not save its message log on stable storage when it takes checkpoint C_{11}. Then, when p_1 and p_2 fail at time t, the content of m_1, which is needed to recover process p_2, will not be available. Message m_1 can only be regenerated by first rolling back process p_1 to a state preceding the sending of m_1, and by then deterministically reproducing the execution that led p_1 to send m_1. Even if p_1 has saved its determinant log on stable storage, this requires process p_1 to roll back past its latest checkpoint.

Suppose instead that p_1 does not save its determinant log on stable storage when it takes checkpoint C_{11}. Note that p_1 and p_2 are the only two processes that have a copy of $\#m_2$ in their determinant logs. Then, when p_1 and p_2 fail at time t, the determinant of m_2 is lost, and m_3 becomes an orphan message. Since the state of p_1 saved in C_{11} reflects the delivery of m_3, p_1 must be rolled back past its latest checkpoint in order for the system to reach a consistent global state.

6.8.2 Coordinated Checkpointing

The coordinator-based scheme proposed in Manetho [Eln93] to coordinate checkpoints can be used, unchanged, by FBL protocols. Experiments conducted in Manetho show that the the difference in performance between coordinated and independent checkpoints is negligible. The main advantage of coordinated checkpointing is that it does not require the implementation of a potentially complex mechanism for garbage collecting data saved on stable storage.

In Section 7.2 we briefly discuss a novel way to use coordinated checkpointing to expedite recovery from a failure.

6.8.3 Checkpointing and Recovery

Phase 1 of the recovery procedure for process p described in Section 6.7 must be modified in order to account for checkpoints. If process p takes coordinated checkpoints during its execution, then p can be restarted from its latest checkpointed state rather than from its initial state. Furthermore, p needs to collect only the determinants of the delivery events preceding the failure that were originally executed after the latest checkpoint. If p takes independent checkpoints, then, in addition to its latest checkpoint, p must also be given the contents of the determinant log and message log saved on stable storage at the time of the latest checkpoint.

6.9 Communication with the environment

All message-logging protocols must address the problem of recovering processes that communicate with the environment, such as sensors, terminals and actuators. It is natural to model such communication in terms of the sending and delivery of messages. Since one end of the communication is not a process, the schemes used to guarantee that such messages will be replayed in the correct order during recovery are usually different from the ones used for messages exchanged between processes. It is not sound, in general, to assume that the environment will be willing to maintain the determinant and message logs necessary to implement an optimal message-logging protocol. FBL protocols, therefore, treat communication with the environment as follows:

Input Messages: A message from the environment cannot in general be regenerated through determinism. Hence, the message must be stored in stable storage. Two approaches are possible:

1. The content of the message can be logged on stable memory by the receiver process. Such process must not send messages until the logging is complete. The drawback of this approach is that it introduces blocking.

2. The content of the message can be piggybacked on application messages, using the FBL's scheme to disseminate determinants, until at least $f+1$ process have a logged a copy of it in their volatile memory. The drawback of this approach is that the content of the message being piggybacked can be quite large.

Output Messages: The environment cannot in general be expected to roll-back in order to reconstitute a consistent global state after a process failure. Hence, when a process p wants to send a message m to the environment, a special output commit protocol must be run to ensure that, in case of a failure, p will never roll back to a state prior to the one in which m was sent. In FBL, the output commit protocol simply consists of p saving to stable storage the contents of $\mathcal{DS}_p$ before sending m. By doing so, p makes stable all the determinants needed to reproduce, during recovery, the causal path that led p to the state in which m was sent. The same scheme is also used by Manetho [Eln93].

6.9.1 Discussion

FBL's output commit protocol has several desirable properties:

- It does not require p to synchronize with the other processes in the system before sending m.
- It writes only a small amount of data to stable storage. Furthermore, stable storage needs to be accessed only by p.

In contrast, output commit protocols for optimistic protocols typically require the sender of a message to the environment to synchronize with the other processes, in order to guarantee that the message is sent from a recoverable state. This synchronization requires extra communication, and often results in several processes copying their volatile logs to stable storage.

Pessimistic protocols use an output commit protocol even simpler than the one used in FBL. In fact, communication with the environment in pessimistic protocols is handled the same way as normal inter-process communication: before sending a message, the sender waits until the determinants of all the messages it delivered are stable. The price for this simplicity is of course that pessimistic protocols may block even when performing inter-process communication.

An undesirable property common to all the above output commit protocols is that they introduce blocking. This is true even for optimistic and causal protocols, that allow for non-blocking interprocess communication. In Section 7.2 we briefly discussed a different output commit protocol, that promises to achieve non-blocking communication with the environment.

Chapter 7

Conclusions

This chapter reviews and discusses in Section 7.1 the contributions of this dissertation, and presents in Section 7.2 directions for future research.

7.1 Contributions

This dissertation contains four main contributions:

1. It derives, in Chapter 4, the first specification of message logging, thereby providing a sound theoretical foundation for this technique.

2. It introduces, in Chapter 4, the new notion of causal message logging, which shows the profound relationship that exists between message logging and causality.

3. It proposes, in Chapter 5, a notion of optimality for message-logging protocols.

4. It presents, in Chapter 6, family-based logging, a novel message-logging technique that results in protocols that are causal and optimal. Furthermore, FBL protocols can be parameterized depending on the maximum number of concurrent failures they must tolerate. Hence, applications only pay the cost of the fault-tolerance they really need.

A "meta-contribution" of this thesis is to reiterate the role that a formal characterization of a problem can play in deriving new and more efficient ways to solve it. The specification of the condition guaranteeing that no orphans can be created during an execution allowed us initially to realize that a pessimistic/optimistic duality was not intrinsic to message logging. Then, the specification immediately guided us to derive the piggybacking scheme used in family-based logging to prevent orphans without introducing blocking.

7.2 Future Directions

We plan to extend the results of this dissertation in several directions.

7.2.1 Combining FBL with Asynchronous Receiver-Based Logging

Recovery in message logging requires replaying to the recovering process the messages it received before failing. This makes recovery in message logging typically slower than in active replication or primary-backup. This undesirable characteristic of message logging can be accentuated by using an FBL protocol. FBL protocols, in order to avoid at the same time both blocking and orphans, log $\#m$

in the determinant logs of descendants of the process p that delivers m. If p fails, then, before replaying m to p, it is first necessary to receive $\#m$ from one of p's descendants, adding to the time necessary to complete recovery. In order to address this problem we plan to use a novel combination of FBL and asynchronous receiver-based logging of determinants. In this scheme, p asynchronously writes $\#m$ to stable storage — as if using an optimistic receiver-based protocol. However, in order to avoid orphans, p uses FBL to guarantee that $\#m$ cannot be lost before the write to stable storage completes. If p fails, then it can immediately begin recovery by replaying the messages whose determinants are logged on stable storage, while waiting to receive those determinants that are only logged at one of its descendants.

7.2.2 Communication with the Environment

All existing message-logging protocols, as we noted in Section 6.9.1, may force a process to block before sending a message to the environment. The current design of FBL protocols is no exception. Hence, we are developing an approach that employs techniques used in primary-backup protocols [BMST92] in order to achieve FBL protocols that do not block when communicating with the environment.

7.2.3 Combining Checkpointing with Recovery

We are considering a novel approach that integrates coordinated checkpointing with recovery. In this approach, a recovering process p, after being restored to its latest checkpointed state, begins a distributed snapshot [CL85]. We believe that this approach can be used to guarantee recovery without forcing correct processes to delay the delivery of application messages. Furthermore, with this approach p does not need to restore in its determinant log all the determinants of delivery events executed by processes that have successfully checkpointed their state as part of the snapshot protocol.

7.2.4 Fault-Tolerance for DSM Architectures

We are extending the results presented in this dissertation to distributed shared memory (DSM) architectures. The goal is to provide a unified framework for expressing the consistency conditions that any distributed shared memory system must obey, independent of the specific memory coherency model it provides. We hope to show that these conditions can be easily specialized for each individual memory coherency model to obtain fault-tolerant protocols that are *optimal* according to definitions similar to those given in Chapter 5. Some preliminary results in this direction are presented in [AM94a].

Appendix A

Three FBL Logging Components

This Appendix contains a detailed description of an implementation of the logging components of Π_{Det}, $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} assuming a system $\mathcal{N}$ of n processes, of which at most f can fail concurrently. Section A.1 discusses the data structures used by the three implementations. Sections A.2, A.3 and A.4 describe the logging component of Π_{Det}, $\Pi^{+}_{|Log|}$ and Π^{+}_{Log}, respectively.

A.1 Data Structures

The implementations of the logging components of Π_{Det}, $\Pi^{+}_{|Log|}$ and Π^{+}_{Log} require each process p to maintain the following data structures.

Send sequence number: SSN_p is an integer, used to uniquely identify and order each message sent by p. SSN_p is initialized to 0.

Ssn Table: $SsnTable_p$ is an n-vector of send sequence numbers. $SsnTable_p[q]$ records the highest send sequence number of any message sent by process q

and delivered by process p. The entries of $SsnTable_p$ are initialized to 0.

Dependency Matrix: DM_p is process p's copy of the $n \times n$ matrix discussed in Section 6.6.1. The entries of DM_p are initialized to 0. As discussed in Section 6.6.1, DM_p is used to compute both $Log(m)_p$ and $|Log(m)|_p$.

Message Log: $MsgLog_p$ is a set, initially empty, of elements of the form

$$e = (data, ssn, dv, dest).$$

$MsgLog_p$ contains an entry for each of the messages sent by process p. In particular, if e is the entry of $MsgLog_p$ corresponding to a message m sent by process p, then $e.data = m.data$, $e.ssn = m.ssn$ and $e.dest = m.dest$. Field $e.dv$ is an n-vector of integers that is used to update p's estimate of $Log(m)$ and $|Log(m)|$ when p receives an acknowledgment for m. The entries of $e.dv$ are initialized as follows:

- $\forall q \neq m.dest$: $e.dv[q] = \mathrm{DM}_p[p, q]$
- $e.dv[m.dest] = \mathrm{DM}_p[p, m.dest] + 1$

Stable: $Stable_p$ is an n-vector of receive sequence numbers. For all processes q in $\mathcal{N}$, $Stable_p$[q] contains the highest receive sequence number of any message m delivered by q that process p estimates to be stable. The entries of $Stable_p$ are initialized to 0.

Determinant Log: $DetLog_p$ is a set, initially empty, of elements of the form

$$e = (source, dest, ssn, rsn, logged\_at).$$

$SSN_p \leftarrow SSN_p{+}1$
$m.data \leftarrow$ data
$m.ssn \leftarrow SSN_p$
$m.piggyback \leftarrow \emptyset$
for all $e \in DetLog_p$ **st** $((e.rsn > Stable_p[e.dest]) \wedge (e.rsn > DM_p[m.dest, e.dest]))$
  $m.piggyback \leftarrow m.piggyback \cup \{(e.source, e.dest, e.ssn, e.rsn)\}$
$DM_p[p, m.dest] \leftarrow DM_p[p, m.dest]{+}1$
$SendLog_p \leftarrow SendLog_p \cup \{(m.data, SSN_p, DM_p[p,*], m.dest)\}$
$DM_p[p, m.dest] \leftarrow DM_p[p, m.dest]{-}1$
Add_Eval_Help()
send m **to** $m.dest$

Figure A.1: Process p sends m to process $m.dest$

Add_Eval_Help()
  skip

Figure A.2: Procedure *Add_Eval_Help*

> $DetLog_p$ contains the determinant of each delivery event $deliver_m(m.dest)$ for which p is a member of $Log(m)$. If e is the entry of $DetLog_p$ corresponding to $\#m$, then $e.source = m.source$, $e.dest = m.$dest, $e.ssn = m.ssn$ and $e.rsn = m.rsn$. Field $e.logged\_at$ is a set, containing process p's view of the membership of $Log(m)$: p updates $e.logged\_at$, until $|Log(m)|_p > f$. Notice that $e.logged\_at = Log(m_p)$

Additional data structures are needed to implement $\Pi^+_{|Log|}$: they are described in Section A.3.1.

receive m
if $m.type =$ "Failed-Recovering"
  $Recover(m.source)$
else if $(m.ssn \leq SsnTable_p[m.source])$
  acknowledge and discard m
else if $(m.ssn > SsnTable_p[m.source])$
  $DM_p[p,p] \leftarrow DM_p[p,p] + 1$
  $SsnTable_p[m.source] \leftarrow m.ssn$
  $m.WDV \leftarrow Process\_Eval\_Help(m.piggyback)$
  $Update\_Logged\_Dets(m.source, m.piggyback)$
  $Log\_New\_Dets(m.source, m.ssn, m.piggyback)$
  $Update\_D(m.source, m.WDV)$
  $Update\_Eval\_Help(f)$
  deliver $m.data$

Figure A.3: Process p delivers message m

array [1..n] of int *Process_Eval_Help*(PBack: list of determinants)
  for all $i \in \mathcal{N}$
    $m.WDV[i] \leftarrow 0$
    $\mathcal{E} \leftarrow \{e \in PBack \mid e.dest = i\}$
    if $\mathcal{E} \neq \emptyset$
      Let l be the element of $\mathcal{E}$ with the highest receive sequence number
      $m.WDV[i] \leftarrow l.rsn$
  return $m.WDV$

Figure A.4: Procedure *Process_Eval_Help*

Update_Logged_Dets (source: proc_id; PBack: list of determinants)
  for all $e \in PBack$ **st** $(|e.logged\_at| \leq f) \wedge$
  $(\exists e' \in DetLog_p$ **st** $((e'.dest = e.dest) \wedge (e'.rsn = e.rsn))))$
    $e'.logged\_at \leftarrow e'.logged\_at \cup \{source\}$

Figure A.5: Procedure *Update_Logged_Dets*

Log_New_Dets (source: proc_id; ssn: int; PBack: list of determinants)
$DetLog_p \leftarrow DetLog_p \cup \{(source, p, ssn, DM_p[p,p], \{p\})\}$
for all $e \in PBack$ **st** $((|e.logged\_at| \leq f) \wedge$
$(\nexists e' \in DetLog_p$ **st** $((e'.dest = e.dest) \wedge (e'.rsn = e.rsn))))$
$DetLog_p \leftarrow DetLog_p \cup \{(e.source, e.dest, e.ssn, e.rsn, \{source, e.dest, p\})\}$

Figure A.6: Procedure *Log_New_Dets*

$Update\_D$ (source: proc_id; WDV: array[1..n] of int)
for all $i \in \mathcal{N}$
$DM_p[source, i] \leftarrow$ **max** $(DM_p[source, i], WDV[i])$
$DM_p[p, i] \leftarrow$ **max** $(DM_p[p, i], WDV[i])$
$DM_p[i, i] \leftarrow$ **max** $(DM_p[i, i], WDV[i])$

Figure A.7: Procedure *Update_D*

$Update\_Eval\_Help$(f: int)
$Compute\_Stable(f : int)$

Figure A.8: Procedure *Update_Eval_Help*

Let $l \in SendLog_p$ **st** $l.ssn = ssn$
$DM_p[l.dest, *] \leftarrow$ **max** $(DM_p[l.dest, *], l.dv)$
for all $e \in DetLog_p$ **st** $((|e.logged\_at| \leq f) \wedge (e.rsn \leq l.dv[e.dest])$
$e.logged\_at \leftarrow e.logged\_at \cup \{l.dest\}$
$Update\_Eval\_Help(f)$

Figure A.9: Transport protocol informs process p of ack(ssn)

$Compute\_Stable$ (f: int)
for all $i \in \mathcal{N}$
$Stable_p[i] \leftarrow (f+1)st\ largest\ entry\ in\ DM_p[*, i]$

Figure A.10: Procedure *Compute_Stable*

A.2 The Logging Component of Π_{Det}

Figures A.1 through A.10 give a pseudo-code description of the logging component of Π_{Det}.

Figures A.1 and A.2 contain the steps taken by the logging component when a process p executes $send_p(m, m.dest)$. The field $m.piggyback$ contains all the non-stable determinants in $DetLog_p$ that p estimates process $m.dest$ may not have yet seen. Let e be the entry in DetLog_p corresponding to a determinant $\#m'$. Process p estimates that $\#m'$ is stable by comparing the value of $e.rsn$ with the value of $Stable_p[e.dest]$. In order to determine if $m.dest \notin Depend(m')$, process p uses the contrapositive of Condition 6.9.a, comparing $e.rsn$ with p's local estimate of $WDV_{m.dest}[e.dest]$, namely $DM_p[m.\mathrm{dest}, e.dest]$. Process p then adds to $MsgLog_p$ the entry for message m and executes procedure *Add_Eval_Help*. The purpose of this procedure is to piggyback to m any information on $|Log(m)|_p$ or $Log(m)_p$ that p may want to send to $m.dest$ in addition to the determinants. In Π_{Det}, p piggybacks to $m.dest$ only the determinants, and *Add_Eval_Help* consists of a *skip* operation.

Figures A.3 through A.8 contain the steps taken by the logging component when a process p executes $deliver_p(m)$. Procedure *Process_Eval_Help* processes the information piggybacked on m: in Π_{Det}, *Process_Eval_Help* returns a vector $m.WDV$, such that $m.WDV[i]$ is the highest receive sequence number of any determinant $\#m' \in m.piggyback$ for which $m'.dest = i$.

Procedure *Update_Logged_Dets* (shown in Figure A.5) uses the information in *m.piggyback* to update the estimate of *Log*(m') for those determinants $\#m'$ that are already in $DetLog_p$ when m is delivered. Procedure *Log_New_Dets* (shown in Figure A.6) first adds to $DetLog_p$ the determinant for event $deliver_p(m)$; then adds to $DetLog_p$ all the determinants in *m.piggyback* that p did not previously receive.

Procedure *Update_D* uses *m.WDV* to update DM_p. First, process p updates $DM_p[m.source, *]$ — process p's estimate of the weak dependency vector of *m.source* — to indicate that process *m.source* knows of all the delivery events encoded in *m.WDV*; then, p updates its own weak dependency vector $DM_p[p, *]$ to account for the determinants piggybacked on m. Finally, p updates the elements on the diagonal of DM_p to indicate that each process i must have delivered at least $m.WDV[i]$ messages.

Before delivering m, process p calls procedure *Update_Eval_Help* (shown in Figure A.10) that updates auxiliary data structures needed by the protocol. In Π_{Det}, process p must only recompute the entries of vector $Stable_p$. Process p considers a message m delivered by process i to be stable if more than f entries of $DM_p[*, i]$ are greater than $m.rsn$. The highest receive sequence number of any such message is simply given by the $(f+1)st$ largest value in column $DM_p[*, i]$.

Figure A.9 shows the steps taken by the logging component when a process p receives an acknowledgment for a message m. Process p updates DM_p, $RsnLog_p$ and $Stable_p$ to indicate that process $m.dest$ has received message m and the information piggybacked on it.

```
Add_Eval_Help()
  m.evhelp ← SM_p
```

Figure A.11: Procedure *Add_Eval_Help*

A.3 The Logging Component of $\Pi^{+}_{|Log|}$

A.3.1 Data Structures

In addition to the data structures introduced in Section A.1, protocol $\Pi^{+}_{|Log|}$ requires each process to maintain the following matrix:

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

A.3.2 The Logging Component

The structure of $\Pi^{+}_{|Log|}$ is similar to that of Π_{Det}. In fact, our implementation of $\Pi^{+}_{|Log|}$ uses a large subset of the pseudo-code for Π_{Det}. The differences between Π_{Det} and $\Pi^{+}_{|Log|}$ are shown in Figures A.11 through A.17.

In $\Pi^{+}_{|Log|}$, in addition to non-stable determinants in its DetLog, each process p piggybacks on every message the current value of SM_p. Hence, Procedure *Add_Eval_Help*, shown in Figure A.11, piggybacks on each message the current value of SM_p.

receive m
if $m.data =$ "q recovering"
  $Recover(q)$
else if $(m.ssn \leq SsnTable_p[m.source])$
  acknowledge and discard m
else if $(m.ssn > SsnTable_p[m.source])$
  $DM_p[p,p] \leftarrow DM_p[p,p] + 1$
  $SsnTable_p[m.source] \leftarrow m.ssn$
  $m.SMat \leftarrow Process\_Eval\_Help()$
  $Update\_Logged\_Dets(m.source, m.piggyback)$
  $Log\_New\_Dets(m.source, m.ssn, m.piggyback)$
  $OldWDV \leftarrow DM_p[p,*]$
  $Update\_D(m.source, m.SMat[1,*])$
  $Extract\_SMat(f)$
  $Update\_SMat(OldWDV, m.SMat)$
  $Compute\_Stable(f)$
  deliver $m.data$

Figure A.12: Process p delivers message m

array $[1..f+1,\ 1..n]$ of int $Process\_Eval\_Help()$
  $m.SMat \leftarrow m.evhelp$
  return $m.SMat$

Figure A.13: Procedure *Process_Eval_Help*

$Update\_Eval\_Help$ (f :int)
  $Extract\_SMat(f)$
  $Compute\_Stable(f)$

Figure A.14: Procedure *Update_Eval_Help*

$Compute\_Stable$ (f: int)
  $Stable_p \leftarrow SM_p[f+1,*]$

Figure A.15: Procedure *Compute_Stable*

Extract_SMat (int f)
  for $i = 1$ **to** $f + 1$
    for all $j \in \mathcal{N}$
      $SM_p[i, j] \leftarrow \mathbf{max}(SM_p[i, j], \textit{i-th largest element in } DM_p[*, j])$

Figure A.16: Procedure *Extract_SMat*

Update_SMat (WDV:array [1..n] of int; SM: array[1..$f + 1$, 1..n] of int)
  for all $j \in \mathcal{N}$
    Let i be the smallest index for which $SM[i, j] \leq WDV[j]$
      $SM[i, j] \leftarrow WDV[j] + 1$
      for $k = i - 1$ **downto** 2
        $SM[k, j] \leftarrow SM[k - 1, j]$
  for $i = 1$ **to** $f + 1$
    for all $j \in \mathcal{N}$
      $SM_p[i, j] \leftarrow \mathbf{max}(SM_p[i, j], SM[i, j])$

Figure A.17: Procedure *Update_SMat*

Figure A.12 contain the steps taken by the logging component when a process p executes $deliver_p(m)$. It is virtually identical to the *deliver*(m) operation for the previous protocol, except for the code needed to maintain the stability matrix,

Procedure *Process_Eval_Help* returns in $m.SMat$ the value of the *SMat* matrix piggybacked by the sender of m. Process p then updates the information logged in $DetLog_p$ and saves in *OldWDV* the current value of its dependency vector. This saved value will be used to determine whether p has received a determinant for the first time. Then, p updates matrix DM_p and calls procedure *Extract_SMat*, shown in Figure A.16, to obtain from DM_p a preliminary value for SM_p.

Procedure *Update_SMat* computes the final value of SM_p. First each entry of SM_p is set to the maximum of the corresponding entries of SM_p and $m.SMat$:

Add_Eval_Help()
  $m.evhelp \leftarrow DM_p$

Figure A.18: Procedure *Add_Eval_Help*

this is equivalent to setting $|Log(m')|_p$, for all $\#m'$ piggybacked on m to the maximum of the current values of $|Log(m')|_p$ and $|Log(m')|_{m.source}$. Then, p updates SM_p to reflect that, for each determinant $\#m'$ that p received for the first time, $|Log(m')|_p$ is one more than $|Log(m')|_{m.source}$. Finally, p updates SM_p to ensure that if $deliver_m(q) \rightarrow deliver_{m'}(q)$ then $|Log(m)|_p \geq |Log(m')|_p$.

Procedure *Compute_Stable* of Figure A.15 simply stores in vector $Stable_p$ the $f+1$ row of SM_p.

A.4 The Logging Component of Π^+_{Log}

Figures A.18 through A.22 give the modifications to protocol Π_α necessary to obtain Π_γ. In addition to all the non-stable determinants stored in its DetLog, with Protocol Π_γ each process p piggybacks on every message it sends the current value of DM_p.

The Protocol

Figures A.18 through A.22 give the modification to protocol Π_α necessary to obtain Π_γ.

Procedure *Add_Eval_Help*, shown in Figure A.18, allows process p to piggyback on each message the current value of DM_p.

```
receive m
if m.data = "q recovering"
   Recover(q)
else if (m.ssn ≤ SsnTable_p[m.source])
   acknowledge and discard m
else if (m.ssn > SsnTable_p[m.source])
   DM_p[p, p] ← DM_p[p, p] +1
   SsnTable_p[m.source] ← m.ssn
   m.D ← Process_Eval_Help()
   Log_New_Dets(m.source, m.ssn, m.piggyback)
   Update_Logged_Dets(m.D)
   Update_D(m.source, m.D)
   Update_Eval_Help(f)
   deliver m.data
```

Figure A.19: Process p delivers message m

```
array [1..n, 1..n] of int Process_Eval_Help()
   m.D ← m.evhelp
   returnm.D
```

Figure A.20: Procedure *Process_Eval_Help*

```
Update_Logged_Dets (Dep: array [1..n, 1..n] of int)
  for all i ∈ N
    for all j ∈ N
      if Dep[i, j] > DM_p[i, j]
        for all e ∈ DetLog_p st ((e.dest = j) ∧ (e.rsn ≤ Dep[i, j])
                                              ∧(|e.logged_at| ≤ f))
          e.logged_at ← e.logged_at ∪ {i}
```

Figure A.21: Procedure *Update_Logged_Dets*

Update_D (source: proc_id; Dep: array[1..n, 1..n] of int)
  for all $i \in \mathcal{N}$
    for all $j \in \mathcal{N}$
      if $Dep[i,j] > DM_p[i,j]$
        $DM_p[i,j] \leftarrow Dep[i,j]$
  $DM_p[p,*] \leftarrow$ **max** $(DM_p[p,*], Dep[source,*])$

Figure A.22: Procedure *Update_D*

Figure A.19 gives the code for the operation $deliver_p(m)$. It is analogous to the *deliver* operation of Π_{Det}, with additional code for processing the information contained in the dependency matrix piggybacked on m by process $m.source$.

Procedure *Process_Eval_Help* of Figure A.20 returns in $m.D$ the value of the dependency matrix piggybacked by the sender of m.

Procedure *Update_Logged_Dets* updates the estimate of $Log(m')$ for all determinants $\#m'$ that were piggybacked on message m and that p had previously received.

Procedure *Update_D* uses the information in $m.D$ to update DM_p by taking the element-wise maximum of $m.D$ and the current value of DM_p. The elements of process p weak dependency vector $DM_p[p,*]$ are set to the column-wise maximum of the elements of DM_p.

Bibliography

[AHM93] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *Proceedings of the 23rd Fault-Tolerant Computing Symposium*, pages 145–154, June 1993.

[AM94a] L. Alvisi and K. Marzullo. Deriving optimal checkpointing protocols for distributed shared memory architectures. In K. Birman, F. Mattern, and A. Schiper, editors, *Selected Papers, International Workshop in Theory and Practice in Distributed Systems*, pages 111–120. Springer-Verlag, 1994.

[AM94b] Lorenzo Alvisi and Keith Marzullo. Optimal message logging protocols. Technical Report TR94-1457, Cornell University Department of Computer Science, September 1994.

[AM95] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236. IEEE Computer Society, June 1995.

[BA+92] O. Babaoglu, L. Alvisi, et al. Paralex: An environment for parallel programming in distributed systems. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 178–187, July 1992.

[BBG83] 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.

[BJ87] Kenneth Birman and Tommy Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(2):47–76, February 1987.

[BLL90] B. Barghava, S-R. Lian, and P-J. Leu. Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In *Proocedings of the International Conference on Data Engineering*, pages 182–189, March 1990.

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary–backup protocols: Lower bounds and optimal implementations. In *Proceedings of the Third IFIP Conference on Dependable Computing for Critical Applications*, September 1992.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer System*, 9(3):272–314, August 1991.

[CL85] 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.

[DGB+96] R. Davoli, L. A. Giachini, O. Babaoğlu, S. Amoroso, and L. Alvisi. Parallel computing in networks of workstations with paralex. *IEEE Transactions on Parallel and Distributed Systems*, 1996. To appear.

[EJZ92] E.N. Elnozahy, D.B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[Eln93] E. N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. Ph.D. dissertation, Rice University, October 1993. Available as report COMP TR93-212.

[Eln95] E.N. Elnozahy. On the relevance of communication costs of rollback-recovery protocols. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 74–79, August 1995.

[EZ92] 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.

[EZ94] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers: 24 Annual International Symposium on Fault-Tolerant Computing*, pages 298–307. IEEE Computer Society, June 1994.

[Gra77] James N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmueller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1977. Lecture Notes on Computer Science 60.

[Joh93] D.B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

[JV87] T.Y. Juang and S. Venkatesan. Crash recovery with little overhead. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 454–461. IEEE Computer Society, June 1987.

[JZ87] 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.

[JZ90] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LSP82] L. Lamport, R. Shostack, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[Mat89] Friedmann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms*, pages 215–226. Elsevir Science Publishers B. V., 1989.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pages 46–57, November 1977.

[PP83] 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.

[Ran75] B. Randell. System structure for software fault-tolerance. *IEEE Transaction on Software Engineering*, SE-1(2):220–232, June 1975.

[RST91] M. Raynal, A. Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[SBY88] 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.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[Sch86] Fred B. Schneider. Abstractions for fault-tolerance in distributed systems. In *Proceedings of the Tenth World Computer Congress*, pages 727–733, September 1986.

[Sch90] Fred B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[SS92] A. Sandoz and A. Schiper. A characterization of consistent distributed snapshots using causal order. Technical Report TR92-14, Departement d'Informatique, Ecole Politechnique Fédérale de Lausanne, 1992.

[SW89] 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.

[SY85] R. B. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, April 1985.

[VJ94] S. Venkatesan and T.Y. Juang. Efficient algorithms for optimistic crash recovery. *Distributed Computing*, 8(2):105–114, June 1994.

[vN56] John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. McMarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[WF92] Y.M. Wang and W.K. Fuchs. Optimisitc message logging for independent checkpointing in message-passing systems. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 147–154, October 1992.