# Nonblocking and Orphan-Free Message Logging Protocols*

AMES GRANT
IN-32-CR
136499
P. 24

Lorenzo Alvisi**
Bruce Hoppe
Keith Marzullo§

TR 92-1317
December 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This material is based upon work supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, a National Science Foundation Graduate Research Fellowship and by grants from IBM Glendale Programming Laboratory, IBM T.J. Watson Research Center, Siemens, and Xerox Webster Research Center.
**Also supported by research fellowship number 203.12.191 of the Italian National Research Council (Consiglio Nazionale delle Ricerche).
§Current address: Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Dr. 0114, La Jolla, CA 92093-0114.

# Nonblocking and Orphan-Free Message Logging Protocols *

Lorenzo Alvisi[†] Bruce Hoppe Keith Marzullo[‡]

Cornell University Department of Computer Science
Ithaca NY 14853 USA

**Abstract**

Currently existing message logging protocols demonstrate a classic *pessimistic vs. optimistic* tradeoff. We show that the optimistic–pessimistic tradeoff is not inherent to the problem of message logging. We construct a message–logging protocol that has the positive features of both optimistic and pessimistic protocol: our protocol prevents orphans and allows simple failure recovery; however, it requires no blocking in failure–free runs. Furthermore, this protocol does not introduce any additional message overhead as compared to one implemented for a system in which messages may be lost but processes do not crash.

## 1 Introduction

Message logging protocols are a common method of building a system that can tolerate process crash failures. These protocols require that each process periodically record its local state and log the messages it received after that state. When a process crashes, a new process is created, given the appropriate recorded local state, and then sent the logged messages in the order they were originally received.

Message logging protocols are not the only techniques for making systems robust against process crashes. For example, active replication [12] or passive replication [2] are commonly-used techniques for masking failures more severe than crash failures. However, message

*This material is based upon work supported by Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2–593, a National Science Foundation Graduate Research Fellowship, and by grants from IBM Glendale Programming Laboratory, IBM T. J. Watson Research Center, Siemens, and Xerox Webster Research Center.

[†]Also supported by research fellowship number 203.12.191 of the Italian National Research Council (Consiglio Nazionale delle Ricerche).

[‡]Current address: Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Dr. 0114, La Jolla, CA 92093–0114.

logging protocols are attractive in that they are considered an inexpensive technique: they introduce little or no process replication in failure free runs and the protocols are relatively simple. Another advantage of message logging protocols is that they are readily applied to any process communication structure, while both active and passive replication are typically applied in a client-server setting. Of course, one can use either active or passive replication to implement a message logging system.

The published message logging protocols exhibit a classic tradeoff in performance: there exist *pessimistic* protocols that introduce blocking in order to simplify recovery and *optimistic* protocols that introduce no blocking but may require computation to be undone upon recovery. In this paper, we show that the optimistic–pessimistic tradeoff is not necessary for message logging protocols. We derive a message logging policy that is weaker than that used by pessimistic protocols yet is strong enough to guarantee that computation need not be rolled back. This policy is based on the notion of logging a message when the effects of its delivery are visible to some process other than the process to which it was delivered. This logging policy can be implemented without introducing additional blocking. We then give a protocol that can tolerate non-concurrent failures and discuss its performance.

The paper proceeds as follows. Section 2 describes the system model commonly assumed for message logging protocols. Section 3 discusses message logging protocols and explains the current optimistic–pessimistic tradeoff. Section 4 defines what it means for a message to be relevant, derives logging and recovery rules for relevant messages, and presents a message logging protocol that is nonblocking but creates no orphans. Section 5 discusses the performance of an implementation our protocol, and Section 6 concludes the paper.

## 2 System Model

We assume a set of processes that communicate only by exchanging messages.[1] The system is asynchronous: there exists no bounds on the relative speeds of processes, there exists no bounds on message transmission delays, and there exists no global time source. Hence, the *order* in which a process receives messages is nondeterministic.

The execution of a system of $n$ processes is represented by a *run*, which is an irreflexive partial ordering of the send events, receive events and delivery events ordered by potential causality [9]. Delivery events are local to a process and represent the delivery of a received message to the application. For any message $m$ from process $p$ to process $q$, $q$ delivers $m$ only if it has received $m$, and $q$ delivers $m$ no more than once.

[1]For simplicity, we also assume that a process never sends a message to itself.

We assume a fail-stop failure model for the processes [11] and an intermittent message loss failure model for the channels. Channels are FIFO, in that if process $p$ sends two messages to process $q$ and process $q$ receives both of them, then it will receive them in the order that $p$ sent them. Furthermore, we assume a transport protocol that provides a stronger FIFO property: if process $p$ sends $m_1$ and then $m_2$ to process $q$, then $q$ will not receive $m_2$ before receiving $m_1$.

At any point in an execution, the state of a process is determined by its initial state and the sequence of messages that it has delivered. For any message $m$ delivered by process $p$, its *receive sequence number*, denoted $m.rsn$, represents the order in which $m$ was delivered: $m.rsn = \ell$ if $m$ is the $\ell^{th}$ message delivered by $p$ [14].[2] We denote with $\sigma_p[\ell]$ the state of process $p$ after having delivered $\ell$ messages.

## 3 Issues in Message Logging

In message logging protocols, each process must record both the contents and receive sequence numbers of all the messages it has delivered in a location that will survive the failure of the process. This action is called *logging.* The process may also periodically record its local state (called *checkpointing*), thereby allowing its message log to be trimmed. For example, once process $p$ knows that state $\sigma_p[\ell]$ is logged, then all messages $m$ such that $m.rsn \leq \ell$ can be removed from its message log. Note that the periodic checkpointing of a process's state is only needed to bound the length of its message log (and hence the recovery time). For simplicity we ignore checkpointing in this paper.

Logging a message may take time, and so there is a natural design decision of whether or not a process should wait for the logging to complete before delivering the message to the application. For example, suppose that having delivered message $m$, process $p$ sends message $m'$ to process $q$. If message $m$ were not logged by the time $p$ sent $m'$, then the crash of $p$ may cause information about $m$ to be lost. Then, when a new process $p$ is initialized and replayed old logged messages, $p$ may follow an execution in which $m'$ is not sent to $q$. Hence, process $q$ would no longer be consistent with $p$ once it delivers $m'$. Such a message $m$ is called a *lost message*, message $m'$ an *orphan message* and the state of process $q$ an *orphan state* [14]. Protocols that can create orphans are called *optimistic* because the likelihood of creating an orphan is (hopefully) small.

A *pessimistic* protocol is one in which each process $p$ never sends a message $m'$ until it knows that all messages delivered before sending $m'$ are logged. Pessimistic protocols will

[2]We avoid the term "delivery sequence number" simply to avoid new terminology for an old concept.

never create orphans, and so the reconstruction of the state of a crashed process is very straightforward as compared to optimistic protocols, in which orphans must be detected and rolled back. On the other hand, pessimistic protocols potentially block a process for each message it receives. Doing so can slow down the throughput of the process, and this price must be paid even in an execution in which no process crashes.

Another design decision in message logging protocols is where each message is to be logged. An obvious choice is to log at the receiving process, since it is the receiving process that assigns the receive sequence number to an incoming message [14, 8]. Unfortunately, such a *receiver-based* protocol requires an implementation of stable storage that will survive the crash of the receiving process. Another choice, called *sender-based logging*, is to log each message $m$ in the sender's volatile storage [7]. Doing so requires the receiver to tell the sender what receive sequence number was assigned to $m$. Furthermore, the receiver does not know that $m$ is logged until it receives an extra acknowledgement from the sender indicating that the receive sequence number has been recorded. The receive sequence number can be piggybacked on the acknowledgement of message $m$, but the extra acknowledgement from the sender may require an additional message; and in pessimistic sender-based logging, the receiver still must block until it receives the final acknowledgement [5]. A further problem with sender-based logging is that multiple concurrent failures may make a process unable to recover its previous state even when a pessimistic protocol is used. However, sender-based logging is, in some sense, optimal in the number of resources it uses: it requires no stable storage and an additional process is needed only when recovery of a crashed process begins.

These two design decisions—pessimistic vs. optimistic and receiver-based logging vs. sender-based logging—are independent. In particular, there exist pessimistic receiver-based logging protocols [1, 10], optimistic receiver-based logging protocols [14, 13], pessimistic sender-based logging protocols [7, 5] and optimistic sender-based logging protocols [7, 6].

All message protocols must address the problem of communication with the environment. For input, the data must be stored in a location that is always accessible for the purpose of replay. For output, the process must be in a recoverable state before sending any message to the environment. This means that, in general, even optimistic message logging protocols may block before sending a message to the environment. Such issues are outside of the scope of this paper.

## 4 Nonblocking, Orphan-Free Protocols

In this section, we first review how recovery can be done when there are no orphans. We then develop a nonblocking protocol in which orphans cannot occur by more carefully considering the design decision: by what point must a message be logged?

### 4.1 Recovery with no Orphans

Suppose that process $q$ delivers a message $m$ from process $p$. We define another message $m'$ to *depend on* $m$ if the delivery of $m$ causally precedes the sending of $m'$.

Consider the execution of distributed system and a set of local states $\sigma_1[\ell_1], \ldots, \sigma_n[\ell_n]$, one for each process $1, \ldots, n$. We say that two states $\sigma_p[\ell_p]$ and $\sigma_q[\ell_q]$ are *pairwise consistent* if all messages from $p$ that have been delivered to $q$ by $\sigma_q[\ell_q]$ have been sent by $\sigma_p[\ell_p]$, and all messages from $q$ that have been delivered to $p$ by $\sigma_p[\ell_p]$ have been sent by $\sigma_q[\ell_q]$. The collection of local states is a *consistent global state* if all pairs of states are pairwise consistent [3].[3] With message logging protocols, an inconsistent global state arises when a lost message occurs due to optimistic recovery.

A pessimistic message logging protocol is one in which a message $m$ is always logged by the time any message $m'$ that depends on $m$ is sent. Recovery in pessimistic protocols is straightforward: the crashed process is reinitialized and replayed the old logged messages in increasing receive sequence number order. Since the process is deterministic with respect to message receipt order, it will follow the same path of execution as before. Thus, in the process of recovering, it will send the same sequence of messages as before. Any duplicate message $m$ sent during recovery is acknowledged and discarded if the destination has already received $m$.[4]

The following simple theorem shows that this recovery protocol is correct:

**Theorem 1** *Consider an execution of a pessimistic message logging protocol in which process p crashes. If process p restarts from its initial state and delivers all logged messages in receive sequence number order, then the resulting global state is consistent.*

*Proof.* We will show that the global state is consistent by showing that the recovered local state of $p$ is pairwise consistent with the local states of the other processes.

[3]This definition is different from that of [3] in that it is defined in terms of *deliver* events rather than *receive* events. Our usage is consistent with the literature on message-logging protocols.

[4]Another possibility is for acknowledgements to be logged in the same way as other messages. In this case, a process receiving a repeat message $m$ would discard $m$ without sending an acknowledgement because the message acknowledgement will be replayed. As is discussed in Section 4.2, the choice of implementation is not just a matter of efficiency—which implementation is correct depends on whether or not acknowledgements are visible to the application.

Since a process is deterministic given a sequence of message deliveries, the state that process $p$ recovers to is $\sigma_p[\ell]$ where $\ell$ is the highest logged receive sequence number. Since the protocol is pessimistic, then by definition $p$ has not sent any messages in a state $\sigma_p[\ell']$ for $\ell' > \ell$. Hence, no process $q \neq p$ has received a message that follows $\sigma_p[\ell]$. In addition, process $p$ has not received any messages that were not sent, and so $\sigma_p[\ell]$ is pairwise consistent with each other local state. □

## 4.2 Abstract Message Logging Protocol

A pessimistic protocol guarantees that a message $m$ is logged before any message that depends on $m$ is sent. This fact makes the proof of Theorem 1 straightforward. However, it is not necessary that $m$ be logged until a message that depends on $m$ is delivered, because it is at this point that the effects of $m$ will become visible to another process. Hence, we define a message $m$ to be *relevant* when a process has delivered a message that depends on $m$.

In the following protocol, we assume that message acknowledgements are never delivered to the application: that is, they are only seen by the underlying transport protocol. Thus, an application-level send is nonblocking in that the application does not block waiting for an acknowledgement from the recipient. This assumption allows us to not log acknowledgements, since they carry no information as far as the application is concerned. Note that this assumption is not fundamental, in that if it does not hold, then acknowledgements can be logged and replayed in the same way the other messages are.

A message is logged by including attributes about the message in a set $\mathcal{L}$. A message $m$ has the following five attributes: $m.source$ is the sender of $m$; $m.dest$ is the destination of $m$; $m.data$ is the data that the application $m.source$ sends to $m.dest$; $m.ssn$ is the *send sequence number*: $m.ssn = \ell$ denotes that $m$ is the $\ell^{th}$ message sent by $m.source$; $m.rsn$ is the receive sequence number. Following [7], a message $m$ is *partially logged* if the four attributes $m.source, m.dest, m.data, m.ssn$ are defined in $\mathcal{L}$, and $m$ is *fully logged* if all five attributes are defined in $\mathcal{L}$. Finally, $\mathcal{L}_p$ is a subset of messages $m \in \mathcal{L}$ such that $m.dest = p$.

**Abstract message logging protocol:** The protocol consists of a *logging policy* and a *recovery procedure*:

**Logging policy:** A message $m$ is partially logged by the time it is sent and fully logged by the time it is relevant.

**Recovery procedure:** To recover a crashed process $p$: after any messages that $p$ sent before crashing are either received or dropped due to transient channel faults, $p$ is restarted from its initial state. It is then sequentially sent the messages in $\mathcal{L}_p$ in an order that is consistent with the receive sequence numbers of the fully logged messages and that is consistent with FIFO channels for the partially logged messages.

**Theorem 2** *Consider an execution of the abstract message logging protocol given above in which a process crashes and then recovers. The resulting global state is consistent.*

*Proof:* We will show that the global state is consistent by showing that the recovered local state of the crashed process $p$ is pairwise consistent with the local states of the other processes.

Let $RM$ be the sequence of messages in $\mathcal{L}_p$ in the order that $p$ received them before crashing. Consider a message $m \in RM$ that is fully logged. Any message $m'$ before $m$ in $RM$ is also fully logged by the logging strategy: if $m$ is relevant then so are all messages $m'$ that were received before $m$. Hence, $RM$ consists of a (possibly empty) sequence of fully logged messages ordered by receive sequence number followed by a (possibly empty) sequence of partially-logged messages ordered by the constraint that the channels are FIFO. Let the subsequence $RM[1..\ell_f]$ be the fully-logged messages and the subsequence $RM[\ell_f + 1..\ell]$ be the partially-logged messages.

Consider a message $m$ that was originally sent by $p$ before crashing but is not sent by $p$ during recovery. Since $p$ is deterministic with respect to message delivery order, the recovering process will follow the same execution as before through state $\sigma_p[\ell_f]$ and so $m$ must have been sent in a state after $\sigma_p[\ell_f]$; say, $\sigma_p[\ell_f + \delta]$ where $1 \leq \delta \leq \ell - \ell_f$. However, $m$ could not have been delivered by *m.dest* since otherwise the messages $RM[\ell_f + 1..\ell_f + \delta]$ would be relevant and hence fully logged. Hence, there are no messages that were originally sent by $p$ and delivered to *m.dest* yet not resent during the recovery of $p$.

Finally, $p$ has not received any messages that were not sent. Thus, $p$'s recovered local state is pairwise consistent with each other local state. □

Theorem 2 is concerned with the crash and recovery of a single process and argues that the state of the application is reconstructed to a consistent global state. However, it does not say anything about the state of the logged messages. In particular, if the information about a logged message $m$ can be lost due to a process crash and recovery, then the protocol may only be able to tolerate a single failure in any run. If the following property holds, however, then any sequence of (process crash; process recovery) pairs can be tolerated:

**Stable log:** If a message $m$ is partially logged and will eventually be received, then $m$ will be partially logged after any process $p$ crashes and recovers. And, if $m$ is fully logged and a process $p$ crashes, then $m$ will be fully logged after $p$ recovers.

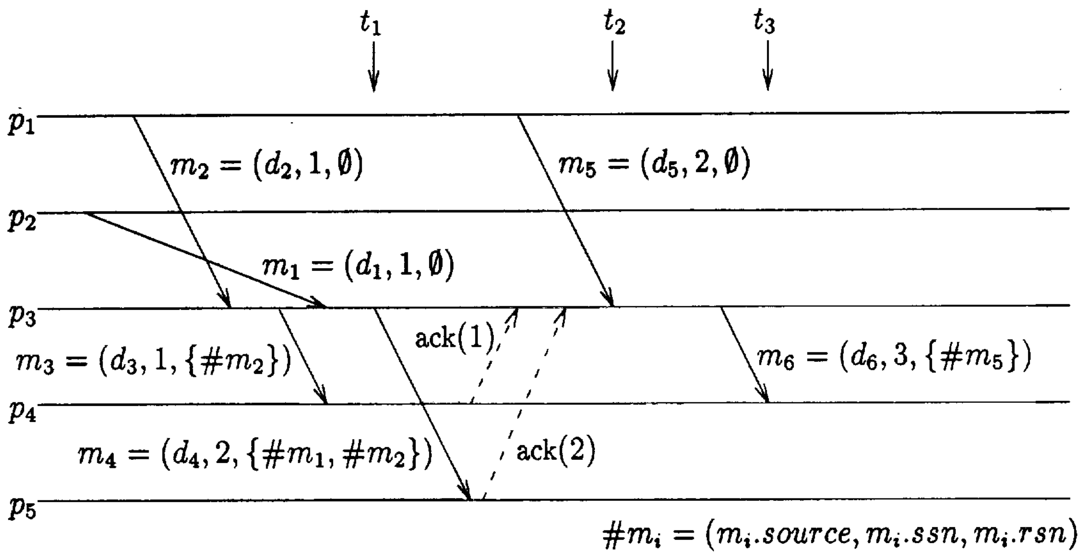
Note that we allow partially logged messages that will never be received to become unlogged. Such a message, however, is never received by any process and so can be lost without any effect.

## 4.3 Family-Based Logging

According to the abstract message logging protocol described in Section 4.2, a message $m$ must be fully logged when it becomes relevant. What process determines when a message becomes relevant? In order to answer this question, we introduce the following definitions: We say a process $p$ is a *parent* of a process $q$, and $q$ is a *child* of $p$, if $q$ delivers a message sent by $p$. Note that $p$ can simultaneously be a parent and a child of $q$.

Consider a message $m$ from process $p$ to process $q$. The relevance of $m$ is not determined by $q$, nor in general by $q$'s parent, $p$. Rather, $m$ becomes relevant when some child of $q$ delivers a message that depends on $m$. Thus, the children of $q$ determine the moment when any message delivered by $q$ must be fully logged, and so it is natural to assign the responsibility of logging $m$'s receive sequence number to the children. To do so, after delivering $m$, $q$ can piggyback $m.rsn$ on every subsequent message and $q$'s children can log these piggybacked receive sequence numbers. If $m$ become relevant, then some child $r$ must have delivered a message that depends on $m$, and so $m.rsn$ is logged at $r$. Of course, $q$ need not piggyback $m.rsn$ on every subsequent message: once $q$ receives an acknowledgement from $r$ for any message that depends on $m$, then $q$ knows that $m.rsn$ is logged at $r$. Finally, the other attributes of $m$ are already located at its sender $p$ that is a parent of $q$, and so we assign to $p$ the responsibility of partially logging $m$ before $m$ is sent. We call this logging strategy *family-based logging.*

Suppose that process $q$ fails. The messages logged at $q$'s parents and the receive sequence numbers logged at $q$'s children must be recombined to recover $q$. Thus, $q$'s children must log more than receive sequence numbers; there must be some means of matching receive sequence numbers to the corresponding messages. To do so, $q$ can piggyback triplets of the form $(m.source, m.ssn, m.rsn)$ where $m$ is the message delivered by $q$. Each triplet $(p, ssn, rsn)$ corresponds to a unique message partially logged at process $p$ with send sequence number $ssn$; and so when $q$ fails, the receive sequence numbers logged at $q$'s children can be matched with the corresponding messages logged at $q$'s parents.



$t_1$
$t_2$
$t_3$
$p_1$
$p_2$
$p_3$
$p_4$
$p_5$
$m_2 = (d_2, 1, \emptyset)$
$m_5 = (d_5, 2, \emptyset)$
$m_1 = (d_1, 1, \emptyset)$
$m_3 = (d_3, 1, \{\#m_2\})$
ack(1)
$m_6 = (d_6, 3, \{\#m_5\})$
$m_4 = (d_4, 2, \{\#m_1, \#m_2\})$
ack(2)
$\#m_i = (m_i.source, m_i.ssn, m_i.rsn)$

Figure 1: Family-Based Logging

### 4.3.1 Data Structures

The protocol requires each process $p$ to maintain the following data structures:[5]

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

**Receive sequence number:** $RSN_p$ is an integer, initially 0, used to uniquely identify and order each message delivered by $p$.

**Send log:** $SendLog_p$ is a set, initially empty, of elements of the form $e = (data, ssn, rsn, dest)$. $e \in SendLog_p$ if there exists a message $m$ sent by $p$ in state $\sigma_p[e.rsn]$ such that $m.data = e.data$, $m.ssn = e.ssn$, and $m.dest = e.dest$.

**RSN log:** $RsnLog_p$ is a set, initially empty, of elements of the form $e = (parent, ssn, rsn, child)$. $e \in RsnLog_p$ if there exists a message $m$ delivered by $p$ such that $m.source = e.parent$, $m.ssn = e.ssn$, and $m.rsn = e.rsn$. If $e.child \neq \perp$ then $m.rsn$ is logged at $e.child$.

**Receive log:** $ReceiveLog_p$ is a set, initially empty, of elements of the form $e = (parent, grandparent, ssn, rsn)$. If $e \in ReceiveLog_p$, then there exists a message $m$ delivered by

[5] $r = (a, b, ..., n)$ a record $r$ of type $a \times b \times ... \times n$, and with $r.i$ the value of field $i$ of record $r$.

process *e.parent* such that $m.source = e.grandparent$, $m.ssn = e.ssn$, $m.rsn = e.rsn$; and furthermore, there exists a message $m'$ with $m'.source = e.parent$ delivered by $p$ such that $m'$ depends on $m$.[6]

**SSN table:** $SsnTable_p$ is a vector of send sequence numbers whose entries are initialized to 0. $SsnTable_p[q]$ records the highest send sequence number of any message from $q$ delivered by $p$.

**Piggyback sequence number:** $PSN_p$ is an integer, initially 0. $PSN_p$ keeps the value of the highest receive sequence number such that the corresponding entry $e \in RsnLog_p$ has $e.child \neq \perp$. Entries $e \in RsnLog_p$ such that $e.rsn > PSN_p$ might not be logged, and will be therefore piggybacked by $p$ on the next outgoing message.

### 4.3.2 Example

As an illustration of how FBL's message logging protocol works in the absence of link failures, consider Figure 1.[7] Each message that carries data is a triple (*data, ssn, piggyback*) where *data* is the data of the message, *ssn* is the message's send sequence number and *piggyback* is the information that is piggybacked on the message.

We consider the execution of the protocol from the perspective of process $p_3$. Notice that $p_3$ piggybacks on each outgoing message $m_i$ enough information to fully log all the partially logged messages on which $m_i$ depends. In particular, consider the situation at time $t_1$: even though process $p_3$ has already piggybacked the information concerning $m_2$ on message $m_3$, $p_3$ has not yet received an acknowledgment for $m_3$, and so cannot assume $m_2$ is fully logged. Hence, $m_4$ contains the information necessary to fully log $m_1$ and $m_2$.

By time $t_2$, however, $p_3$ has received the acknowledgments for both $m_3$ and $m_4$, and therefore knows that $m_1$ is logged at $p_4$ and $m_2$ is logged at $p_5$. It then piggybacks on $m_6$ only the information necessary to fully log $m_5$. Figure 2 presents a snapshot of the system at time $t_3$.

### 4.3.3 Protocol

Figures 3 through 7 give the code for the family-based logging protocol. Due to lack of space, we rely on the example given above instead of giving a detailed explanation of the code. Briefly, Figures 3 and 4 give the code for the *send* and *deliver* operations. The

[6]We avoid "Delivery log" just as we avoided "delivery sequence number."

[7]For the sake of clarity we have included only the acknowledgments for messages $m_3$ and $m_4$. We assume that the acknowledgments for the remaining messages have not yet been received.

| *Process* | *SSN* | *RSN* | *PSN* | *SendLog* | *RsnLog* | *ReceiveLog* | *SsnTable* |
|---|---|---|---|---|---|---|---|
| $p_1$ | 2 | 0 | 0 | $(d_2,1,0,p_3)$ $(d_5,2,0,p_3)$ | $\emptyset$ | $\emptyset$ | (0,0,0,0,0) |
| $p_2$ | 1 | 0 | 0 | $(d_1,1,0,p_3)$ | $\emptyset$ | $\emptyset$ | (0,0,0,0,0) |
| $p_3$ | 3 | 3 | 2 | $(d_3,1,1,p_4)$ $(d_4,2,2,p_5)$ $(d_6,3,3,p_4)$ | $(p_1,1,1,p_4)$ $(p_2,1,2,p_5)$ $(p_1,2,3,\perp)$ | $\emptyset$ | (2,1,0,0,0) |
| $p_4$ | 0 | 2 | 0 | $\emptyset$ | $(p_3,1,1,\perp)$ $(p_3,3,2,\perp)$ | $(p_3,p_1,1,1)$ $(p_3,p_1,2,3)$ | (0,0,3,0,0) |
| $p_5$ | 0 | 1 | 0 | $\emptyset$ | $(p_3,2,1,\perp)$ | $(p_3,p_1,1,1)$ $(p_3,p_2,1,2)$ | (0,0,2,0,0) |

Figure 2: Snapshot of Execution of Figure 1 at time $t_3$

*Process p sends* data *to process q*

```
    $SSN_p \leftarrow SSN_p + 1$
    $m.source \leftarrow p$
    $m.data \leftarrow$ data
    $m.ssn \leftarrow SSN_p$
    $m.piggyback \leftarrow \emptyset$
    for all $e \in RsnLog_p$ such that $e.rsn > PSN_p$
        $m.piggyback \leftarrow m.piggyback \cup \{(e.parent, e.ssn, e.rsn)\}$
    $SendLog_p \leftarrow SendLog_p \cup \{(m.data, SSN_p, RSN_p, q)\}$
    send $m$ to $q$
```

Figure 3: Logging Protocol: Message Send

message attribute *m.piggyback* is the set of triples $(m'.parent, m'.ssn, m'.rsn)$ for messages $m'$ received by $p$ but not yet fully logged. Figure 5 gives the part of the protocol that advances the piggyback sequence number when an acknowledgement is received. Figure 6 gives the protocol a process runs when recovering, and Figure 7 gives the protocol a process executes when it is requested to send its logs to a recovering process. When $p$ recovers, it uses this protocol to collect the send logs from its parents and the receive logs from its children in order to construct the logged messages in the data structure $ReplayLog_p$, and it uses the RSN logs of its parents to reconstruct its receive log. In Figure 7, process $q$ sends a sequence of messages to the recovering process $p$ bracketed by the messages "$q$ begin replay" and "$q$ exit replay". The receives of these bracketing messages are not explicitly shown in Figure 6, but instead are denoted implicitly by the predicates "*m.source* has begun replaying" and "all processes have exited replay to $p$". Reconstruction of the logs is discussed further in Theorem 4 of Section 4.3.4.

*Process q delivers message m*

```
    **repeat**
        **receive** message $m$ from transport protocol
    **until** $(m.ssn > SsnTable_q[m.source])$
    $RSN_q \leftarrow RSN_q + 1$
    $SsnTable_q[m.source] \leftarrow m.ssn$
    $RsnLog_q \leftarrow RsnLog_q \cup \{(m.source, m.ssn, RSN_q, \perp)\}$
    **for all** $e \in m.piggyback$
        $ReceiveLog_q \leftarrow ReceiveLog_q \cup \{(m.source, e)\}$
    **deliver** $m.data$
```

Figure 4: Logging Protocol: Message Deliver

*Transport protocol informs process p of ack(ssn)*

```
    Let $l \in SendLog_p$ such that $l.ssn = ssn$
    **if** $(l.rsn > PSN_p)$
        $PSN_p \leftarrow l.rsn$
        **for all** $e \in RsnLog_p$ such that $((e.child = \perp) \wedge (e.rsn \leq PSN_p))$
            $e.child \leftarrow l.dest$
```

Figure 5: Logging Protocol: Acknowledgement Receive

### 4.3.4 Proofs

**Theorem 3** *Family-based logging is an implementation of the abstract message logging protocol.*

*Proof*: We will show this by giving a refinement mapping from the data structures of the family-based logging protocol to $\mathcal{L}$. For each process $p$ and each entry $e = (data, ssn, rsn, dest) \in SendLog_p$ there exists a message $m \in \mathcal{L}$: $m.orig = p$, $m.dest = e.dest$, $m.ssn = e.ssn$, and $m.data = e.data$. If there exists a process $q$ with an entry $e' = (grandparent, parent, ssn, rsn) \in ReceiveLog_q$ where $m.orig = e'.grandparent$ and $m.ssn = e'.ssn$, then $m.rsn = e'.rsn$; otherwise, $m.rsn = \perp$.

From Figure 3, by the time $m$ is sent by $p$, the corresponding entry for $m$ is in $SendLog_p$. Message $m$ delivered by $p$ first becomes relevant when some process $q$ is the first to deliver a message $m'$ that depends on $m$. Since it is the first such delivered message, $m'$ must have been sent by $p$ and so from Figure 4 the receive sequence number of $m$ is in $ReceiveLog_q$ by the time $m'$ is delivered. Thus, family-based logging implements the logging policy.

Note that the value of $\mathcal{L}_p$ does not reference any of the data structures local to process $p$, and so $\mathcal{L}_p$ is defined when $p$ has crashed and is recovering. Furthermore, Figure 6 references

*Failure recovery for the faulty process p*

```
Reinitialize all message logging data structures
$ReplayLog_p \leftarrow \emptyset$
send ``p crashed/recovering'' to all other processes
while $\neg$ (all processes have exited replay to $p$)
    receive $m$
    if ($m.source$ has begun replaying)
        if ($m = (source, data, ssn)$)
            if $\exists e \in ReplayLog_p$ such that $((e.orig = m.source) \wedge (e.ssn = m.ssn))$
                $e.data \leftarrow m.data$
            else $ReplayLog_p \leftarrow ReplayLog_p \cup \{l\}$ where
                $e.orig = m.source$
                $e.data = m.data$
                $e.ssn = m.ssn$
                $e.rsn = \perp$
        else if ($m = (source, parent, ssn, rsn)$)
            if $\exists e \in ReplayLog_p$ such that $((e.orig = m.parent) \wedge (e.ssn = m.ssn))$
                $e.rsn \leftarrow m.rsn$
                $e.child \leftarrow m.source$
            else $ReplayLog_p \leftarrow ReplayLog_p \cup \{l\}$ where
                $e.orig = m.parent$
                $e.ssn = m.ssn$
                $e.rsn = m.rsn$
                $e.child = m.source$
        else if ($m = (source, grandparent, ssn, rsn)$)
            $ReceiveLog_p \leftarrow ReceiveLog_p \cup \{(m.source, m.grandparent, m.ssn, m.rsn)\}$
    else
        discard $m$
for all $e \in ReplayLog_p$ such that ($e.rsn \neq \perp$), in ascending $e.rsn$ order
    $RSN_p \leftarrow RSN_p + 1$
    $SsnTable_p[e.orig] \leftarrow e.ssn$
    $RsnLog_p \leftarrow RsnLog_p \cup \{(e.orig, e.ssn, e.rsn, e.child)\}$
    deliver $e.data$
for all remaining $e \in ReplayLog_p$, in ascending $e.ssn$ order
    $RSN_p \leftarrow RSN_p + 1$
    $SsnTable_p[e.orig] \leftarrow e.ssn$
    $RsnLog_p \leftarrow RsnLog_p \cup \{(e.orig, e.ssn, RSN_p, \perp)\}$
    deliver $e.data$
```

Figure 6: Recovery Protocol: Recovering Process

*Recovery of process p from the perspective of non-faulty process q*

```
$q$ receives ``p crashed/recovering''
send ``q begin replay'' to $p$
for all $e \in SendLog_q$ such that $e.dest = p$
    $m.source \leftarrow q$
    $m.data \leftarrow e.data$
    $m.ssn \leftarrow e.ssn$
    send $m$ to $p$
for all $e \in ReceiveLog_q$ such that $e.parent = p$
    $m.source \leftarrow q$
    $m.parent \leftarrow e.grandparent$
    $m.ssn \leftarrow e.ssn$
    $m.rsn \leftarrow e.rsn$
    send $m$ to $p$
for all $e \in RsnLog_q$ such that $((e.child = p) \vee (e.child = \perp))$
    $m.source \leftarrow q$
    $m.grandparent \leftarrow e.parent$
    $m.ssn \leftarrow e.ssn$
    $m.rsn \leftarrow e.rsn$
    send $m$ to $p$
send ``q exit replay'' to $p$
```

Figure 7: Recovery Protocol: Non-Faulty Process

the same data structures and fields as the definition of $\mathcal{L}_p$ does when recovering $p$ and resends the messages in an order consistent with the recovery procedure. Thus, family-based logging implements the recovery procedure. □

**Theorem 4** *Family-based logging satisfies the* **stable log** *property.*

*Proof.* Consider three processes $s$, $p$ that is a child of $s$, and $q$ that is a child of $p$. If $p$ crashes, then the values of $\mathcal{L}_s$ that are fully logged may no longer be fully logged, and the value of $\mathcal{L}_q$ becomes undefined.

From Theorem 2, if $p$ sent a message $m$ that was delivered by some process, then $p$ will send the same $m$ (that is, with the same data and send sequence number) when recovering, since otherwise the recovered global state would not be consistent. By doing so, $p$ will re-enter $m$ into $SendLog_p$ with the same values as were in the log before $p$'s crash. The data structure $ReceiveLog_p$ is rebuilt from the entries in $RsnLog_s$ for all parents $s$ of $p$ that have the *child* value set to either $p$ or $\perp$. Sending the latter entries—those with $(child = \perp)$—is necessary because $s$ may not know whether a message delivered by $s$ was fully logged or not, and so it ensures that *all* messages $s$ has delivered are fully logged.

Finally, $RsnLog_p$ is rebuilt from the information collected from the receive logs of $p$'s children (contributing the *child* and *rsn* fields for the messages that were fully logged) and the send logs of $p$'s parents (contributing the *parent* and *ssn* fields). □

## 4.4 Optimizations

In this section we discuss a number of techniques for reducing the cost of family-based logging. Our goal is to reduce the quantity of information piggybacked on each message, without adding too much complexity to the failure recovery protocol.

For example, consider the *grandparent* and *ssn* fields of the receive log. Together, these fields uniquely determine a single message recorded in the appropriate send log; but, since channels are FIFO, the *grandparent* and *rsn* fields can be used to uniquely determine a message. Suppose that process $p$ delivers message $m$ and then later fails. Note that if $m.rsn$ has been logged, then $m$ is relevant, and so every message delivered by $p$ before $m$ must also be relevant, hence fully logged. Thus, by using the *grandparent* and *rsn* fields from the receive logs of its children, the recovering $p$ can compute the order in which its parents sent it relevant messages. If $p$ also constructs the replayed messages from each of its parents in separate sequences, each ordered by send sequence number, then the ordered sequence of parents can be merged with the ordered sequences of messages so that all relevant messages are matched with their original receive sequence numbers. Once the *ssn* field is eliminated from the receive log, the RSN log no longer needs an *ssn* field, and so *ssn* values need not be piggybacked (except for the single $m.ssn$ associated with a message $m$ itself).

Not only can we eliminate send sequence numbers from piggybacks, we can also eliminate most of the receive sequence numbers. Consider a single message $m$ from $p$ to $q$; $m$ carries a sequence of entries from $RsnLog_p$. If we constrain $p$ to piggyback in receive sequence number order, then only the lowest *rsn* value attached to $m$ need be piggybacked, and $q$ can compute the other receive sequence numbers itself. Thus, $p$ need piggyback only a sequence of parents, together with a single receive sequence number.

### 4.4.1 Sender-Based Optimization

Suppose there is some process $p$ with only a single child, $q$. In this special case, we can easily optimize the usual message logging protocol. Because of FIFO channels, $p$ need never piggyback the same entry in $RsnLog_p$ to $q$ more than once. The message logging protocol for such a process $p$ can thus be modified as follows: Whenever $p$ sends a message $m$ to $q$ in state $\sigma_p[RSN_p]$, then $p$'s piggyback sequence number, $PSN_p$, can immediately be set to $RSN_p$ (without waiting for $q$ to acknowledge $m$).

This optimization can be generalized to an arbitrary process. For any process $p$, no entry in $RsnLog_p$ need be piggybacked to any single destination more than once. To implement this, $p$ must keep track of a different piggyback sequence number for each child of $p$. We store these in a new data structure: the *PSN table*. For any pair of processes $p$ and $q$, let $\mathcal{Q}_p$ be the set of entries of $RsnLog_p$ that have been piggybacked from $p$ to $q$. Then define $PsnTable_p[q] = \max\{e.rsn : e \in \mathcal{Q}_p\}$. If $PSN_p$ is maintained as before (based on acknowledgements), then whenever $p$ sends a message $m$ to $q$, $p$ need only piggyback all $e \in RsnLog_p$ such that $e.rsn > \max\{PSN_p, PsnTable_p[q]\}$. After $m$ has been sent, $p$ must update $PsnTable_p[q]$ to reflect the information piggybacked on $m$.

### 4.4.2 Receiver-Based Optimization

Using the above optimizations, we have transformed the piggyback data structure from a sequence of the form $\{(p_1, ssn_1, rsn_1), \ldots, (p_k, ssn_k, rsn_k)\}$ to a sequence of the form $\{rsn_1, p_1, p_2, \ldots, p_k\}$, such that no information is piggybacked to a single child more than once. We now consider one method of further compressing the piggybacked sequence of parent id's.

Suppose there is some process $p$ with only a single parent, $q$. In this case, we can again easily optimize the usual message logging protocol. Note that the send sequence numbers assigned by $q$ define a total ordering of the messages delivered by $p$. Thus, $p$ need not keep any RSN log at all. Should $p$ fail, $q$'s send log contains sufficient information to recover $p$. (Likewise, $p$ does not need $RSN_p$, $PSN_p$, and $PsnTable_p$; however, $p$ does need $SsnTable_p$ and $ReceiveLog_p$ in case $q$ fails.)

Process $p$ logs receive sequence numbers in order to record the nondeterministic characteristics of 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 send logs of $p$'s parents is sufficient to recover $p$ from failure.

Family-based logging can easily accommodate this general optimization. Each process $p$ can maintain an *interleave sequence number*, or $ISN_p$. $ISN_p$ is initially zero, and is incremented in state $\sigma_p[\ell]$ if $\ell = 1$ or if the source of the $\ell^{th}$ message is different from the source of the $(\ell - 1)^{th}$ message.

The RSN log can then be modified to record interleave sequence numbers. An entry $e \in RsnLog_p$ contains two new fields: Field $e.isn$ equals the value of $ISN_p$ in state $\sigma_p[e.rsn]$. Field $e.runlength$ equals the number of consecutive messages delivered by $p$ from $e.parent$ since state $\sigma_p[e.rsn]$. The message logging protocol can then be modified so that $p$ adds a new entry to its RSN log only when $p$ increments its interleave sequence number. Thus, $e.isn$ serves to count RSN log entries just as $e.rsn$ does in the unoptimized protocol. If $p$ delivers a message but does not increment its interleave sequence number, then $p$ has delivered a run of messages from the same parent, and $p$ must increment the *runlength* field in the last entry of its RSN log. Together with *e.rsn*, *e.runlength* encodes all the receive sequence numbers corresponding to entry $e \in RsnLog_p$.

Given these modifications, consider a message logging protocol in which $PSN_p$ and $PsnTable_p$ contain interleave sequence numbers rather than receive sequence numbers, but otherwise function as before. Whenever $p$ delivers consecutive messages from a single parent, $RsnLog_p$ will stop growing, and $p$ will piggyback nothing on its outgoing messages. Whenever $p$ interleaves messages from different parents, $p$ will record the interleaving in new entries of $RsnLog_p$, and then piggyback the information in these new entries on outgoing messages as before. (Note that the sequence of parent id's actually piggybacked by $p$ can also be compressed with run length encoding, but this optimization is independent of the changes to the message logging data structures described here.)

Finally, we must strengthen our failure recovery protocol in order to guarantee correctness. Suppose process $p$ fails. Note that $p$ may have delivered many consecutive relevant messages from one parent before crashing. Only the first receive sequence number of this sequence has necessarily been piggybacked and recorded in a child's receive log. However, if we require that $p$ deliver replayed messages such that runs of consecutive messages from the same parent are maximized (subject to the receive sequence number ordering imposed by $p$'s replay log) then failure recovery will return the system to a consistent state.

## 5 Performance

We have described in Section 4 an optimal message logging protocol, in the sense that it requires no extra messages and no extra processes in a failure-free run. However, this measure of optimality ignores the most interesting cost of family-based logging: the extra information that must be piggybacked on application messages. In this section we discuss theoretical bounds and empirical measurements of this cost. We also briefly discuss failure recovery performance. Our discussion concerns family-based logging as described in Section 4.3 with the additional optimization provided by the PSN table of Section 4.4.1.

### 5.1 Predicted Performance

For a message $m$ from $p$ to $q$, we will measure the quantity of piggybacked information by the number of RSN log entries contained in $m$. Unfortunately, for arbitrary $m$, we can only bound this number by the total number of messages delivered by $p$. However, let $\mu$ denote the *average* piggyback size on all messages sent during a run $\mathcal{R}$ of a set of processes $P$, $|P| = n$ in which no process crashes. That is, $\mu$ equals the total number of piggybacked RSN log entries divided by the total number of send events. We can bound $\mu$ as follows:

If $\mathcal{R}$ contains $s$ send events, $r$ receive events, $d$ delivery events, and $f$ transient channel failures, then $f = s - r$. Note that the total number of RSN log entries recorded by all $p \in P$ is at most $d$. Using the PSN table optimization, each entry can be piggybacked no more than $n-1+f$ times; thus, the total number of piggybacked RSN log entries is at most $d \cdot (n-1+f)$. Dividing by the total number of send events, we obtain $\mu \leq d \cdot (n-1+f)/(r+f)$. Since $d \leq r$ and $f \geq 0$, this bound simplifies to $\mu \leq n-1+f$.

This bound is achieved if each $p \in P$ runs the following application:

```
do forever
    for all $q \in P$ such that $q \neq p$
        send message to $q$
    for $i \leftarrow 1$ to $n-1$
        receive message
```

The second time the do-loop iterates, each process must piggyback $n-1$ RSN log entries on every outgoing message. Assuming no channel failures, the average piggyback size will quickly approach $n-1$ as the loop repeats.

This example illustrates the worst possible environment for family-based logging. We expect that many applications will not approach the $n-1+f$ worst case. The practical behavior of family-based logging depends largely on two factors. First, the frequency of acknowledgements has an obvious effect. Piggyback size will decrease if acknowledgements arrive promptly. Thus, a positive acknowledgement protocol is an ideal setting for family-based logging. Negative acknowledgement protocols may delay acknowledgements and increase piggyback size compared to positive acknowledgement schemes; however, the $n-1+f$ bound applies regardless of the underlying transport protocol.

Second, the application communication pattern strongly affects the performance of family-based logging. As our example shows, family-based logging suffers when each process has a large family of active parents and children. Note that in this case the set of parents and children will tend to intersect: each process will tend to send messages to and receive messages from some common set of processes. In such an environment, a negative acknowledgement protocol should be very effective, since extra acknowledgement packets will rarely be sent. Thus, in worst-case applications, family-based logging should actually perform better using a negative acknowledgement protocol.

## 5.2 Observed Performance

We have completed an initial implementation of the message logging and failure recovery protocols described in Section 4.3, with the addition of the PSN table described in Section 4.4.1. In addition, we have developed a special application to measure the performance of family-based logging under a wide variety of conditions. This section describes our current implementation and presents some empirical results.

### 5.2.1 Experimental Setup

We implemented family-based logging and have used it on a set of four Sun workstations—two SPARC2s and two IPXs—running Sun OS 4.1.1. Communications were implemented using UDP datagram sockets, layered over IP on a 10 Mbit/sec Ethernet; we used Sun LWP to manage program concurrency. Because the family-based logging protocol needs to receive the acknowledgements from the underlying data link layer protocol, we implemented a simple data link layer using a positive acknowledgement sliding window protocol; the data link layer can send 1024-byte messages between any pair of processes at the sustained rate of 2.5 milliseconds per message.

### 5.2.2 Application

We designed a unique application in order to test our implementation of family-based logging. The application is controlled by a master process. Based on user input, the master determines the number of additional processes to create and the characteristics of the communication among these processes. The master then writes a script for each process and sends each script to the appropriate process. Once each process has received its script, it executes the instructions contained in the script (e.g., "send to $p$", "receive") while monitoring message logging statistics.

There are two important properties of this application. First, it reduces application computational overhead to near zero. Writing, sending, and receiving scripts is completed before any performance data is collected. Each application event requires only a single array access and a test of the resulting value; the rest of the measured time is devoted to interprocess communication. Thus, we measure the message logging overhead as compared to virtually pure communication cost.

Second, our single application can model a wide range of application communication patterns in a controlled and repeatable manner. In addition to specifying the number of processes and the number and size of application messages, the user can control two parameters: the *blast factor* and the *branch factor*. The blast factor determines the relative frequency of sends and receives for each process. A process in a "blasty" run typically sends many messages before receiving any, and then receives many before sending again; in a non-blasty run it usually alternates between a single send and a single receive. The branch factor determines the relative number of parents and children for each process. A process in a "branchy" run typically sends messages to and receives messages from many different processes; in a non-branchy run it communicates with a small subset of processes.

### 5.2.3 Results

We measured message logging overhead for two distinct application scenarios. The worst-case example of Section 5.1 can be tested with a script that is maximally blasty and branchy. We call this scenario `blast`. To test family-based logging in a different setting, we set the blast factor to its minimum value, and kept the branch factor at its maximum value; a process in such a run typically alternates between sending one message and receiving one message, and communicates with all other processes equally often. We call this scenario `spray`.

In Figure 8 we summarize our results. Both scenarios send and receive 5000 1024-byte messages system-wide (approximately 1250 at each process). Using the Sun system clock, we measured the total run-time for three different logging settings: first with no message logging, then with piggybacking only (that is, the send log is not updated), and finally with full message logging. The piggybacking figures do not truly isolate the cost of piggybacking, but do suggest the relative cost of piggybacking for different scenarios. For example, `spray` spends more time piggybacking less data than `blast`: `spray` almost always has to recompute a new piggyback for each send, while `blast` can re-use the same piggyback for each sequence of sends. Our message logging overhead of 25 percent is comparable to the overhead of optimized pessimistic sender-based logging [5].[8] We also tested failure recovery by crashing a single process half-way through each scenario, and at the end of each scenario. Recovery in `blast` is significantly slower than in `spray`: `blast` causes complete redundancy in receive logs, giving a recovering process extra work to do, whereas `spray` tends to piggyback each RSN log entry to only one or two children.

[8]This implementation of sender-based logging does not block and sends no extra messages, and so avoids the theoretical cost of the protocol entirely.

| Application Scenario | No Log | P'back Only | Full Log | Half-run Recovery | Full-run Recovery | P'back Size | P'back Ovhd | Log Ovhd |
|---|---|---|---|---|---|---|---|---|
| **spray** | 11.8 | 13.9 | 14.7 | 7.5 | 15.4 | 1.6 | 18% | 25% |
| **blast** | 11.0 | 12.4 | 13.7 | 8.4 | 16.2 | 3.0 | 13% | 25% |

Figure 8: Family-Based Logging Performance (time in seconds)

## 6 Conclusions and Further Directions

In this paper, we developed a message logging protocol that introduces no additional blocking to the application and does not create orphans. Furthermore, the protocol is very efficient in that it only sends the application messages (possibly resent due to link failures) and their acknowledgements. Thus, our protocol does not use any more messages in a failure-free run than a message delivery protocol for a system in which transient link failures can occur but processes do not crash. The protocol may make application messages arbitrarily larger, but from our observations the average amount of overhead is small.

The major limitation of this protocol is that it can only withstand a sequence of (process crash; process recovery) pairs. For example, if process $p$ sends messages to process $q$ and both $p$ and $q$ simultaneously crash, then orphans may be created and $q$ may find itself trying to reconstruct a message $m \in \mathcal{L}_q$ for which there exists only a receive sequence number. However, we are designing a protocol that can tolerate $f \geq 1$ simultaneous crashes, implements the **stable log** property, and has the same efficiency as the protocol presented here. We are trying to prove this protocol correct by extending the abstract message logging protocol described in Section 4.2.

We are also examining how the message logging protocol can be further optimized by using the semantics of the application. For example, this research was first motivated by discussions with a group at IBM FSC in the AAS project [4]. In this system, processes are assumed to be usually functional and can recover by simply receiving new messages. This idea can be generalized to the existence of messages that a process $p$ can receive in any order with respect to messages from other processes without changing the sequence of messages that $p$ subsequently sends. If such optimizations are taken into account, then the amount of logged information can be further diminished.

**Acknowledgements** This work originated through discussions with Fred B. Schneider and with Jon Dehn and Burnie Witt of IBM FSC. We would like to thank Robbert Van Rennesse for his help in understanding the intricacies of Sun LWP and socket-level communication.

## References

[1] 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.

[2] 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.

[3] 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.

[4] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Digest of Papers: 20th IEEE International Conference on Fault-Tolerant Computing.* IEEE Computer Society, June 1990.

[5] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989. Available as report COMP TR89-101.

[6] D. B. Johnson. Personal communication, November 1992.

[7] 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.

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

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

[10] 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.

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

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

[13] 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.

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