Sender-Based Message Logging

David B. Johnson
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas

Abstract

Sender-based message logging is a new low-overhead mechanism for providing transparent fault-tolerance in distributed systems. It differs from conventional message logging mechanisms in that each message is logged in *volatile* memory on the machine from which the message is *sent*. Keeping the message log in the sender's local memory allows us to recover from a single failure at a time without the expense of synchronously logging each message to stable storage. The message log is then asynchronously written to stable storage, without delaying the computation, as part of the sender's periodic checkpoint. Maintaining the sender-based message log requires at most one extra network packet over non-fault-tolerant reliable message communication and imposes little additional synchronization delay. It can be applied transparently to existing distributed applications and does not require specialized hardware. It is currently being implemented on a network of SUN workstations.

1 Introduction

Sender-based message logging is a new low-overhead mechanism for providing fault tolerance in distributed systems. It can be applied transparently to existing applications and does not require the use of specialized hardware. It supports the recovery of processes executing in a distributed system from a single concurrent failure in the system at any time (i.e., no process can fail while another process has failed or is recovering). We are using sender-based message logging to add fault tolerance to compute-intensive applications executing in parallel on a collection of diskless workstations connected by a local area network.

In a network of personal workstations, individual machines often become unavailable from hardware failure or from the workstation owner reclaiming his machine. It is this type of failure from which we wish to recover. We do not currently support recovery from more complicated failure modes such as multiple concurrent failures or network partitioning, but instead concentrate on this situation of a single failure at a time. Also, we do not address in this paper the issues of maintaining the consistency and availability of static data such as file systems and databases [5] or the constraints of real-time applications [6, 7].

Sender-based message logging differs from other types of message logging mechanisms [2, 9, 13] in that the messages are logged in the local *volatile* memory on the machine from which each is *sent*, as illustrated in Figure 1. Keeping the message log in the sender's local memory allows us to recover from a single failure at a time without the expense of synchronously logging each message to a special logging or backup process or to stable storage, and without having to roll back any processes other than the failed one to achieve a consistent state following recovery. The message log is then asynchronously written to stable storage as part of the sender's periodic checkpoint. This allows the stable storage logging to proceed independently of computation, much the same as in Strom and Yemini's optimistic recovery protocol [13]. The sender-based message logging protocols accomplish this volatile logging with a minimum of overhead. They require at most one extra message over non-fault-tolerant reliable message communication and impose little additional synchronization delay. This technique also distributes message logging overhead proportionally over all processes sending messages and avoids the single point of failure possible with a centralized logging facility.

This paper describes the design and operation of the sender-based message logging mechanism. In Section 2, the model of a distributed system assumed by sender-based message logging is described. An overview of the design and the motivation behind it is presented in Section 3,

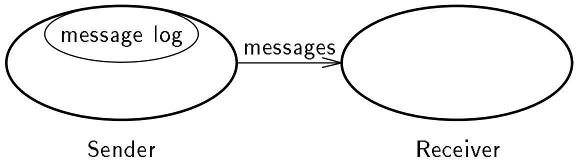

FIGURE 1: Process and message log configuration

This research was supported in part by the National Science Foundation under grant DCR-8511436 and by an IBM Faculty Development Award.

and Section 4 describes the data structures necessary for its realization. Section 5 provides a detailed description of the message logging and failure recovery protocols used in sender-based message logging, and an informal argument of their correctness. This section also discusses an "optimistic" version of the logging protocol that is currently under development. Related work is covered in Section 6, and conclusions and avenues for further work are presented in Section 7.

2 Distributed System Model

Sender-based message logging is designed to work with existing distributed systems without the addition of specialized hardware to the system or specialized code to applications. We make the following assumptions about the hardware and the applications:

- The system is composed of a network of fail-stop processors [12].
- Packet delivery on the network is not guaranteed, but reliable delivery can be implemented by retransmitting the packet a limited number of times and waiting for an acknowledgement from the destination. If no acknowledgement is received, the destination host is assumed to have failed.
- The network supports broadcast communication. All processors can be reached by a broadcast through a limited number of retransmissions of the packet.
- A network-wide stable storage service is always accessible to all processors in the system.
- Processes communicate with each other only through messages.
- All processes in the system are *deterministic* in the sense that, if two processes start in the same state, and both receive the identical sequence of inputs, they will produce the identical sequence outputs and will finish in the same state. The state of a process is thus completely determined by its starting state and by the sequence of messages it has received.

3 Design and Motivation

In sender-based message logging, each message transmitted is stored in the volatile memory of the machine from which it was sent. Additionally, each process is occasionally checkpointed to stable storage, but there is no coordination between the checkpoints of individual processes. When a process receives a message, it returns to the sender a *receive sequence number*, or *RSN*, which is then added to the log with the message. The return of the RSN may be merged with any acknowledgement required by the existing network protocol. This RSN indicates the order in which that message was *received* relative to other messages sent to the same process from other senders. This ordering information, which is not normally available to the sender, is required for successful recovery since the messages must be replayed from the log *in the same order* as they were received before the failure. Recovery of a failed process is done by restarting the failed process from its checkpoint and replaying the messages from the logs at the senders in ascending order by RSN.

Figure 2 shows an example of a distributed log resulting from this protocol. In this example, process Y initially had an RSN value of 6. Y first received two messages from process X_1, then two messages from process X_2, and finally another message from X_1. For each message received, Y incremented its current RSN and returned this new value to the sender. As the correct sender got the RSN from Y, it added it to its local log along with the message. After receiving these five messages, the current value of the RSN for Y is 11.

This design is motivated by the desire to minimize the overhead on the system imposed by the provision of fault tolerance. In general, there are three components to this overhead: message logging, checkpointing, and failure recovery. We concentrate here on minimizing the overhead of message logging. Since each message in the system must be logged, this overhead places a continuous burden on the system even when no faults occur. The checkpointing frequency can be tuned to balance its expense against the time needed for recovery or the space needed to store the log of messages received since the last checkpoint. Also, the overhead of failure recovery should be less important than that of message logging if failures are infrequent.

The method used for logging messages here is derived from a simple analysis of the minimum-cost method of doing the required logging. When one process sends a message to another, both the sender and the receiver naturally get (or already have) a copy of the message. Rather than synchronously sending a copy of it elsewhere for logging, it is faster to simply save a copy in local memory on either the sending or the receiving machine. Since the purpose of the logging is to recover the receiver if it fails, the receiver can not do this; however, the sending machine can easily save a copy of each message sent. Keeping the message log in the sender's local memory also distributes the logging overhead proportionally over all processes that send messages and avoids the possible single point of fail-

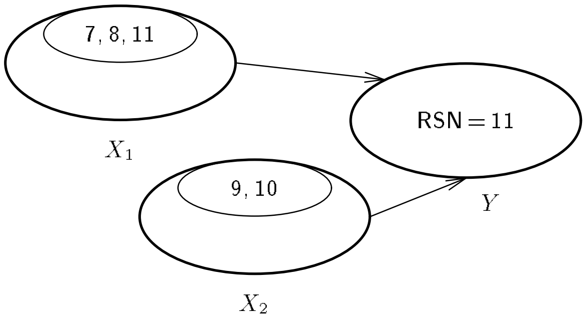

FIGURE 2: An example message log for sender-based message logging

ure of a centralized log. It is this idea that forms the basis of the sender-based message logging mechanism.

4 Data Structures

The inclusion of sender-based message logging in a distributed system requires the maintenance of the following items of system data for each participating process:

- A *send sequence number* or *SSN*: a sequence number of messages sent by the process. This is used for duplicate message suppression during recovery. Distributed systems that do not provide fault tolerance typically already require such a sequence number for suppression of duplicate messages. When this sequence number is included in the checkpoint of a process, it can be used to suppress duplicates caused by process recovery as well.
- A *receive sequence number* or *RSN*: a sequence number of messages *received* by the process. The RSN is incremented each time a new message is received, and the value *after* being incremented is assigned as the RSN for this message and is returned to the sender.
- A *message log* of messages *sent* by the process. This must contain the entire message that was sent including the data, the identification of the destination process, and the SSN used for that message. When the RSN for a message is returned by the receiver, it is also added to the log. After a process is checkpointed, all messages sent to that process and received before the checkpoint can be removed from the logs in the sending processes.
- A table recording the highest SSN value received in a message sent by each process with which this process has communicated. This is used for duplicate message detection.
- A table maintaining the RSN value that was returned for each message received since the last checkpoint of this process. This table is indexed by the SSN of the message and may be purged when the process is checkpointed.

Each of these data items except the last must be included when the process is checkpointed. When a process is restarted from its checkpoint, their values will be restored along with the rest of the checkpointed data.

5 The Protocols

The act of logging a message under sender-based message logging is not atomic, since the message data is entered into the log when it is sent but the RSN can only be entered after it has been received by the target process. It is thus possible for the receiver to fail while some messages do not yet have their RSNs recorded at the sender; such messages are called *partially logged* messages. The sender-based message logging protocols are designed so that any partially logged messages that exist for a failed process can be sent to it in any order after the sequence of *fully* logged messages have been sent to it in ascending RSN order.

5.1 Message Logging Protocol

With the sender-based message logging protocol, the following steps are required to send a message M from process X to process Y:

1. Process X sends the message M to process Y and inserts the message in its local volatile message log.
2. Process Y returns an acknowledgement to X and includes with this acknowledgement the RSN value it assigned to M.
3. Process X adds the RSN for this message to its log and sends an acknowledgement for the RSN back to Y.

The operation of this protocol in the absence of transmission errors is illustrated in Figure 3.

If either the message acknowledgement and RSN packet or the RSN acknowledgement packet is not received within some time, the preceding packet is retransmitted. If no response is received after some maximum number of such retransmissions, the destination machine is assumed to have failed. After returning the RSN, the receiver can continue execution without waiting for the RSN acknowledgement, but it must not send any messages (including input or output with the "outside world") until the RSNs of all messages that it has received have been acknowledged. The sender may continue normal execution immediately after the message is sent, but it must continue to retransmit the original message until the RSN packet arrives.

In comparison to the typical protocols used for reliable message delivery without fault tolerance, this protocol requires one extra network packet, used to acknowledge the RSN. The sender does not experience any extra delay, but does incur the overhead of copying the message and the RSN to the log. The receiver may or may not experience some delay depending on whether it needs to send messages immediately after receipt of the original message.

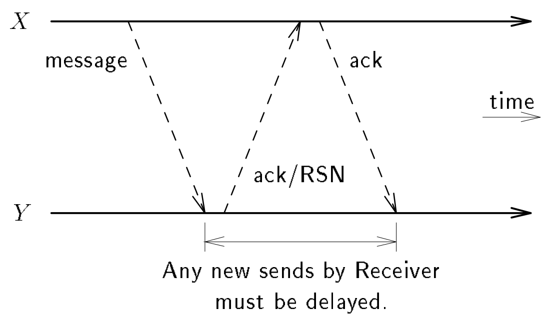

FIGURE 3: Operation of the message logging protocol in the absence of transmission errors

5.2 Failure Recovery Protocol

To recover a failed process, it is first restarted on some available processor from its most recent checkpoint. All fully logged messages for this process are then resent to it in ascending order of their logged RSNs. Only messages for which both the message data and the RSN have been recorded in the log are resent at this time; any partially logged messages are then sent to the process in any order after this. There is a separate message log stored at each process that sent messages to the failed process since its last checkpoint. The recovering process broadcasts requests for its logged messages, which are then replayed to it in ascending RSN order, beginning with the next message following the current RSN recorded in the checkpoint.

As the recovering process executes from its checkpointed state, it resends any messages that it sent after the checkpoint before the failure. Since the next SSN to use in sending messages is included in the process checkpoint, the SSNs used during recovery are the same as those used when these messages were originally sent before the failure. When receiving a message, If its SSN is less than or equal to the the highest SSN already received from this sender, the message is rejected as a duplicate. If the receiver has not checkpointed since originally receiving this message, it returns an acknowledgement including the RSN that it assigned when it first received this message. This RSN value is retrieved from its table recording the correspondence between the SSN of each message received and the RSN value assigned to that message. However, if the receiving process has been checkpointed since this message was first received, this table entry will have been purged, and an indication that this message need not be logged at the sender is returned instead.

5.3 Correctness

We show that in the case of a single failure at a time, this mechanism will correctly restore the state of the failed process to be consistent with the states of the other processes in the system.

First, during recovery, the process restored from its checkpointed state, and the sequence of fully logged messages are replayed to it in the same order as they were received before the failure (in ascending RSN order), beginning following the checkpointed RSN value. By the assumption of a single concurrent failure, these messages are all available in the volatile logs, and thus, by the assumption of determinism, the process reaches the same state as it had after receipt of these messages before the failure.

Next, the partially logged messages are replayed to the process in any order. Since processes are restricted from sending new messages until all messages they have received are fully logged, no processes other than the receiver can be affected by the receipt of a message that is still only partially logged. Thus, any change in the order of receipt of the partially logged messages during recovery can also only affect the state of the receiver and can not alter its consistency with other processes in the system.

While a process is recovering, it will resend the same messages that it sent after the checkpoint before the failure. Since the next SSN to use in sending messages is part of the checkpoint, these duplicates will be correctly detected and rejected by their receivers.

The data structures necessary for further participation in the protocol (Section 4) are correctly restored since they are recovered from the checkpoint and then modified as a result of receiving the same sequence of messages. In particular, the volatile message log in the failed process is recreated such that it may be used in the recovery of some other process after the current recovery is completed. Normally, the original RSN is returned in response to the duplicate message and is added to the log. However, if the receiver has checkpointed since this message was originally received, this message can not be needed for any future recovery of the receiver. In this case, an indication that the message is not needed is returned instead, the partially logged message is removed from the volatile log, and no RSN is recorded. In either case, correct further operation of the protocol is assured.

Finally, this mechanism avoids the problem of the domino effect [10, 11] since no processes other than the failed one need to be rolled back to recover from a failure.

5.4 An Optimistic Alternative

This protocol is an alternative to the basic message logging protocol of Section 5.1 that allows the receiver to send new messages to other processes without waiting for all messages it has received to be fully logged at their senders. This is an *optimistic* protocol in that it makes the optimistic assumption that the logging will eventually be completed (through retransmissions if necessary) before a failure occurs and maintains enough extra information to be able to roll back the system and to recover a consistent state if the assumption turns out to be wrong. Although this protocol is still under development, this section presents an initial overview of its design.

Using this optimistic protocol, it is now possible for a process to enter a state that is not consistent with the system state that may be created from recovery after a failure. For example, the scenario shown in Figure 4 is now possible. Here, process X has received a message M and then sent a message N to process Y. Process X then failed before the RSN for message M had been added to the log at its sender. During recovery, we cannot guarantee that message M is resent in the same order as it was received before the failure. Thus, process X potentially can not recreate the state from which message N was sent, and process Y may then exist in a state that is not consistent with the state recreated for process X after its recovery.

We introduce the following terminology to describe this situation. The state of a process is *unrecoverable* until all messages it has received are fully logged at their senders. If a process fails in an unrecoverable state, its state is *lost*; otherwise, its state may again become recoverable if all messages it has received are eventually fully logged by the return of their RSNs. When one process receives a message

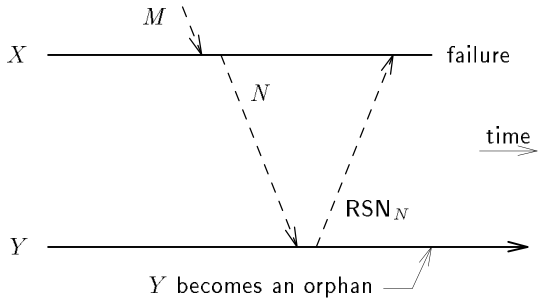

FIGURE 4: A possible scenario when using the optimistic logging protocol

from another, the state of the receiver *depends on* the state of the sender at the time the message was sent because any part of the sender's state might have been included in the message. If a process depends on a state that becomes lost, the process becomes an *orphan process* and the state of the process is then an *orphan state*.

In short, an orphan process Y is a process that has received a message N from a failed process X that sent message N after receiving a message M that was not fully logged at the time of X's failure (Figure 4). If the current RSN of a process is included in all messages sent by the process, and if each process maintains a table of the highest RSN it has received from any process, process Y has become an orphan from the failure of process X, if the value for X in its RSN table is higher than the RSN to which X was able to recover from the sequence of fully logged messages. To determine whether its failure has caused other processes to become orphaned, X broadcasts the value of the RSN to which it was able to recover. Any process that has a higher RSN value for X recorded in its table of highest RSN values received concludes that it has become an orphan. In addition to being invoked after a process failure, this orphan-detection algorithm must also be used before a process is checkpointed, since if the process does become an orphan, a checkpoint from before the state was orphaned will be needed for recovery.

After recovering the state of a process, the states of any orphaned processes are recovered by forcing them to fail one at a time and recovering them from their checkpoints and message logs in the same manner as is used for normal failed processes. Some of the messages logged for an orphaned process may have been recorded in the memory of the failed process, but this log information will be reconstructed during the recovery of that process. After the failed process and all orphans are recovered, their states will be consistent as of the time that the last fully logged message was received before the failure.

This form of the logging protocol has a number of advantages in addition to the added concurrency of allowing the receiver to proceed asynchronously from the receipt of the RSN acknowledgement. For instance, the sender could delay sending the acknowledgement of the RSN packet for a substantial period of time and piggyback it on the next message it needs to send to the receiving process, with a timeout mechanism if no such message is present. This would reduce the number of network packets to the same number as for reliable delivery in a system without fault tolerance. Extending this further, if processes use a remote procedure call protocol to communicate, there often is no explicit acknowledgement packet since the return from the RPC call implicitly acknowledges the call [1]. In this case, the RSN can be piggybacked on the RPC return packet and the RSN acknowledgement can be piggybacked on the next call packet, again without any additional network packets for the provision of fault tolerance, even with this highly optimized protocol.

6 Related Work

A number of fault-tolerance systems require applications to be written according to specific computational models to simplify the provision of fault tolerance. For example, the ARGUS system [8] requires applications to be structured as a (possibly nested) set of atomic actions on abstract data types. Since sender-based message logging is a transparent mechanism, it does not impose such restrictions on the applications.

The Auros distributed operating system [2] and the PUBLISHING mechanism [9] both use message logging but require specialized hardware to assist with the logging. Auros uses special networking hardware that atomically sends each message also to backup processes for the sender and the receiver. PUBLISHING uses a centralized logging machine that must reliably receive every network packet. Since sender-based message logging requires no such specialized hardware, it can be used over a broader class of existing systems without loss of efficiency.

Strom and Yemini's optimistic recovery mechanism uses an optimistic asynchronous message logging protocol that does not delay the sender or the receiver for synchronization with stable storage logging [13]. *Causal dependency tracking* and process rollback are used to recreate a consistent system state after a failure. The presence of a volatile log in sender-based message logging allows us to recover from a single failure at a time without rollback, while still maintaining the asynchrony between computation and stable storage logging. Furthermore, their desire to recover from more than a single concurrent failure leads to protocols that are significantly more complicated than sender-based message logging.

7 Conclusion

The sender-based message logging mechanism offers a simple, low-overhead solution to providing fault tolerance in distributed systems. Keeping a volatile log allows us to recover from a single failure at a time without rollback, and avoids the expense of synchronously logging each message to stable storage. Organizing the volatile log by sender results in an efficient logging protocol, with minimal extra network communication and synchronization delay. This results in an efficient fault-tolerance protocol that works naturally within the constraints of a distributed system. No special knowledge of fault tolerance is required by programs or programmers to use sender-based message logging. Since it does not rely on any specialized hardware to achieve fault tolerance, sender-based message logging can be added easily to existing distributed systems, as well as being designed into new systems.

We are currently implementing a prototype of sender-based message logging under the V-System [4, 3] on a collection of SUN workstations connected by an Ethernet network. Although the V-System differs slightly from the distributed system model assumed in this work, we believe that this can be satisfactorily handled in the implementation. We are also continuing development of the optimistic logging protocol discussed in Section 5.4. Finally, we are considering the extension of sender-based message logging with causal dependency tracking similar to that used in Strom and Yemini's optimistic recovery protocol [13] to allow for recovery from multiple concurrent failures. The presence of the volatile log in the sender should greatly reduce the occurrence of orphaned processes, thus reducing the need to roll back processes other than those that have failed.

Acknowledgements

We would like to thank Ken Birman, David Cheriton, Elaine Hill, Ed Lazowska, and Rick Schlichting for their comments on earlier drafts of this paper. We would also like to thank the referees for their help in improving the clarity of the presentation.

References

[1] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

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

[3] David R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[4] David R. Cheriton and Willy Zwaenepoel. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 129–140, ACM, October 1983.

[5] J. N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, chapter 3. F., pages 393–481, Springer-Verlag, New York, 1979.

[6] H. Hecht. Fault-tolerant software for real-time applications. *ACM Computing Surveys*, 8(4):391–407, December 1976.

[7] H. Kopetz. Resilient real-time systems. In T. Anderson, editor, *Resilient Computing Systems*, chapter 5, pages 91–101, Collins, London, 1985.

[8] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[9] Michael L. Powell and David L. Presotto. PUBLISHING: a reliable broadcast communication mechanism. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pages 100–109, ACM, October 1983.

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

[11] David L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[12] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant distributed computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[13] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.