An Integrated Approach to Fault Tolerance

Elmoootazbellah N. Elnozahy
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas*

1 Introduction

Manetho is an experimental system whose goal is to explore the extent to which transparent fault tolerance can be added to long-running distributed applications [5, 6]. Transparent techniques are attractive because they can automatically add fault tolerance to existing applications that were written without consideration for reliability.

Previous techniques for providing transparent fault-tolerance relied on rollback-recovery [7, 13]. However, rollback-recovery is not appropriate for server processes where the lack of service during rollback is intolerable. Furthermore, rollback-recovery assumes that a process can be restarted on any available host. As a result, extended downtime cannot be tolerated for example in file servers, which have to run on the host where the disks reside. Manetho solves these problems with an integrated approach by using process replication for server processes and rollback-recovery for client processes. The main features of Manetho are:

- **An integrated approach to providing reliability by process replication and rollback-recovery.** Replication is used for adding reliability to server processes with high availability requirements, while less expensive rollback-recovery is used for adding reliability to client processes.

- **Transparency.** Both replication and rollback-recovery are transparent to the application programs. Furthermore, if a process fails, the system will use the same recovery protocol regardless of whether the process is replicated or checkpointed.

- **Consistency of the system's behavior despite failures.** If the computation transmits a message outside the system domain (to the outside world), no future failure will cause the system to be in a state that is inconsistent with transmitting this message. This feature of Manetho is similar to the durability property of atomic transactions.

- **Containment of the effects of failures.** The failure in one process does not affect the rest of the computation in the system. The failed process is recovered to a state that is consistent with the functioning process.

Combining replication and rollback-recovery reflects our view that there is no technique for fault-tolerance that is appropriate for all applications. By following an approach that offers different methods within the same system, we are able to accommodate a wider variety of application requirements. In this paper, we briefly overview the process replication aspects of the system.

2 Process Replication

Manetho replicates an application process by a *troupe* that consists of a number of identical processes that execute the same deterministic application program [4]. The system transparently translates every application message directed to a replicated process into an *application-multicast* directed to the corresponding troupe. The failure of all but one process in each troupe is tolerated. It is assumed that failures are of the fail-stop type [12], that the application program is deterministic, and that the communication subsystem delivers multicast messages after an arbitrary delay to all, some, or none of the troupe members.

Manetho uses a new negative-acknowledgment, ordered multicast protocol to ensure that the troupe members receive the application-multicasts in the same order. The protocol depends on a novel combination of antecedence graph maintenance [5], a form of sender-based message logging [7], and the fact that all troupe members execute the same deterministic application program.
this protocol, each troupe is structured according to the leader-cohort model [2, 3]. When the troupe receives an application-multicast, the leader defines the receipt order and transmits it in an unreliable sequence-multicast to the cohorts. The cohorts do not acknowledge receiving sequence-multicasts or application-multicasts. The protocol delivers the application message to the application program at the leader without waiting for the delivery order information to reach all cohorts. At each cohort, the application message is delivered as soon as its sequence-multicast is received. Thus, by assuming that both the application-multicast and its sequence-multicast will not be lost, the protocol eliminates the overhead of control messages that would be used to achieve agreement among all troupe members. This works well in an environment in which failures are not frequent. Nevertheless, should a failure occur, the protocol will use the antecedence graph information, piggybacked on every application-multicast and recorded by the receiving troupe, to retrieve the ordering information that might have been lost due to the failure [6].

Manetho's multicast protocol improves on existing negative-acknowledgment multicast protocols in that it does not delay the message delivery to the application program. Existing protocols piggyback the receipt order of an application-multicast on future multicasts [3, 8, 10]. This reduces the number of control messages required by the protocol because the special acknowledgment messages are eliminated. However, an application-multicast cannot be delivered until its receipt order reaches all receivers, which occurs after receiving several subsequent application-multicasts. In Manetho, the application-multicast is delivered without the need for explicit acknowledgment messages or for delay in delivery until all receivers agree on the order. The information propagated in the antecedence graph piggybacked on application messages is used to recover from failures. This has the advantage of requiring a small number of control messages, like existing negative-acknowledgment protocols, while it eliminates the delay in the message delivery.

3 Protocol Overview

3.1 Definitions

We have formally defined the antecedence graph and described the protocol in detail elsewhere [6]. Here, we use an example to explain the idea behind the multicast protocol.

The execution of an application process (replicated or checkpointed) consists of a sequence of piecewise deterministic state intervals [7], each started by the receipt of an application message. Figure 1 shows the execution and state intervals of three application processes, p, q, and r. In this example, processes q and r are replicated while process p uses checkpointing and rollback-recovery. For clarity, we do not show the individual members of troupes q and r. The notation \( \sigma_p^i \) denotes the \( i \)th state interval of \( p \), where \( i \) is referred to as the index of \( \sigma_p^i \).

The directed, acyclic antecedence graph (AG) of a state interval \( \sigma_p^i \), \( AG(\sigma_p^i) \), is defined recursively as follows:

- \( i = 0 \): The graph consists of a node that represents \( \sigma_p^0 \) with no incoming edges. The node contains \( p \)'s identifier and the state interval index \( i = 0 \).
- \( i \neq 0 \): Suppose \( \sigma_p^i \) is created by receiving a multicast \( m_q^k \) from \( q \) sent at state interval \( \sigma_q^j \). \( AG(\sigma_p^i) \) consists of the union of \( AG(\sigma_p^{i-1}) \), \( AG(\sigma_q^j) \), and a node representing \( \sigma_p^i \), with two incoming edges: one from \( \sigma_p^{i-1} \) and one from \( \sigma_q^j \). The node representing \( \sigma_p^i \) contains \( p \)'s identifier, the state index \( i \), and the multicast's identifier \( k \).

The graph does not contain a copy of the message itself. Figure 2 shows the graph of \( \sigma_p^i \), \( AG(\sigma_p^i) \), in the example of Figure 1.

3.2 Protocol Operation

Consider the transmission of message \( m_p^j \). The system logs a copy of the message data in a volatile log maintained at the sender \( p \). The system also appends the antecedence graph of the current state of the sender \( p \) on
the message. This graph consists of the node \( s_q^p \). Since the destination of this message is replicated, the system transmits \( m^p_q \) as a multicast to troupe \( q \).

When the leader of \( q \) receives \( m^p_q \), it sends to its cohorts a sequence-multicast that defines the receipt order of \( m^p_q \). The graph appended to \( m^p_q \) is merged with the local graph at the leader. The system then delivers \( m^p_q \) to the application program without waiting for the sequence-multicasts to reach the cohorts.

When a cohort of \( q \) receives \( m^p_q \), it expects the corresponding sequence-multicast to be sent from the troupe leader within a short period. When the sequence-multicast arrives, the cohort performs the required antecedence graph maintenance as in the leader case, and delivers the message to the application program. The cohort does not acknowledge the receipt of the sequence-multicast.

Now consider the transmission of \( m^p_q \). At the leader of \( q \), the system adds a copy of \( m^p_q \) in a local volatile log. The system then appends the graph of the current state \( s_q^p \) of troupe \( q \) to \( m^p_q \) and transmits it to \( p \) in a one-to-one message, since \( p \) is not replicated. Only the leader of troupe \( q \) transmits the message over the network, the cohorts only add a copy of the message in their local volatile logs.

When \( p \) receives \( m^p_q \), it merges the appended graph into its own resulting in the graph in Figure 2, and the message is delivered to the application program.

### 3.3 Failures

A cohort may miss either or both of an application-multicast and its corresponding sequence-multicast. To prevent a cohort from "falling behind" the leader by missing both of these multicasts for several consecutive messages, the leader expects each cohort to periodically send a one-to-one synchronization message that shows the maximum state interval index known by the cohort. The leader's reply to a synchronization message contains the unique identifier, the sender's identifier, and receipt order for each application-multicast that the cohort has missed, if any. These messages are retrieved from the volatile logs of their corresponding senders.

If the leader fails while some sequence-multicasts have been lost, the antecedence graph is used to define the lost order. In the example in Figure 1, if \( q \)'s leader fails while the sequence-multicasts that carried the receipt orders of \( m^p_q \) and \( m^p_q \) to \( q \)'s cohorts were lost, \( p \)'s antecedence graph contains these missing orders and the cohorts can retrieve the messages in the correct order from the volatile logs of \( p \) and \( r \). The recovery protocol ensures that these messages will always be available. If their senders also fail, the messages will be recreated during recovery. The recovery protocol is further complicated by the fact that messages can be arbitrarily delayed over the network and that failures can occur during recovery.

We have presented the recovery protocol and a complete proof of its correctness as well as the issues of garbage collection and integration after failures elsewhere [5, 6].

### 3.4 Current Status

The rollback-recovery part of the system has been implemented, and the performance measurements show that the cost of maintaining the graph and message logs is reasonable. For long-running applications that use checkpointing, the running time of the program increases by less than 3%. The implementation of the replication part of the system is in progress.

### 4 Summary

Manetho is an experimental system whose goal is to explore the extent to which transparent fault tolerance can be added to long-running distributed applications. Our work is targeted at existing application programs that were written without consideration for fault tolerance. By using a transparent technique, fault tolerance can be added to such applications without additional programmer effort. Our work thus differs from approaches that provide the programmer with a set of primitives that facilitate writing fault-tolerant application programs, such as group-oriented systems [1, 11], or systems that rely on atomic transactions [9].

Manetho combines rollback-recovery and replication in the same protocol. The less expensive rollback-recovery is used for client processes while process replication is used for server processes where high availability is required. Manetho's support for replication is centered around a new negative-acknowledgment ordered multicast protocol. While many negative-acknowledgment multicast protocols have been published recently, our desire to maintain transparency of the replication from the application program precluded the use of protocols that provide weak ordering and exploit the semantics of the applications to achieve efficient ordered delivery [1, 11].

Manetho's multicast avoids the latency in message delivery common in negative-acknowledgment multicast protocols. The leader of a troupe delivers the messages to the application program without waiting to ensure that the corresponding sequence-multicasts reach every cohort. Similarly, a cohort delivers the message to the application program as soon as the corresponding sequence-multicast is available, even if the latter does not reach the rest of the cohorts.
Manetho's multicast is specifically designed for process-replication. For this purpose, we argue that the combination of antecedence graph maintenance and message logging at the sender offers a better tradeoff in terms of the number of required overhead messages and the delay in message delivery than any existing protocol. Like negative-acknowledgment multicast protocols, Manetho reduces the overhead during failure-free operation. In the normal case, a cohort does not acknowledge receiving application-multicasts or sequence-multicasts. By assuming that multicasts are seldom lost, the overhead of the acknowledgments is eliminated. This matches well with modern networks where communication failures are infrequent.

References


