Replicated Distributed Processes in Manetho

Elmootazbellah N. Elnozahy
Willy Zwaenepoel

Department of Computer Science
Rice University
Houston, Texas*

Abstract

This paper presents the process-replication protocol of Manetho, a system whose goal is to provide efficient, application-transparent fault tolerance to long-running distributed computations. Manetho uses a new negative-acknowledgment multicast protocol to enforce the same receipt order of application messages among all replicas of a process. The protocol depends on a combination of antecedence graph maintenance, a form of sender-based message logging, and the fact that the receivers of each multicast execute the same deterministic program. This combination allows our protocol to avoid the delay in application message delivery that is common in existing negative-acknowledgment multicast protocols, without giving up the advantage of requiring only a small number of control messages.

1 Introduction

This paper presents the process-replication protocol of Manetho. The goal of the Manetho system is to provide efficient, *application-transparent* fault tolerance for long-running distributed applications [12]. The system uses a combination of process-replication and rollback-recovery: process-replication is used for server processes that are constrained by high availability requirements, and rollback-recovery is used for all other client processes. In this paper, we concentrate on the process-replication aspects of Manetho; the rollback-recovery protocol has been published elsewhere [12].

In Manetho, process-replication follows the leader-cohort model [4, 7]. Each application process is replicated by a *troupe* [8] that consists of a *leader* and $r-1$ *cohorts*, where each troupe member executes the same application program.[1] Manetho assumes that the application process is deterministic in the sense that its execution is completely defined by its initial state and the sequence of messages it receives. Manetho tolerates $r-1$ fail-stop [20] failures in each troupe,[2] but it does not currently tolerate network partition.

Every application message between two application processes is translated internally into an *application-multicast* between the troupes implementing the two processes. To maintain the consistency among the troupe members, it is sufficient that each of them receives the same application-multicasts in the same order. Manetho uses a new negative-acknowledgment, ordered-multicast protocol to implement inter-troupe multicasts. Manetho's multicast protocol depends on a combination of *antecedence graph* maintenance [12], a form of sender-based message logging [14, 15], and the fact that a leader and its cohorts execute the same deterministic program. The graph at one troupe records the *receipt* order of application-multicasts in other troupes on which the local state of the troupe depends. The message logs are used to retransmit application-multicasts to recover from communication and processor failures. This combination allows the protocol to avoid the delay in application message delivery that is common in existing negative-acknowledgment protocols, without giving up the advantage of requiring only a small number of control messages.

The paper is organized as follows. Section 2 motivates the need for a new multicast protocol. Section 3 states the assumptions about the distributed system and distributed computations. Section 4 defines the new multicast protocol. Sections 5 and 6 show how the system recovers from failures. Section 7 describes how the system reclaims the storage used by the antecedence graph and message logs.

*This work was supported in part by NFS Grants CDA-8619893 and CCR-9116343, and by an IBM Graduate Fellowship.

[1] We use the term troupe instead of group to stress that all replicas execute the same program.

[2] Throughout this paper, we assume that the degree of replication r is the same for each troupe to simplify the presentation, although the algorithms presented in this paper do not depend on this fact.

Section 8 compares our system with related work. Finally, Section 9 presents conclusions.

2 Why a New Multicast Protocol?

To enforce consistency among troupe members in the absence of *any* information about the application program, the system requires a multicast protocol that satisfies the *agreement* and *order* conditions [10, 22]. The former condition requires that each troupe member receive the same set of messages, while the latter requires that each troupe member receive the messages in the same order.

Existing multicast protocols that satisfy the agreement and order conditions trade latency in delivery of multicast messages to the application program against the number of control messages. In positive-acknowledgment protocols, such as the original implementation of ABCAST of ISIS [5], the receivers run an agreement protocol to determine the receipt order of each application-multicast. The multicast can be delivered as soon as its receipt order is agreed upon, at the expense of the overhead caused by the control messages that are used to reach agreement. For example, the two-phase agreement protocol of this implementation of ABCAST requires r point-to-point messages and one overhead multicast to determine the receipt order of an application-multicast sent to r receivers. In contrast, negative-acknowledgment protocols [7, 17] attempt to reduce the number of control messages by piggybacking the ordering information on application-multicasts. However, reducing the number of control messages or eliminating them altogether introduces latency in achieving agreement on the receipt order of an application-multicast, which in turn introduces latency in delivering the multicast to the application program. For example, the r-resilient protocol by Chang and Maxemchuck requires only one overhead message per application-multicast, but it cannot deliver a message to the application program until $r-1$ "token transfers" have occurred, each requiring one message [7].

Realizing that satisfying the agreement and order conditions is expensive, some researchers have introduced efficient multicast protocols that provide weaker ordering. An example is ISIS's CBCAST protocol which provides *causal* ordering [3]. However, CBCAST does not enforce identical receipt orders for two multicasts sent from two sources that are not causally related [21]. Another example is the Psync multicast protocol based on the *context* order [19]. Analogous to CBCAST, two multicasts that are not related by the context order may not have a unique receipt order. To enforce such a unique order, a deterministic filter function must be applied on top of the protocol, which delays the delivery of the application-multicast until several subsequent multicasts [19].

Thus, existing multicast protocols that satisfy the agreement and order conditions are expensive, and cheaper protocols based on weaker ordering do not guarantee the ordering required by process-replication in the absence of information about the application program.

3 Assumptions

Manetho assumes that a distributed computation consists of a number of application processes that communicate only through messages. The processes are deterministic and do not have real-time requirements.

Each application process is replicated by a troupe [8] of r fail-stop [20] process replicas. Each replica has a distinct ordinal position within the troupe and executes the same application program. Because each troupe member executes the same deterministic program, it follows that if all members receive the same set of messages in the same order, no execution of a replica will diverge from that of the other troupe members. In each troupe, a distinguished member is called the *leader*, while the remaining $r-1$ replicas are called *cohorts*. Manetho tolerates $r-1$ failures in each troupe, but it does not currently tolerate network partition.

Manetho assumes that each troupe has access to a local group membership protocol that maintains a list of the members in the troupe [5, 9, 18]. The group membership protocol detects the changes in the troupe membership (due to failures and recoveries) and reliably notifies its members of such changes.

The communication subsystem supports multicast addressing and unreliable multicast delivery. Every troupe subscribes to a multicast address and exclusively uses multicast for inter-troupe communication. The communication subsystem may deliver a multicast message to all, some, or none of the troupe members, and multicasts may be arbitrarily delayed. Each multicast message has a unique identifier.

The execution of a troupe consists of a sequence of piecewise deterministic state intervals [14], each started by the

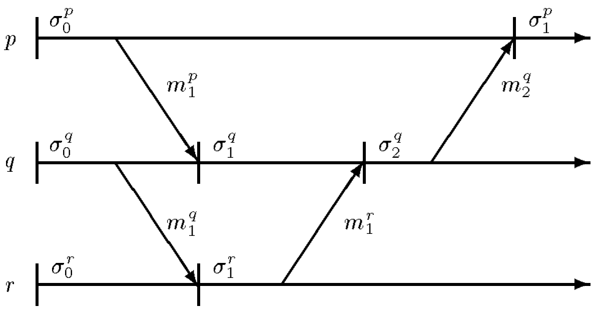

Figure 1 Example Execution

receipt of an application-multicast. Figure 1 shows the execution of three troupes and their state intervals. The horizontal lines represent the execution of the troupes, and arrows between troupes denote multicasts. For clarity, we do not show the individual members of each troupe. The notation σ_i^p denotes the i^{th} state interval of troupe p, where i is referred to as the *index* of σ_i^p. The notation m_i^p denotes the i^{th} application-multicast sent by troupe p. We will refer to this example throughout the paper.

4 Protocol Specification

4.1 The Antecedence Graph

The directed, acyclic *antecedence graph* (AG) of a state interval σ_i^p, $AG(\sigma_i^p)$, is defined recursively as follows [12]:

> $i = 0$: The graph consists of a node that represents σ_0^p with no incoming edges. The node contains the troupe identifier p and the state interval index $i = 0$.
>
> $i \neq 0$: Suppose σ_i^p is created by receiving a multicast m_k^q from troupe q sent at state interval σ_j^q. $AG(\sigma_i^p)$ consists of the union of $AG(\sigma_{i-1}^p)$, $AG(\sigma_j^q)$, and a node representing σ_i^p with two incoming edges: one from σ_{i-1}^p and one from σ_j^q. The node representing σ_i^p contains the troupe identifier p, the state index i, and the multicast identifier k.

The graph does *not* contain a copy of the multicast message itself. Figure 2 shows the graph $AG(\sigma_1^p)$ in the example of Figure 1.

4.2 Sending an Application-Multicast

Each troupe member maintains a *volatile* copy of the AG of its current state interval and a *volatile* log in which it stores a copy of the data of each multicast the troupe *sends*. When the application program sends a message from process p to process q, the leader of the troupe implementing p sends the message in an *application-multicast* addressed to troupe q. The cohorts of p do *not* send the message over the network; they only add the message to their volatile message logs. When a troupe leader sends a multicast, it (conceptually) piggybacks the AG of its current state interval on the message.

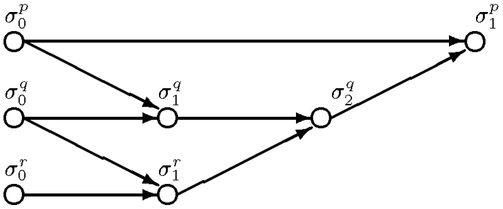

Figure 2 Antecedence Graph of state interval σ_1^p, $AG(\sigma_1^p)$.

4.3 Receiving an Application-Multicast

When a troupe receives an application-multicast, the leader defines the order in which it should be delivered to the application program and sends a *sequence-multicast* to its cohorts. The sequence-multicast contains the defined receipt order, the application-multicast's unique identifier, and the identifier of the sender troupe. The leader delivers the message to the application program without waiting for the sequence-multicast to reach the cohorts.

After a cohort receives an application-multicast, it expects the corresponding sequence-multicast from the leader within a short period. When the cohort receives the sequence-multicast, it delivers the message to the application program. The cohort does not acknowledge the sequence-multicast.

The leader does not acknowledge receiving the application-multicast to the sending troupe. Manetho only provides delivery of multicast messages subject to the agreement and order conditions. It does not, by itself, ensure reliable inter-troupe communication. Reliable FIFO inter-troupe channels can be easily provided on top of Manetho by an end-to-end protocol that uses sequence numbers and acknowledgments.

4.4 Antecedence Graph Maintenance

When a replica (leader or cohort) receives an application-multicast and its receipt order becomes available, a new state interval starts at that replica. The replica merges the AG piggybacked on the message with the AG of the previous state interval. The replica then creates a node representing the new state interval, with two incoming edges as described in Section 4.1.

4.5 Cohort Synchronization

Because communication failures are possible, a cohort may miss an application-multicast, its corresponding sequence-multicast, or both. 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 to the cohort. The leader's reply to a synchronization message contains the unique identifier, the sender troupe identifier, and receipt order for each application-multicast that the cohort has missed, if any.

4.6 Incremental Piggybacking of the Graph

The full AG need not be appended on every application-multicast. Instead, incremental piggybacking is used. The operation of the protocol specifies two techniques for pruning the graph appended to application-multicasts.

The first technique is applicable between any pair of troupes. As defined in Section 4.1, $AG(\sigma_i^p)$ is a proper subset of $AG(\sigma_{i+1}^p)$. Thus, if the leader of a troupe p detects that troupe q has received a prior application-multicast that was sent from state interval σ_i^p, then p need not append $AG(\sigma_i^p)$ on future application-multicasts sent to troupe q. Each troupe q that communicates with p includes with each message sent to p the maximum state interval index j such that the node representing σ_j^p is present in the AG of the current state interval of q. When p sends an application-multicast from σ_i^p, it includes only $AG(\sigma_i^p) - AG(\sigma_j^p)$.

The second technique relies on cohort synchronization. When the leader sends an application-multicast, the AG that corresponds to the state interval of the slowest troupe member need not be appended to the outgoing multicast. The leader determines the slowest troupe member as the one with the smallest state interval index as indicated in its last synchronization message. The information in the AG of that state interval is available to each troupe member, since for any p and i, $AG(\sigma_i^p)$ is a proper subset of $AG(\sigma_{i+1}^p)$. This graph will be available regardless of future failures, since Manetho assumes that no more than $r - 1$ failures can occur in each troupe.

The period between each synchronization by a particular cohort is an implementation concern. The implementor must weigh the overhead of processing the graph information and the probability of failures against the overhead of processing synchronization messages.

4.7 Handling Communication Failures

Manetho detects and recovers lost multicasts as follows:

- When a cohort receives a sequence-multicast for an application-multicast that it has not received, the sequence-multicast contains the identifiers of the application-multicast and the sender troupe. The cohort uses these identifiers to request a retransmission of the application-multicast from the sender troupe's message log.
- If a cohort receives a sequence-multicast that is out of order, it will detect that it has missed more than one application-multicast. In this case, the cohort synchronizes with the leader by sending a synchronization-message as described in Section 4.5.
- When a cohort receives an application-multicast, it expects to receive the corresponding sequence-multicast shortly thereafter. If the sequence-multicast is not received, the cohort requests it from the leader. The request contains the identifiers of the application-multicast and the sender troupe.
- The leader will determine that it has missed an application-multicast if it receives from one of its cohorts a request for a sequence-multicast corresponding to an application-multicast that the leader has not received. The leader requests the retransmission of the multicast from the corresponding sender's log.
- During cohort synchronization, a cohort determines the set of missed application or sequence-multicasts, if any. The leader's reply to the synchronization-message contains sufficient information for the cohort to request the missing application-multicasts from their senders and to deliver them to the application program in the correct order.

4.8 Advantages of the Protocol

Like other negative-acknowledgment multicast protocols, Manetho reduces the overhead during failure-free operation. In the normal case, a cohort does *not* acknowledge receiving application-multicasts and it acknowledges the sequence-multicasts only during synchronization. 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.

Manetho's multicast avoids the latency in message delivery common in negative-acknowledgment multicast protocols. The leader delivers the messages to the application program *without* waiting for the corresponding sequence-multicasts to 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.

5 Cohort Failure and Recovery

Detecting the failure of a cohort and integrating a new one into a troupe is done by the underlying group membership protocol. The ordinal position occupied by the failed cohort is not reused. The leader discards delayed messages from failed cohorts by checking if the sender's cohort identifier belongs to the current troupe membership. The new cohort starts normal processing after copying the state of the leader.

6 Leader Failure and Recovery

If the leader fails, the cohorts will need to determine whether the leader has accepted some application-multicasts that they have missed because of combined communication and leader failures. A troupe is considered to have failed when its leader has failed, and a recovery protocol must be run to bring the surviving cohorts to a state consistent with the leader's state before failure.

Recovery of a failed troupe takes place in two phases. First, the troupe elects a new leader. Second, the new leader runs a troupe recovery protocol. During this protocol, the elected leader represents the troupe in communicating with other troupes to retrieve the receipt order information that might have been lost due to the failure. This information is distributed across the AG's of the other troupes in the system. During both phases, the troupe does not accept application-multicasts from any troupe.

The recovery protocol is complicated by the possibility of concurrent failures and recoveries in other troupes and that application-multicasts sent from failed troupes are not bounded by a finite network delay.

6.1 Incarnation Numbers

Because application-multicasts are not bounded by a finite network delay, it is necessary to order the perception of a troupe failure with application-multicasts that were sent from that troupe. For this purpose, Manetho uses an *incarnation number* for each troupe. During troupe recovery, the troupe increments its incarnation number and does not resume normal processing before it reliably informs all other troupes of its new incarnation number (see Section 6.3). Each application-multicast is tagged with the current incarnation number of the sending troupe. Thus, all other troupes in the system are able to detect the multicasts that were sent before the failure of their senders and reject them.

6.2 Phase One: Leader Election

If one or more cohorts detect the leader failure, they will use the following protocol to elect a new leader. The protocol is an adaptation of the *invitation* protocol [13] in which the winner of the election is the cohort that has the highest state interval index.

- One cohort (or more) starts leader election by sending a *recovery-multicast* to the troupe. The multicast contains the cohort's current state interval index and ordinal position within the troupe.
- When a replica receives a recovery-multicast carrying a state interval index larger than its own, it sends back a *leadership-acknowledgment* message, and aborts its own leadership election, if it has started one. Otherwise, when a replica receives a recovery-multicast with a state interval smaller than its own, it starts its own leadership election, if it has not already done so. Ties are broken in favor of the cohort with the smaller ordinal position.
- The initiator collects the responses from every member of the troupe. The initiator retransmits the recovery-multicast until it receives a corresponding leadership-acknowledgment from every surviving member of the troupe, as determined by the underlying troupe membership protocol.
- The new leader increments the troupe incarnation number.
- The new leader forces each cohort to synchronize to bring all cohorts to the most recent state interval. The leader informs the cohorts of the new incarnation number during synchronization.

Provided that there is at least one surviving troupe member, the protocol elects a single leader and terminates [13]. If the initiator of the protocol fails, the protocol is restarted.

6.3 Phase Two: Troupe Recovery

The recovery protocol is based on the following observation [12]: Define a state interval σ_i^p as *visible* outside of troupe p if the AG of the current state interval of some other troupe q contains a node that represents σ_i^p. Then, $AG(\sigma_i^p)$ is a subgraph of the AG of the current and all subsequent state intervals of q. If the leader of troupe p fails, the newly elected leader negotiates with all other troupes to determine the AG's of its visible state intervals. By merging these AG's, the troupe can reconstruct the AG of the most recent visible state interval. The new leader uses this AG to determine the receipt order of application-multicasts whose corresponding sequence-multicasts were lost. Using the unique identifier of each application-multicast as indicated by the AG, the newly elected leader requests them from their corresponding senders. If the sender has also failed, its message log will be reconstructed during its recovery, and the message will become available, as will be shown in Section 6.4. The recovering troupe executes up to its most recent "visible" state interval from before failure. This brings the troupe to a state consistent [6] with the other troupes in the system.[3]

6.4 Protocol Description

Figure 3 shows the troupe recovery protocol. The newly elected leader starts recovery by calling the procedure $RECOVER$ with arguments p, S, $INCNUM$, AG and $STATEINDEX$. The recovering troupe's identifier is p. Set

[3] The rollback-recovery protocol of Manetho uses the same concepts presented here, although the replication aspects require special treatment in the recovery algorithm. This allows Manetho to conceptually use the same recovery protocol, despite whether the process is using replication or rollback-recovery.

S contains a list of the troupes that participate in the computation. *INCNUM* is the new incarnation number of the recovering troupe. AG is the graph of the current state interval of the troupe, and *STATEINDEX* is the index of that state interval. The new leader of troupe p performs a *GET_AG* remote procedure call *(RPC)* at the leader of every troupe. Messages exchanged for the purpose of recovery are considered out-of-band and do not carry AG information. Recovering troupes respond to *GET_AG* calls.

In *GET_AG* at each troupe q, the leader of troupe q determines m, the index of the most recent state interval σ^p_m of troupe p in q's AG. The leader then calls *SYNC_COHORT* at each of its cohorts. In *SYNC_COHORT*, each cohort copies the argument *LAG* into its local AG, and discards every application-multicast whose order has not been defined in *LAG*. The cohort then adds m to *REJECT_LIST*. Until it receives an *UNSYNC_COHORT* call from the leader, the cohort does not accept any application-multicast (from any sender) whose appended AG contains a state interval of troupe p whose index is greater than m. While waiting for the *SYNC_COHORT* calls to return, the leader does not process application-multicasts and postpones its response to any *GET_AG* call. When all *SYNC_COHORT* calls return, the leader of troupe q returns its current incarnation number and $AG(\sigma^p_m)$ to the leader of troupe p. The leader of troupe q adds m to *REJECTLIST*. Until q receives a *CONFIRM* call from the leader of p, q rejects any application-multicast (from any sender) whose appended AG contains a state interval of troupe p whose index is greater than m. The *SYNC_COHORT* call makes the cohorts "witness" the answer returned by q's leader. The *REJECTLIST* prevents troupe q from observing a state of troupe p that was not reflected in q's response to p's *GET_AG* call. The cohorts also do not retain any application-multicast, for which the corresponding *sequence-multicast* has not been received. If the current leader of q fails, the state of each cohort will show σ^p_m as the most recent state interval of troupe of p in the AG of troupe q.

```
procedure $RECOVER(p, S, AG, INCNUM, STATEINDEX)$
  $INCLIST[p] \leftarrow INCNUM$;
  for all $q \in S,\ q \neq p$, do in parallel
    $(QAG,\ QINC) \leftarrow$ RPC at leader of $q$ : $GET\_AG(p)$;
    $AG \leftarrow AG \cup QAG$;
    $INCLIST[q] \leftarrow QINC$;
  for all $q \in S,\ q \neq p$, do in parallel
    RPC at $q$ : $CONFIRM\ (p, INCLIST)$;
  RPC at every cohort $RECOVER\_COHORT\ (INCLIST)$;
  $v \leftarrow$ max $j$ such that $\sigma^p_j \in AG$;
  $SI \leftarrow STATEINDEX$;
  while $SI \leq v$ do
    execute up to next message receipt without
    sending application-multicasts;
    update message log;
    $SI \leftarrow SI + 1$;
    request multicast that started interval $SI$ from sender;
    receive and process application-multicast;
  return;

procedure $GET\_AG(p)$
  $m \leftarrow$ max $j$ such that $\sigma^p_j \in AG$;
  RPC at each cohort: $SYNC\_COHORT(p, m, AG)$;
  $REJECTLIST[p] \leftarrow m$;
  return $(AG(\sigma^p_m), INCNUM)$;

procedure $CONFIRM\ (p, ILIST)$
  for all $r \in S$, do
    $INCLIST[r] \leftarrow$ max$(ILIST[r], INCLIST[r])$;
  RPC at each cohort: $UNSYNC\_COHORT(p, INCLIST)$;
  $REJECTLIST[p] \leftarrow \infty$;
  return;

procedure $RECOVER\_COHORT\ (ILIST)$
  $INCLIST \leftarrow ILIST$;
  return;

procedure $SYNC\_COHORT\ (p, m, LAG)$
  $AG \leftarrow LAG$;
  discard application-multicasts with unspecified receipt order;
  $REJECTLIST[p] \leftarrow m$;
  return;

procedure $UNSYNC\_COHORT\ (p, ILIST)$
  $INCLIST \leftarrow ILIST$;
  $REJECTLIST[p] \leftarrow \infty$;
  return;
```

Figure 3 The Troupe Recovery Protocol.

When each *GET_AG* call returns to p, it merges the returned graph into AG and updates its list of incarnation numbers *INCLIST*. When all *GET_AG* calls have returned, p performs a *CONFIRM* remote procedure call at the leader of every troupe q. In *CONFIRM*, the leader of q updates its incarnation list and updates *REJECTLIST* to indicate that it no longer has any restriction on accepting messages that contain state intervals of p, provided they belong to its new incarnation. The leader of q then calls *UNSYNC_COHORT* at every cohort to update the cohort's *REJECTLIST* and *INCLIST*.

The leader of troupe p calls *RECOVER_COHORT* at each of its cohorts to update the cohort's *INCLIST*. The leader of troupe p determines v, the largest state interval index among the troupe's visible state intervals. It proceeds to re-run the pre-failure execution, requesting messages as indicated by the reconstructed AG from their senders, which retransmit the corresponding application-multicasts from their log to p. The leader uses the AG to define the receipt order of these multicasts and sends the corresponding sequence-multicasts to the troupe. The leader of p does not send application-multicasts while it is recovering, but it stores these messages in its volatile log.

Throughout recovery, the troupe restarts the recovery protocol if its leader fails. If a cohort fails, it is eliminated from the troupe as described in Section 5.

6.5 Correctness

Definition 1 *Two distributed computations are* equivalent *if and only if the final state of each process is the same in both computations.*

Consider the failure and recovery of some troupe p:

Definition 2 *Let G^p be the graph computed by p during $RECOVER$.*

Definition 3 *All state intervals σ_i^p, $i > v$, that occurred* before *failure are called* lost *state intervals.*

Definition 4 *A troupe q whose leader was not recovering when it responded to p's* GET_AG *call is called a* live *troupe.*

Let C be the computation as executed by the system including failures and recoveries. We show that there exists some legal computation C' in which no failures occur, and which starts in the same state as C, such that C and C' are equivalent.

We first show that the graph computed by $RECOVER$ is indeed $AG(\sigma_v^p)$.

Lemma 1 *$G^p = AG(\sigma_v^p)$.*

Proof There are two cases to consider:
Case 1: $v = STATEINDEX$. Running $RECOVER$ in this case did not add to the knowledge of the new leader about the execution of the failed leader, and $AG(\sigma_v^p)$ is available at each cohort after the end of the election protocol.
Case 2: $v > STATEINDEX$. Let troupe q be some troupe that returned $AG(\sigma_v^p)$ in its response to p's GET_AG call. If q has the complete subgraph representing $AG(\sigma_v^p)$ in q's own graph, then the lemma is true. Otherwise, $AG(\sigma_v^p)$ must be missing one or more subgraphs, since some other troupes have synchronized their cohorts before sending the application-multicasts that should have included these missing subgraphs. In this case, these troupes must have the missing subgraphs available despite any failure (up to $r - 1$ failures in each troupe). Therefore, p will receive the missing subgraphs of $AG(\sigma_v^p)$ during the GET_AG calls at these troupes. □

Lemma 2 *After all GET_AG calls return but before any $CONFIRM$ call is issued during p's recovery, no lost state interval σ_i^p appears in the AG of any troupe q.*

Proof Consider the point in $RECOVER$ at which p has received all the results of GET_AG calls but has not sent any $CONFIRM$ calls. No state interval σ_i^p that occurred after σ_v^p has a corresponding node in the AG of any troupe q, or else, q should have returned $AG(\sigma_i^p)$ during its reply to p's GET_AG call. After returning p's GET_AG call and before receiving the $CONFIRM$ call, the use of $REJECTLIST$ prevents every member of troupe q from accepting any application-multicast whose appended AG carries a node that corresponds to σ_i^p, where $i > v$. □

Because of the unbounded network delays, there may be some application-multicasts still in transit in the communication channels that carry a node that represents a lost state interval in the appended AG. We show that these multicasts will be rejected.

Lemma 3 *A message whose appended AG carries a node that corresponds to a lost state interval of p will be rejected by any troupe that receives it.*

Proof Assume that r sends to q an application-multicast m_k^r whose appended AG contains a node that represents a lost state interval σ_i^p. From Lemma 2, the multicast cannot originate from the current incarnation of r. Hence, the multicast originates from a previous incarnation of r. There are three cases:
Case 1: m_k^r arrives at troupe q before p's GET_AG call executes at q. In this case, the leader of q did not receive the message, while one or more cohorts did. No cohort will retain m_k^r after it synchronizes with the leader and discards the unordered messages during the $SYNC\_COHORT$ call.
Case 2: m_k^r arrives at troupe q after p's GET_AG call executes at q, but before p's $CONFIRM$ call executes at q. The multicast will be rejected because of the use of $REJECTLIST$ as in Lemma 2.
Case 3: m_k^r arrives at troupe q after p's $CONFIRM$ call executes at q. Because p broadcasts the current incarnation of every troupe in $CONFIRM$, q detects that the incarnation of r tagging m_k^r is old and rejects it. □

Lemmas 2 and 3 establish a safety property of the protocol: Lost state intervals cannot affect the computation.

We now show that despite an arbitrary number of failures in the troupe leaders, including additional failures during recovery, troupe p restores a state consistent with the rest of the computation.

Lemma 4 *$\forall i, q$ such that $\sigma_i^q \in G^p, AG(\sigma_i^q)$ will always be available at q.*

Proof If q was a live troupe when it returned p's GET_AG call, then the lemma is true despite of any subsequent failures in p or q, because all q's cohorts synchronize with their leader before returning p's call, making $AG(\sigma_i^q)$ available to all replicas of q. Subsequent failures of q will not affect the availability of $AG(\sigma_i^q)$.

Otherwise, troupe q was recovering when it returned p's GET_AG call. There are two cases:
Case 1: $AG(\sigma_i^q)$ is a subgraph of the AG of a state interval of some troupe r, and r was live when it returned p's GET_AG call. There are two cases:

Case i: r returned p's *GET_AG* call before q's *GET_AG* call executed at r. Thus, troupe r's synchronization made $AG(\sigma_i^q)$ available at each cohort of r despite of future failures in r. $AG(\sigma_i^q)$ will be returned to q because of q's call at r (despite of any subsequent failures of r or q).

Case ii: r returned p's *GET_AG* call after q's. $AG(\sigma_i^q)$ must have been returned to q's call, since r could not have added $AG(\sigma_i^q)$ to its own AG after q's call, because of the use of *REJECTLIST*. This also holds if r fails after q's call has returned but before p's call, because a recovering troupe does not accept application-multicasts until it finishes recovery.

Case 2: $AG(\sigma_i^q)$ is not a subgraph of the AG of the current state interval of any live troupe. Hence, p must have received $AG(\sigma_i^q)$ from some troupe s that was recovering and had $AG(\sigma_i^q)$ as a subgraph of the AG of the state interval of the new leader before it started troupe recovery. Hence, both p and q will receive $AG(\sigma_i^q)$ from s, despite of any subsequent failures of p, q or s. □

Lemma 5 *The troupe recovery protocol restores the execution up to state interval σ_v^p.*

Proof Construct graph F^p by removing from G^p the nodes that correspond to state intervals in live troupes or that occurred before the current state intervals of the new leaders in recovering troupes. Every state interval in F^p will be recreated. The proof proceeds by induction on the topological sort of F^p, which must exist because F^p is acyclic.

Base case: Each node at level 0 of the topological sort represents a state interval σ_i^p such that troupe p is recovering and the current state interval of p is σ_{i-1}^p. To recreate σ_i^p, p must receive some application-multicast m_k^q, such that either q is a live troupe or the application-multicast was sent from a previous state interval at some recovering troupe. In both cases, a copy of m_k^q must be available in the volatile message log of q. Thus, p can request a replay of m_k^q.

Induction hypothesis: Assume that the lemma is true for all nodes of topological level n.

Induction step: For each node at topological level $n+1$, the application-multicast that created the corresponding state interval is available either because it was recreated and added to its sender's log during recovery by the induction hypothesis, or was already available in the log of the sender as in the base case. □

Lemma 6 *The protocol is deadlock-free.*

Proof No deadlock can occur during the phase of collecting the AG, because recovering troupes return *GET_AG* calls. Cohort synchronization during *SYNC_COHORT* is internal to the troupe and does not block. Lemma 5 shows that no deadlock can occur while recreating the state intervals. □

Lemmas 4, 5 and 6 establish the liveness property of the protocol: Each troupe that fails will recover to its maximum visible state interval.

Lemma 7 *No troupe's state becomes inconsistent with the rest of the system because of p's failure.*

Proof Follows immediately from Lemma 4, Lemma 5, and the definition of σ_v^p. □

Lemma 7 establishes the remaining safety property of the protocol:

Theorem 1 *Computation C is equivalent to some legal computation C' that starts from the same initial state.*

Proof Before any failure occurs in C, the state of the system is consistent [6]. After the failure of a troupe p, it recovers to a state consistent with the rest of the system, and no other troupe becomes inconsistent with the rest of the system because of p's failure, as shown by Lemma 7. Furthermore, the effects of lost state intervals of previous incarnations cannot affect the computation, by Lemmas 2 and 3. Lemmas 4, 5, and 6 establish that the recovery of each troupe eventually completes. Therefore C, the execution of the system after all failures and recoveries have completed, is a possible execution of the system C' in which no failures have occurred. Since all processes are assumed to be deterministic, by executing C and C' from the same initial state and with the same sequence of exchanged multicasts, C and C' must both complete in the same final states. □

7 Garbage Collection

We state without proof the conditions for removing a message from the message log and for removing an edge from the AG.

Lemma 8 *If the slowest member of a troupe p has already received and delivered application-multicast m_i^q, then troupe q may remove the message from its log.*

Lemma 9 *If the state interval of the slowest member in a troupe p is σ_i^p, then all nodes that correspond to σ_j^p, where $j \leq i$, are no longer needed for recovery.*

Lemmas 8 and 9 form the basis for many possible garbage collection protocols. For example, two troupes can periodically exchange the information about the state interval and the identifiers of messages received by the slowest member of either troupe. Alternatively, this information can be periodically propagated with the AG appended on application-multicasts. The implementation must balance the frequency of exchanging garbage collection information against the resulting overhead and the available storage.

8 Comparison with Related Work

Unlike many other multicast protocols, Manetho's multicast is specifically designed for process-replication. For this purpose, the combination of antecedence graph maintenance and message logging at the sender offers a better tradeoff in terms of the number of overhead messages and the delay in message delivery than the protocols that have been published in the literature. We restrict the comparison to systems that operate in environments similar to the one assumed in this paper, namely, an unreliable asynchronous network and applications with no real-time requirements.

CIRCUS was one of the earlier systems to support process-replication in an asynchronous network [8]. CIRCUS uses replicated remote procedure calls (RPCs) to implement inter-troupe communication. If no identical receipt order at each replica is required, a many-to-many RPC incurs between $r+1$ to $2r$ multicasts. Identical receipt order is achieved by structuring the many-to-many RPC as a transaction that deadlocks if two members of the troupe receive messages in different orders. Committing this transaction requires at least r additional multicasts. In contrast, Manetho provides ordered multicast delivery with only one overhead multicast per application-multicast.

The protocol of Ahamad et al. [1] uses transactions to structure the replicas. At commit time, only one replica succeeds while the remaining cohorts abort. This allows non-deterministic execution in each replica, but the application must be structured as a sequence of transactions. In contrast, Manetho adds replication to deterministic processes in an application-transparent manner.

The idea of having a sequencer define the receipt order of a multicast was used in the multicast protocol of Chang and Maxemchuck [7], the Amoeba atomic broadcast protocol [16], and the Delta-4 XPA system [2]. The r-resilient protocol of Chang and Maxemchuck relies on negative-acknowledgment and leadership transfer to achieve reliable total ordering. However, a multicast must be delayed for $r-1$ leadership transfers before it can be delivered. Like Chang and Maxemchuck, our protocol incurs few overhead control messages, but it avoids the delay in delivering the multicast by using the information in the antecedence graph.

Amoeba's atomic broadcast protocol uses negative-acknowledgment for the 0-resilient version, and positive acknowledgments for the r-resilient version. The Amoeba protocol is highly tuned for the 0-resilient operation mode. The r-resilient version of Amoeba requires $r-1$ overhead messages for each application-multicast. Manetho does not require such overhead messages.

The Delta-4 XPA multicast protocol uses positive acknowledgments. Delta-4 XPA relies on a special network adapter to provide the ordering and reliability, and to mask the overhead of acknowledgment messages from the application program. In contrast, Manetho does not depend on special network support.

Both Manetho and the new implementation of ISIS's ABCAST [3] rely on a single site to define the multicast's receipt order. ABCAST relies on an underlying transport protocol that guarantees that messages are reliably delivered in FIFO order. This transport protocol is a major source of overhead in ISIS [3]. In contrast, Manetho adopts weaker assumption about the network reliability.

The *context graph* of the x-kernel's Psync protocol [19] is the basis of another general-purpose multicast protocol. Unlike our protocol, Psync does not guarantee the identical receipt ordering required by process-replication in the absence of information about the application's semantics. Such ordering can be provided in Psync by applying an ordering filter on the context graph, which delays the delivery of the application-multicast at each site for several application-multicasts [19].

The atomic broadcast protocol of Melliar-Smith et al. [17] uses no control messages during normal operation at the expense of a large delay in message delivery. This delay depends mainly on the rate of incoming application-broadcasts. Manetho pays the overhead of maintaining the graph and one overhead multicast, in return for reducing the latency in message delivery independently of the rate of incoming multicasts.

9 Conclusion

This paper has presented the process-replication protocol of Manetho, a fault-tolerant distributed system whose purpose is to provide application-transparent fault tolerance to long-running applications. The system uses a new ordered-multicast protocol which is designed specifically to support process-replication. The protocol relies on a combination of antecedence graph maintenance, volatile message logging at the sender, and the fact that the receivers of the multicast execute the same deterministic program. Unlike many general-purpose multicast protocols published in the literature, ours is able to use negative acknowledgments to reduce the number of overhead messages, and at the same time avoids the delays in message delivery typically incurred by negative acknowledgment protocols. These advantages come at the expense of maintaining the antecedence graph and the need for a more elaborate recovery protocol under some rare failure scenarios. Nevertheless, an implementation of the antecedence graph shows that, by using incremental piggybacking, the cost of maintaining the graph is only a small fraction of the cost of receiving a message [11]. Furthermore, assuming that failures are rare, the recovery protocol will seldom have to be run.

Acknowledgments

We are indebted to J. Carter, A. Cox, K. Fletcher, P. Keleher, M. Mazina, H. Garcia-Molina, A. Schaffer, R. Schlichting, H. Youssef and the anonymous referees for many useful comments about earlier drafts of this manuscript. The insightful comments of David Johnson helped improve the clarity of the proofs and the presentation.

References

[1] M. Ahamad, P. Dasgupta, and R.J. LeBlanc. Fault-tolerant atomic computations in an object-based distributed system. *Distributed Computing*, 4:69–80, 1990.

[2] P.A. Barrett, A.M. Hilborne, P. Verissimo, L. Rodrigues, P.G. Bond, D.T. Seaton, and N.A. Speirs. The Delta-4 extra performance architecture XPA. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 481–488, June 1990.

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

[4] K.P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.

[5] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[6] 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.

[7] J. Chang and N.F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[8] E.C. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, December 1985.

[9] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, pages 206–211, June 1988.

[10] F. Cristian, R. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing*, June 1985.

[11] E.N. Elnozahy and W. Zwaenepoel. Manetho: A low overhead rollback-recovery system with fast output commit. Technical Report TR91-152, Rice University, March 1991.

[12] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers Special Issue On Fault-Tolerant Computing*, 41(5), May 1992.

[13] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):48–59, January 1982.

[14] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing.* PhD thesis, Rice University, December 1989.

[15] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 14–19, June 1987.

[16] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[17] P. M. Melliar-Smith, L.E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[18] L.E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 480–489, May 1991.

[19] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

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

[21] F. Schmuck. *The Use of Efficient Broadcast Primitives in Asynchronous Distributed Systems.* PhD thesis, Cornell University, 1988.

[22] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–320, December 1990.