# External Logging on a Logical Ring

Jerry James    Ambuj K. Singh

Department of Computer Science
University of California, Santa Barbara

**Abstract**

We explore fault-tolerance of object-oriented programs in a shared-nothing environment. The object-oriented model considered allows remote method invocations, concurrent threads within an object, nested transactions, and transparent object migration. We adopt a logical ring structure on the set of processors to simplify the placement of logs, changes to the processor set, and recovery from failures. This ring is used only for fault tolerance messages; application messages use the full underlying network topology. We present algorithms for maintenance of the logical ring and the necessary recovery procedures. We present a message combining technique that reduces the cost of sending multiple log messages.

# 1 Introduction

We consider fault-tolerance of object-oriented programs in a shared-nothing environment, where disks of a processor become unavailable after a crash and therefore cannot be used for recovering the failed processor. The log or recovery information for a processor thus has to be maintained *externally* on other processors. We consider crash failures in which a failed processor stops taking steps and failures are eventually detected.

Our motivation for this work comes from the Kan system[1] that is currently being developed to support parallel and distributed programming. Kan, which is Java-based, targets an object-oriented programming model that supports (i) asynchronous method calls for supporting concurrent methods within an object; and (ii) nested transactions for supporting arbitrary atomic actions of a set of objects. Asynchronous method calls return a *future*, which is used to fetch results and for synchronization. Transactions provide a way of specifying arbitrary atomic actions on an object; nested transactions [23] provide such actions on multiple objects. The Kan system also allows transparent migration of objects, a necessary condition for masking the failure of processors.

A number of current systems support the primitives of the Kan system to various degrees. Arjuna [30] supports distributed fault-tolerant applications through nested atomic actions and replication, and also reentry of recovered nodes into the system. Concert [11] supports distributed objects with an aim towards providing fine-grained parallelism. Object-based concurrency control and encapsulation, and a dynamic concurrency model are provided. Orca [3] provides an object-based distributed shared memory, with guarded atomic actions on the distributed objects. The Obliq [8] language supports distributed object-oriented programs through migrating threads. A computation can roam over the network, while maintaining connections between its constituent parts. Nile [32] provides object replication and fault tolerance for extremely large-scale systems using CORBA as a data management layer. Distributed Oz [35] has language constructs for dataflow synchronization and the specification of object mobility patterns. Thor [13, 22] is an object-oriented database that supports mobile and replicated objects. Systems such as Totem [2] and Isis [6] provide virtual synchrony using reliable ordered message delivery over process groups. In contrast, Kan provides failure transparency via a transactional mechanism. Totem, in particular, also maintains a logical ring, but uses it for circulating a token that gives permission to broadcast. Our logical ring is used only by the fault tolerance subsystem; application messages travel directly on the underlying network. Our focus on the Kan system is to isolate a set of fault-tolerance requirements for distributed object-oriented programs. The algorithms and techniques developed in this paper will be applicable for any system that uses similar language primitives.

The Kan system distinguishes three types of objects to support different usage patterns (this is similar to the Munin Distributed Shared Memory system [9]). The object types are determined at runtime and can change over the lifetime of an object. *Migratory* objects are accessed by a single node at a time. *Producer-consumer* objects are accessed by a set of writing nodes (*producers*) and a set of reading nodes (*consumers*). Finally, *general* objects are neither migratory nor producer-consumer. For these, a home-based protocol with updates is used, along with a dynamic replication scheme based on access counts [36]. Read actions are performed locally, but write actions are sent to the home node, where they are serialized and broadcast to all replicas. In this paper, we assume that all objects are of the "general" type. (Extensions to the fault-tolerance system for supporting other object types are skipped for brevity.) In other words, every object has a home node where all the update methods are executed. Furthermore, the identity of the home node is part of each distributed object reference. Every object is linearizable [18]. This is supported through a protocol in which the home node acquires exclusive locks on all replicas. The replica nodes acquire a local read lock before executing any "read" methods. Nodes without a replica send requests to the home node. If a node

[1] http://www.cs.ucsb.edu/~dsl/Kan/

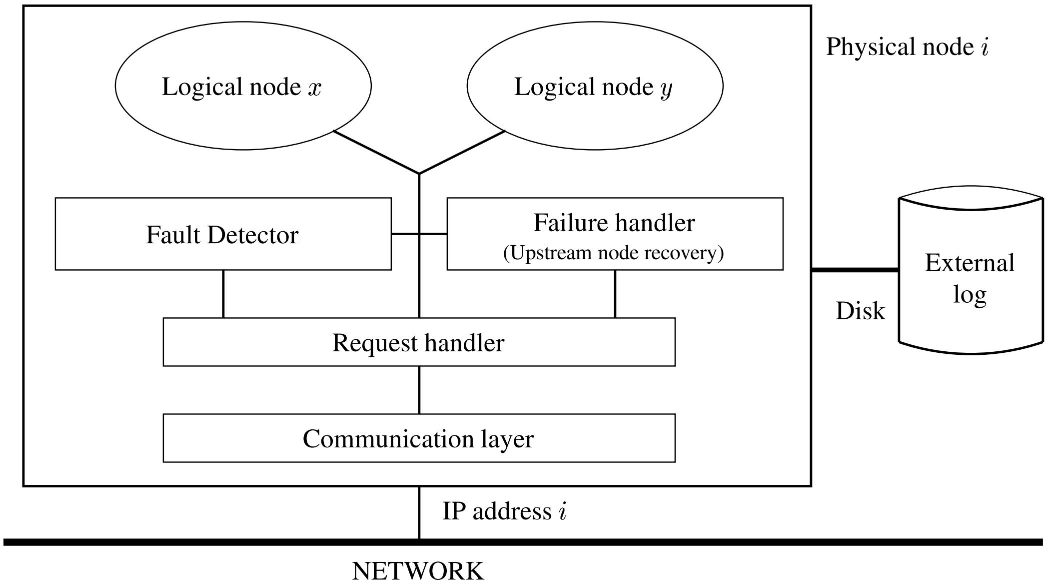


Figure 1: Basic system architecture

frequently invokes "read" operations, it will eventually be given its own replica.

Multi-object atomic operations are effected through nested transactions. Such transactions may be forced to abort, so we cannot change any object permanently until all can be changed. We copy all objects touched by a nested transaction and operate on the copies. In the event of an abort, we simply discard the copies, retaining the original version of each object. If all subtransactions complete successfully, we use a 2-phase commit algorithm [17] to install the changed version of each object at its home node. The home node updates its replicas upon receiving the new version.

Fault tolerance is a concern for distributed object-oriented systems. The failure of a single node can block the entire system, due to unreachable objects and incomplete subthreads from the failed node, even if the underlying system does not block. The ability to recover objects and threads from failed nodes enables long-lived applications, such as collaborative editors and other shared environment applications.

Our fault tolerance design is targeted at large distributed systems. In such systems *backward recovery*, where all nodes are rolled back to some checkpoint on a failure, may incur intolerably high overheads due to the frequency of failures. Instead, we adopt a *forward recovery* scheme, in which the failed nodes are recovered to their pre-failure states. We use logs to record system activity. Large systems also tend to be shared-nothing; this explains our interest in external logging techniques.

The basic architecture of our system is shown in Figure 1. Each physical node has a communication layer that can fail due to a timeout, and a request handler that processes incoming messages. Timeout failures are handled by the fault detector, which notifies the failure handler of failures. The failure handler in turn is responsible for recovery actions. These include maintenance of a logical ring and local recovery of other *logical nodes*. A logical node is a collection of user applications, home node managers for a set of objects, replica managers for some objects, and transaction managers. We mask the failures of logical nodes by migrating them across physical nodes; this may result in multiple logical nodes residing at a physical node. The disk is used for storing the external log of other physical nodes.

The fault-tolerance algorithms proposed in this paper build on a number of existing ideas in the literature,

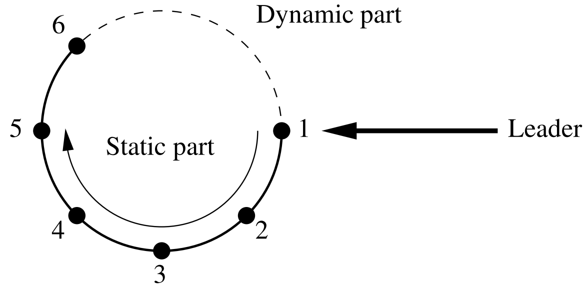


Figure 2: Logical ring with leader

namely view maintenance [6, 10, 28, 19, 20, 21, 24, 27], virtual synchrony [4, 6, 7, 14, 26, 33], causal logging [1], and two-phase commit [17]. The former two have been used to keep actions consistent in the presence of failures. Causal logging is aimed at providing logging services without adding any additional messages. The cost is that messages tend to become very large. We propose a technique for message combining that reduces the size of messages. Finally, two-phase commit is a classic algorithm for ensuring uniform actions among a group of processes. We use a similar idea to ensure that all correct nodes add a recovering node back into ring, even if failures occur during the recovery process. Our contribution in this paper is twofold. First, we use a logical ring to simplify and localize the recovery task. Second, we present a message-combining technique that reduces the message and time complexity of logging multiple messages. Coupled together, these algorithms provide a mechanism for achieving fault-tolerance of object-oriented applications over large networks by providing a basis for logging, much as view maintenance and virtual synchrony provide a basis for coordinated actions in the presence of failures.

We assume that the user application is free from data races. This means that conflicting accesses to objects occur inside atomic actions (or transactions which for us play the role of acquire-release blocks in release consistency [15]), i.e., at no reachable state of the system are non-commuting actions enabled. This assumption simplifies the recovery of threads since the reads-from information of a thread needs to be gathered only at the points where the thread acquires a read lock [12, 25, 29, 31, 34, 37].

The rest of the paper is organized as follows. Section 2 discusses the idea of a logical ring and explains how it simplifies the various recovery actions. Section3 discusses the ring maintenance algorithm. The protocol for the addition of new nodes into the ring and the recovery of the leader is also presented there. Section 4 discusses our message combining technique that reduces the overhead of logging. We end the main body of the paper with some concluding remarks in Section 5.

## 2 Fault Tolerance on a Logical Ring

We use a logical ring structure to minimize the need for global synchronization and recovery. This ring is only used for logging, in contrast to systems such as Totem [2]; applications send their messages directly on the underlying network. The messages that we are interested in logging are remote method invocations. There is no logging of messages associated with transactions, since the transaction managers use a 2-phase commit protocol for fault-tolerance.

The logical ring, illustrated in Figure 2, is divided into two parts: a *static* part containing the active correct nodes and a *dynamic* part containing nodes that are trying to join the ring. Global synchronization is avoided over the static part of the ring; in fact if no nodes ever joined the ring then there would be no dynamic part and no global information exchange. The possibility of new nodes joining the ring forces a dynamic part in the ring. The ring has a *leader* and a sense of direction (clockwise is *downstream*, counter-clockwise is *upstream*). The size of the ring is $n$, the number of processors. Let parameter $T$ indicate the maximum number of node failures that need to be tolerated. We assume that $n > 2T$, where $n$ is the number of nodes. A number of fault-tolerance and recovery actions are simplified with the help of a ring. We discuss some of these here before presenting the complete algorithm.

*Logging location:* With an external logging system, each logical node is logged at $T$ physical nodes. For optimistic or pessimistic logging schemes, these nodes must be chosen so that the log can be reassembled on a node failure. The reassembly process must work even if some of the $T$ logging locations have failed. With the logical ring, a logical node $x$ is logged at the next $T$ physical nodes in the ring. In particular, if $T$ or fewer failures occur, the recovery node chosen for any failed node will already have a copy of the failed node's log, simplifying recovery.

*Message combining:* One of our goals is to reduce logging traffic over pessimistic and optimistic logging, and to reduce message size over causal logging. By ensuring symmetry in the transmission of log messages, we are able to combine them. This not only reduces the message traffic but also the number of non-stable messages for causal logging.

*Logical node recovery:* Initially, there is a one-to-one mapping from logical nodes to physical nodes. As physical nodes fail, logical nodes are remapped to different physical nodes. To simplify locating a logical node, we remap a logical node from a failed process to the next process in the logical ring. For example, in Figure 2, if physical node 4 fails, then the logical node that it supports is remapped to physical node 5, and physical node 4 is removed from the logical ring. This remapping is almost always local (as opposed to requiring leader participation in other protocols); the only instance when we require leader participation is when the remapped node lies across the dynamic segment of the ring (e.g., the failure of node 6 in Figure 2).

*Object location:* Each distributed object reference in the application program consists of a logical node identifier (the home node) and a sequence number. The home node is responsible for maintaining (possibly stale) information on the object's whereabouts, including its replica set. Finding an object whose logical node host has failed then becomes a process of waiting for the next physical node in the ring to recover the logical node. This reduces the possibly expensive cost of searching (or informing) the entire system for an object after the failure of a home node.

*Recovery of nodes and leader:* Addition of a recovered node to a system can be just as complex as tolerating faults in the first place. In our protocol, the leader coordinates the return of recovered nodes to the system. A recovering node queues up at the leader and waits until the leader installs the correct view at the new node and updates the ring structure (by extending the static part) at the rest of the nodes. If the leader fails, the next downstream node is elected as the new leader. The logical ring structure again greatly simplifies the addition of new nodes and the recovery of the leader.

# 3 Logical Ring Maintenance

In this section, we describe how the logical ring is maintained in the presence of failing and recovering nodes. As mentioned earlier, the static part of the ring does not need any global maintenance. The dynamic segment can be maintained using any virtual synchrony algorithm [4, 14, 26]. However, the use of a logical ring structure allows us to simplify, as described below. Management of the static segment is also simplified, due to the leader. This leader coordinates the return of recovered nodes to the ring. If the leader fails, then the next node in the ring becomes the new leader. In this way, our protocol is similar to the Isis protocol [5, 6], where if a view manager fails, the new view manager is the oldest non-failed process.

| Variable | Description |
|---|---|
| $failed_i(j)$ | true if $i$ has detected the failure of $j$ |
| $ready_i$ | true if $i$ is ready to accept requests for its logical nodes |
| $map_i(x)$ | the physical node on which logical node $x$ resides |
| $view_i$ | $i$'s *view* of the logical ring (membership and order) |
| $leader_i$ | $i$'s view of the leader of the ring |
| $pending_i$ | the set of physical nodes that $i$ suspects of failing |
| $recovery\_count_i$ | the number of logical nodes that need to be recovered |
| $new\_node_i$ | the most recent physical node to try to enter the ring |
| $pending\_addition_i$ | true if some physical node is currently trying to enter the ring |
| $adding\_leader_i$ | the node trying to add $new\_node_i$ to the ring |
| $inform\_set_i$ | the nodes to inform of the entry of $new\_node_i$ into the ring |

Figure 3: Ring Variables

Each node $i$ participating in our protocol maintains the variables listed in Figure 3, where $j$ ranges over physical nodes and $x$ ranges over logical nodes. The $failed_i(j)$ predicate records whether $i$ has detected the failure of $j$. Since $j$ assumes a new identity on recovery, this predicate is *stable*. Note that $i$ may be forced to set $failed_i(i)$ and recover if some other process is unable to communicate with it. The $ready_i$ predicate records whether $i$ is active. There is an initial set of ready nodes. New nodes become ready when they are linked into the ring.

### 3.1 Failure Handling

We assume a failure detector at each physical node. When node $i$ is informed of the failure of node $j$ by the failure detector, it adds $j$ to *pending*. The processing of these failure notifications is shown in Figure 4.

First consider the $\neg pending\_addition_i$ case, where node $i$ is not trying to add any other node to the ring. We test for the following subcases, where $j$ is the failed node:

- If every node upstream of $i$ has failed, then $i$ must become the new leader. It remaps all logical nodes from the upstream physical nodes, and informs the other correct nodes by sending a remap message. It then recovers the logical nodes.
- If the leader has failed but there is some upstream node $k$ that will become the new leader, then just update the *map* and *leader* variables to reflect the new situation.
- If the failed node $j$ is upstream of $i$, then just update *map*. If $i$ is the next downstream node from $j$, also recover the logical nodes from $j$.
- If $j$ is downstream of $i$ and there is some node $k$ downstream of $j$, then just update *map*.
- If $j$ is downstream of $i$ and there is no node downstream of $j$, then wait for the leader to update *map*. This is necessary since the leader may know of some recovering node that will recover $j$.
- If $i$ is the leader and must recover $j$, then change *map*, send a remap message to change the correct nodes' maps, and recover all logical nodes that are now mapped locally.

| Condition | Action |
|---|---|
| Upon detecting failure of $j$ | $pending_i := pending_i \cup \{j\}$ |
| Receive $\langle \mathbf{remap}, x, j, k \rangle \wedge failed_i(j : k) \wedge leader_i = k$ | $map_i(x) := k$ |
| $pending_i \neq \emptyset \wedge$ all recovery is complete for the previously removed element of $pending_i$ | $j :=$ lowest priority element of $pending_i$, where priority is assigned clockwise from $leader_i$ to $i$ to $leader_{i-1}$<br>$pending_i := pending_i - \{j\}$<br>$failed_i(j) :=$ true;<br>Case<br>$\neg pending\_addition_i \wedge j = leader_i \wedge failed_i(j : i-1)$:<br>$leader_i := i$;<br>$(\forall x : map_i(x) = j ::$<br>$map_i(x) := i$;<br>send $\langle \mathbf{remap}, x, j, i \rangle$ to all nodes;<br>$recovery\_count_i := recovery\_count_i + 1$;<br>fork $Node\_Recovery(x, i)$; )<br>$(\forall k : k \in inform\_set_i \wedge i < k ::$<br>send $\langle \mathbf{release}, new\_node_i, adding\_leader_i \rangle$ to $k$; )<br>$\neg pending\_addition_i \wedge j = leader_i \wedge \neg failed_i(j : i-1)$:<br>$k :=$ first non-failed process in $(j : i-1)$;<br>$(\forall x : map_i(x) = j :: map_i(x) := k;)$<br>$leader_i := k$;<br>$\neg pending\_addition_i \wedge leader_i < j < i$ :<br>$k :=$ first non-failed process in $(j : i)$;<br>if $i \neq k$ then<br>$(\forall x : map_i(x) = j :: map_i(x) := k;)$<br>else /* recover $j$ and its failed downstream neighbors at $i$ */<br>$(\forall x : map_i(x) = j ::$<br>$map_i(x) := i$;<br>$recovery\_count_i := recovery\_count_i + 1$;<br>fork $Node\_Recovery(x, i)$; )<br>end if<br>$\neg pending\_addition_i \wedge leader_i \leq i < j \wedge \neg failed_i(j : leader_i - 1)$:<br>$k :=$ first non-failed process in $(j : leader_i - 1)$;<br>$(\forall x : map_i(x) = j :: map_i(x) := k;)$<br>$\neg pending\_addition_i \wedge leader_i < i < j \wedge failed_i(j : leader_i - 1)$:<br>skip /* wait for leader to change $map$ */<br>$\neg pending\_addition_i \wedge leader_i = i \wedge failed_i(j : leader_i - 1)$:<br>$(\forall x, k : failed(k : j) \wedge map_i(x) = j ::$<br>$map_i(x) := i$;<br>send $\langle \mathbf{remap}, x, k, i \rangle$ to all nodes<br>$recovery\_count_i := recovery\_count_i + 1$;<br>fork $Node\_Recovery(x, i)$; )<br>$pending\_addition_i \wedge failed_i(leader_i : i-1)$:<br>$Link\_Node$ (true)<br>end Case; |

Figure 4: Ring maintenance algorithm: failure actions

In the second case, there is some node ($new\_node_i$) trying to join the ring when $i$ processes the failure of $j$. Then $i$ waits to see if the new node will recover $j$. If $i$ is the leader (or will become the leader due to failures), $i$ tries to finish the addition of $new\_node_i$ to the ring by asking other nodes to link it in.

### 3.2 Physical Node Recovery

When a physical node comes back up, it looks for the current leader. It sends a join message to the leader, asking to be linked into the ring. The leader works in three phases. In the first phase, it sends acquire messages to $T$ downstream nodes in turn. The first phase completes after these nodes have responded with grant messages. The leader begins the second phase by sending add messages to all nodes in the ring on behalf of the recovering node. Upon receiving an add message, each node updates its view to include the recovering node, positioned just upstream of the leader (i.e., at the end of the static segment). Once the add messages are acknowledged, the leader sends an ack_add message to the recovering node, containing a copy of the physical-to-logical node map, the failed nodes, and the current ring membership. Then, in the third phase, the leader sends release messages to the $T$ downstream nodes as in the first phase.

### 3.3 Recovery of Leader

The leader of the ring maintains information on the current membership of the ring, and manages addition of nodes to the static part. When the leader fails, the next downstream node in the ring must take over this task. There is a possibility that the leader was in the middle of adding a node to the ring. If a failure occurs in the first phase, then either the new leader received the acquire message or it did not. If it did not, then it begins by sending release messages to the next $T$ downstream nodes, thereby cancelling any in-progress addition of a node. If it did receive the acquire, then it continues the protocol on behalf of the recovering node. If the leader fails at any point before sending a release message to the eventual new leader, the new leader will restart the recovery process. The complete protocol is shown in Figures 5 and 6.

### 3.4 Other Recovery Actions

We briefly summarize a number of important recovery components that we do not have space to discuss. An object can be locked in a *read* mode, a *write* mode, a *committing read* mode, and a *committing write* mode. A *read* lock on an object is obtained either at the home node of the object or at a replica node. A *write* lock is always obtained at the home node of the object. A write lock does not preclude read locks at the replica nodes. The *committing* locks are held by a transaction that has prepared to commit. Committing locks conflict with all other lock types.

When ready to commit, the transaction manager logs the new states of all the objects it updated. Then it requests the appropriate committing locks by contacting the home nodes of the respective objects. In order to grant a *committing write* lock, the home node contacts all the replicas and waits until the read locks (if any) on the object replicas are released. After the transaction manager has obtained all the required locks, it writes a commit record to its log. After this, the transaction manager releases all the locks.

In the event of the failure of the transaction manager, the *read* and *write* locks can be released unilaterally. For the *committing* locks, the home nodes of the objects wait until the transaction manager is recovered. After recovery, the transaction manager consults the log to decide on whether to abort the transaction or to complete its commitment.

A home node logs the new state after each update of the object. In the event of failure of a home node of an object, the nodes holding a replica of the object discard the object. The home node will be eventually

| Condition | Action |
|---|---|
| $leader_i = i \wedge$<br>$recovery\_count_i = 0 \wedge$<br>$\neg pending\_addition_i \wedge$<br>receive $\langle \mathbf{join} \rangle$ from $j$ | $pending\_addition_i := \mathsf{true}$;<br>$Add\_node(j)$; |
| $leader_i \neq i \wedge$<br>receive $\langle \mathbf{join} \rangle$ from $j$ | send $\langle \mathbf{nack} \rangle$ to $j$; |
| $Node\_Recovery(x, j, i)$ | Install logical node $x$ at $i$ using log;<br>$recovery\_count_i := recovery\_count_i - 1$; |
| $Add\_Node(j)$ | $inform\_set_i := \emptyset$;<br>$new\_node_i := j$;<br>$adding\_leader_i := i$;<br>$k := i$;<br>for $local := 1$ to $t$ do<br>    $k :=$ next non-failed node in $(k : *)$;<br>    $inform\_set_i := inform\_set_i \cup \{k\}$ ;<br>end do;<br>$Link\_Node$ (false) |
| $Link\_Node(check\_view)$ | if $\neg check\_view \vee new\_node_i \notin view_i$ then<br>    $(\forall k : k \in inform\_set_i \wedge i < k ::$<br>        sequentially along the logical ring,<br>          send $\langle \mathbf{acquire}, new\_node_i, adding\_leader_i, inform\_set_i \rangle$ to $k$;<br>        wait until $failed_i(k)$ or $\langle \mathbf{grant} \rangle$ is received from $k$; )<br>    Add $new\_node_i$ to $view_i$ with $i$ as its downstream neighbor;<br>end if<br>$(\forall k : k \in view_i \wedge k \neq new\_node_i ::$<br>    sequentially along the logical ring<br>        send $\langle \mathbf{add}, new\_node_i, adding\_leader_i \rangle$ to $k$;<br>    wait until $failed_i(k)$ or $\langle \mathbf{ack\_add} \rangle$ is received from $k$; )<br>send $\langle \mathbf{add\_new}, map_i, failed_i, view_i \rangle$ to $new\_node_i$;<br>wait until $failed_i(new\_node_i)$ or $\langle \mathbf{ack\_add} \rangle$ is received from $new\_node_i$;<br>$(\forall k : k \in inform\_set_i \wedge (i < k \vee \neg check\_view) ::$<br>    send $\langle \mathbf{release}, new\_node_i, adding\_leader_i \rangle$ to $k$; )<br>$pending\_addition_i := \mathsf{false}$; |
| Receive $\langle \mathbf{acquire}, k, l, S \rangle$<br>from $j \wedge leader_i = j \wedge$<br>$recovery\_count_i = 0$ | $pending\_addition_i := \mathsf{true}$;<br>$inform\_set_i := S$;<br>$new\_node_i := k$;<br>$adding\_leader_i := l$;<br>send $\langle \mathbf{grant} \rangle$ to $j$; |
| Receive $\langle \mathbf{release}, k, l \rangle$<br>from $j \wedge leader_i = j$<br>$\wedge\ new\_node_i = k$<br>$\wedge\ pending\_addition_i$<br>$\wedge\ adding\_leader_i = l$ | $pending\_addition_i := \mathsf{false}$; |
| Receive $\langle \mathbf{add}, l, k \rangle$ from $j$<br>$\wedge\ leader_i = j$ | Add $k$ to $view_i$ with $l$ as its downstream neighbor;<br>send $\langle \mathbf{ack\_add} \rangle$ to $j$; |

Figure 5: Ring maintenance algorithm: recovery actions

| Condition | Action |
|---|---|
| Upon recovery | Obtain a new physical node ID $i$;<br>$failed_i(i) :=$ false;<br>$ready_i :=$ true;<br>$recovery\_count_i := 0$;<br>$pending\_addition_i :=$ false;<br>repeat<br>    $leader_i := j$; /* set by guess or polling */<br>    send $\langle \mathbf{join} \rangle$ to $leader_i$;<br>    wait until $failed_i(j)$ or $\langle \mathbf{nack} \rangle$ or $\langle \mathbf{add\_new}, f, g, h \rangle$ is received;<br>    if $\langle \mathbf{add}, f, g, h \rangle$ was received then break;<br>forever;<br>$map_i := f$;<br>$failed_i := g$;<br>$view_i := h$; |

Figure 6: Ring maintenance algorithm: recovery actions (cont.)

installed at another physical node.

The reads-from relationship is maintained locally at each replica and home node. When a thread obtains a read lock on an object, an access record indicating the thread id and the version number of the object it reads is written to the memory. This access record is logged when the corresponding object is updated or when the replica is removed from the node. The assumption of data-race-freedom allows us to monitor and log the access records at these specified points.

Application threads are recovered using the method invocation logs (in order to restart the threads). The reads-from log tells us which version of each object the thread must read on replay. Finally, version number information is logged when an object is written. This gives us the proper ordering between writing threads.

# 4 Combining Log Messages

Since we adopt forward recovery, a message needs to be logged before it is delivered (pessimistic logging) or the causal past of a message needs to be carried with it (causal logging). Our logging scheme is based on an efficient message-combining technique that reduces the overhead of logging multiple messages. This technique can be used with either pessimistic logging or causal logging. When used with pessimistic logging, our technique reduces the length of the blocking period and the number of messages, and when used with causal logging, the technique stabilizes messages quickly so that the causal past carried with each message can be reduced. Our challenge is that naive message combining can make logs inaccessible when a node fails early in the combining process. Integration with either a pessimistic or causal logging scheme enables us to recover a consistent state in such a case. Similar message combining techniques have been explored elsewhere [16] in the context of multicasting a set of messages.

Our message combining technique can be used whenever processors on a ring need to send a group of messages to downstream processors. Assume that in a ring (with a sense of direction) of $n$ processors, every processor needs to send a message to $T$ downstream nodes. Assume that $T = 2^t - 1$ and that $n$ is a multiple of $T$. We first describe how the message of a single node $x$ is diffused downstream (using the ring's sense of direction) to $T$ nodes. Later, we discuss how these diffusion trees can be combined together.

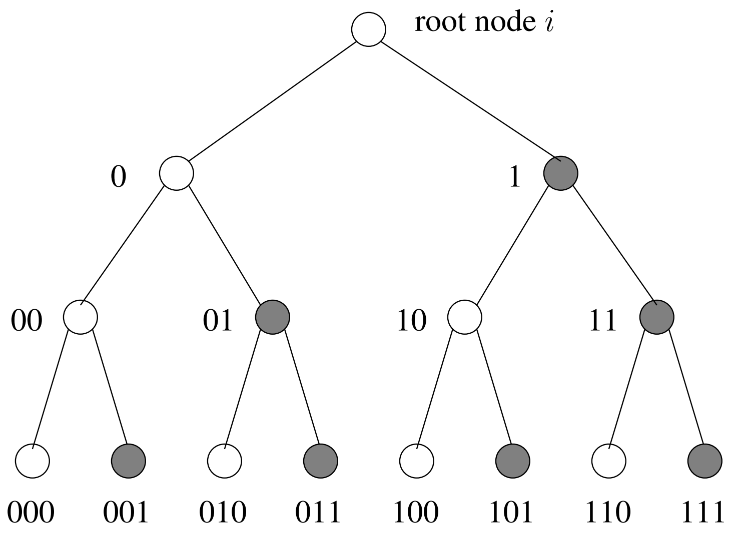


Figure 7: Diffusion tree

| Condition | Action |
|---|---|
| Root node | send $\langle 1, m\rangle$ to downstream node at distance $(T+1)/2$<br>Assume label $0$ |
| Upon receiving a message $\langle u.m\rangle$ | assume label $u$<br>log message $m$ |
| Node labeled $u$ | if $(T+1) > 2^{\|u\|}$ then<br>    send $\langle u1, m\rangle$ to downstream node at distance $(T+1)/2^{\|u\|}$;<br>    assume label $u0$<br>else<br>    stop |

Figure 8: Diffusion Algorithm

### 4.1 Diffusion Algorithm

The diffusion algorithm works in $t$ rounds and can be understood with the help of a "diffusion" tree shown in Figure 7. Node $x$ that wishes to send the message is the root node of a diffusion tree of height $t$. In the first round, the root node sends a message to a node at a distance of $(T+1)/2$; this node is labeled 1 in the figure. The root node now assumes the identity of the node labeled 0 in the figure. In the next round, these two nodes (labeled 0 and 1) send a message to nodes downstream that are a distance $(T+1)/4$ away. These new nodes are labeled 01 and 11 in the figure. The old nodes (labeled 0 and 1) assume the identity of nodes labeled 00 and 10 in the figure. New nodes that receive the transmitted message are shown by shaded circles. As evident from the figure, the parent node forwards the message to the right child and assumes the identity of the left child in each round. This diffusion repeats for $t$ rounds when all downstream nodes at most $T$ away from the root node receive the message. The complete algorithm is shown in Figure 8.

Another way to visualize the diffusion tree is to use a binary representation to measure distance between nodes. A node that is labeled with a string $u$ in the diffusion tree of node $x$ is at a distance $d(u)$ downstream from $x$:

$$d(u) = uv, \text{ where } v = 0^{t-|u|}$$

In other words, distance $d(u)$ is computed by padding it with enough zeroes to the right in order to make it a binary string of length $t$. Thus, $d(0) = d(00) = 0$, $d(1) = d(10) = d(100) = 10^{t-1} = (T+1)/2$, $d(01) = d(010) = 10^{t-2} = (T+1)/4$, and $d(11) = d(110) = 3 \times (T+1)/4$. Examining Figure 7, the message sent by node $x$ is at downstream nodes $d(0), d(1)$ after round 1, at nodes $d(00), d(01), d(10), d(11)$ after round 2, and in general at nodes $\{d(u) \; | u \text{ is any binary string of length } i\}$ after round $i$.

## 4.2 Message Combining Algorithm

In a logical ring of length $n$, there are $n$ diffusion trees and message combining is achieved by operating on these trees simultaneously. In order to motivate the message combination, we first consider the situation when the diffusion trees operate independently. Let a distance $-d(u)$ indicate a node that is at an upstream distance from our reference node $x$.

In the first round, node $x$ sends a message labeled 1 to node $d(1)$ and receives a message labeled 1 from node $-d(1)$. Ignoring the contents of the message for the time being, let us denote a message by $\langle y, z, w : u\rangle$, where $y$ identifies the sender (with respect to the reference node), $z$ identifies the receiver (with respect to the reference node), $w$ identifies the originating diffusion tree (again with respect to the reference node), and $u$ indicates the label. Thus, in round 1 node $x$ sends a message $\langle d(0), d(1), -d(0) : 1\rangle$, and receives a message $\langle -d(1), d(0), -d(1) : 1\rangle$. In the second round, node $x$ sends messages $\langle d(0), d(01), -d(0) : 01\rangle$, $\langle d(0), d(01), -d(1) : 11\rangle$ (corresponding to the two messages it has at the end of last round), and receives messages $\langle -d(01), d(0), -d(01) : 01\rangle$, $\langle -d(01), d(0), -d(11) : 11\rangle$. In general, in round $i$ node $x$ sends messages $\{\langle d(0), d(0^{i-1}1), -d(u) : u1\rangle \, | u \text{ is any binary string of length } i-1\}$, and receives messages $\{\langle -d(0^{i-1}1), d(0), -d(u1) : u1\rangle \; | u \text{ is any binary string of length } i-1\}$. (Let $d$ (null string) $= 0$.) Note that at the end of round $i$, node $x$ has received the cumulative messages $\{\langle *, d(0), -d(u) : u\rangle \; | u \text{ is any binary string of length } i\}$, where the don't-care term is determined by the round in which the message was received as follows: the term is $-d(0^{i-1})$ and the round is $i$ for $u$ that end in 1, the term is $-d(0^{i-2})$ and the round is $i-1$ for $u$ that end in 10, and so on.

Since node $x$ sends all its messages in round $i$ to the same node, namely $d(0^{i-1}1)$, and receives all its messages from the same node, namely $-d(0^{i-1}1)$, all the messages sent by each node can be combined together into a single message. This results in the following message combining algorithm. Set $R$ contains the messages received so far and is initialized to the message to be sent by node $x$.

Initially, $R = \{m_0\}$.
Begin
    For $i := 1$ to $t$ do
        $Comb := \langle m | m \in R\rangle$ ;
        send message $Comb$ to node $d(0^{i-1}1)$;
        Receive message $Comb$ from node $-d(0^{i-1}1)$;
        $R := R \cup \{m | m \in Comb\}$;
    end loop
End

### 4.3 Performance Analysis

The total number of messages sent in the message combining algorithm is $O(n \times t)$. This is a large improvement from $O(n \times T)$ messages sent by an algorithm that does not combine messages. A similar analysis also holds for the response time: assuming a worst case message delay of $\delta$, the message combining algorithm completes in time $O(t \times \delta)$ as opposed to $O(T \times \delta)$. (Note that we can achieve $O(n)$ messages by selecting some node as coordinator, which gathers and then broadcasts all logs. The difficulty is that the coordinator node becomes a bottleneck on the network. Our approach avoids this problem. Note also that $\delta$ need not be a constant; message combining sends no more, and possibly fewer, bits than with no message combining. Hence, the relevant network parameters should not be any worse for the message combining case.)

The above analysis assumes that the nodes can be synchronized into rounds. In an asynchronous system this synchronization can be achieved in a number of different ways. A global synchronization will introduce $O(1)$ messages per node and $O(\log n)$ delay. Another approach is to advance a node through the rounds after a suitable timeout interval when it receives an upstream message or when it has a pending message to send. Another possibility is for the leader to begin the first round periodically by sending a token to its downstream node. The proper optimization choice depends on the application and network characteristics.

The message combining algorithm can be used with either pessimistic logging to reduce the length of the blocking period and the number of logging messages or with causal logging to reduce the size of messages. The impact of sending longer messages due to message combining is likely to be minimal since the messages can be logged using their discriminants that are quite small [1]. Furthermore, most networks scale well with respect to the size of messages.

The choice of pessimistic logging versus causal logging will ultimately depend on the application characteristics. Though pessimistic logging blocks the sending thread, for a multithreaded environment this does not present too much of a problem. Such a blocked thread can be descheduled in favor of another ready thread. For applications where such switching is not possible, causal logging may be adopted. Next, we sketch an algorithm that combines our technique with causal logging.

In causal logging, every non-stable application message carries the discriminants of the non-stable messages of its causal past. Our message-combining algorithm attempts to stabilize each message and thus reduce the non-stable causal past of each message. We maintain a counter with every node that indicates the last logged message of the node. This counter is incremented by the message combining algorithm. A process also keeps an estimate (a lower bound) on the counter values of other nodes. All this information is stored in an array $count$ at each node $x$; $count[y]$ indicates the estimate of the counter of $y$ as known at $x$ and is incremented by piggybacking on logging and application messages. The causal past of every message $m$ to be transmitted by a node $x$ is determined as in the causal logging algorithms. A message $m'$ that is sent by node $y$ and that is in the causal past of message $m$ is non-stable provided its sequence number is higher than $count[y]$. The discriminants of all such messages are concatenated to message $m$. The number of non-stable messages of any node can be estimated by $O(t)$, the number of rounds of the logging (message combining) algorithm. The non-stable causal past of any message then can be estimated by $O(t^2)$, which is also a bound on the number of discriminants carried by each application message.

## 5 Conclusions and future work

In this paper, we have presented an algorithm for logging and recovery of distributed object-oriented programs. The object-oriented model includes nested transactions, transparent object migration, concurrent threads within an object, and asynchronous method calls. It is representative of several current and future languages and systems targeted at parallel and distributed programming.

We have shown how the use of a logical ring structure on the processors simplifies the different recovery actions. The logical ring has a static and a dynamic part and a leader that coordinates the entry of new nodes. For a large system, the single logical ring can be generalized to a tree of rings. This permits smaller memory requirements, $O(s)$ where $s$ is the size of each ring instead of $O(n)$, and concurrent addition of up to $n/s$ new nodes.

We discussed a message combining technique that uses the symmetry of logging to reduce the number of messages from $O(T)$ to $O(\log T)$ per node. This technique can be used with pessimistic logging to reduce the number of messages and the blocking period, and with causal logging to reduce the size of messages.

In the future, we plan to implement the algorithms as a part of the Kan system. We intend to experiment with a number of experimental benchmarks and applications that include different mixes of remote method invocations, object types, and transactions.

# References


[1] Alvisi, L., and Marzullo, K. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Softw. Eng. 24*, 2 (Feb. 1998), 149–59.

[2] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst. 13*, 4 (Nov. 1995), 311–42.

[3] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18*, 3 (Mar. 1992), 190–205.

[4] Birman, K., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug. 1991), 272–314.

[5] Birman, K. P., and Joseph, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 47–76.

[6] Birman, K. P., and Renesse, R. V., Eds. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[7] Bruso, S. A. A failure detection and notification protocol for distributed computing systems. In *ICDCS–5* (Denver, CO, USA, 13–17 May 1985), pp. 116–23.

[8] Cardelli, L. A language with distributed scope. *Computing Systems 8*, 1 (Winter 1995), 27–59.

[9] Carter, J. B. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing 29*, 2 (Sept. 1995), 219–27.

[10] Chang, J., and Maxemchuk, N. F. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 251–73.

[11] Chien, A. A., Dolby, J., Ganguly, B., Karamchetti, V., and Xhang, X. Supporting high level programming with high performance: The Illinois Concert system. In *Proc. of 2nd International Workshop on High-level Parallel Programming Models and Supportive Environments, held at IPPS'97* (Geneva, Switzerland, 1 Apr. 1997), pp. 15–24.

[12] COSTA, M., GUEDES, P., SEQUEIRA, M., NEVES, N., AND CASTRO, M. Lightweight logging for lazy release consistent distributed shared memory. In *OSDI '96* (Seattle, WA, USA, 28–31 Oct. 1996), USENIX Association, pp. 59–73.

[13] DAY, M., LISKOV, B., MAHESHWARI, U., AND MYERS, A. C. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems 2*, 1–4 (Mar.–Dec. 1993), 115–26.

[14] FRIEDMAN, R., AND VAN RENESSE, R. Strong and weak virtual synchrony in Horus. In *Proc. 15th Symposium on Reliable Distributed Systems* (Niagara-on-the-Lake, Ontario, Canada, 23–25 Oct. 1996), pp. 140–9.

[15] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90* (Seattle, WA, USA, 28–31 May 1990), pp. 15–26. Revised in Stanford CSL Technical Report CSL-TR–93–568.

[16] GONZALEZ, T. F. Distributed algorithm for multimessage multicasting. Tech. Rep. TRCS98–23, University of California, Santa Barbara, 1998.

[17] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1993.

[18] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–92.

[19] JAHANIAN, F., FAKHOURI, S., AND RAJKUMAR, R. Processor group membership protocols: Specification, design, and implementation. In *Proc. 12th Symposium on Reliable Distributed Systems* (Princeton, NJ, USA, 6–8 Oct. 1993), pp. 2–11.

[20] JAHANIAN, F., AND MORAN, W. L. Strong, weak, and hybrid group membership. In *2nd Workshop on the Management of Replicated Data* (Monterey, CA, USA, 12–13 Nov. 1992), J. F. Paris and H. Garcia-Molina, Eds., pp. 34–8.

[21] KAASHOEK, M. F., AND TANENBAUM, A. S. Group communication in the Amoeba distributed operating system. In *ICDCS-11* (1991), pp. 222–30.

[22] LISKOV, B., ADYA, A., CASTRO, M., DAY, M., GHEMAWAT, S., GRUBER, R., MAHESHWARI, U., MYERS, A. C., AND SHRIRA, L. Safe and efficient sharing of persistent objects in Thor. In *SIGMOD '96* (Montreal, Quebec, Canada, 4–6 June 1996), pp. 318–29.

[23] LYNCH, N., MERRIT, M., WEIHL, W., AND FEKETE, A. *Atomic Transactions*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[24] MISHRA, S., PETERSON, L. L., AND SCHLICTING, R. D. A membership protocol based on a partial order. In *DCCA–2* (Tucson, AZ, USA, 18–20 Feb. 1991), J. F. Meyer and R. D. Schlichting, Eds., pp. 309–31.

[25] MORIN, C., AND PUAUT, I. A survey of recoverable distributed shared virtual memory systems. *IEEE Transactions on Parallel and Distributed Systems 8*, 9 (Sept. 1997), 959–69.

[26] MOSER, L. E., AMIR, Y., MELLIAR-SMITH, P. M., AND AGARWAL, D. A. Extended virtual synchrony. In *ICDCS-14* (Poznań, Poland, 21–24 June 1994), pp. 56–65.

[27] MOSER, L. E., MELLIAR-SMITH, P. M., AND AGRAWALA, V. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems 5*, 5 (1994), 459–73.

[28] MOSER, L. E., MELLIAR-SMITH, P. M., AND NARASIMHAN, P. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems 4*, 2 (1998), 81–92.

[29] NEVES, N., AND FUCHS, W. K. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *FTCS '98* (Munich, Germany, 23–25 June 1998), pp. 38–47.

[30] PARRINGTON, G. D., SHRIVASTAVA, S. K., WHEATER, S. M., AND LITTLE, M. C. The design and implementation of Arjuna. *Usenix Computing Systems Journal 8*, 3 (Summer 1995), 255–308.

[31] PLANK, J. S., AND ELWASIF, W. R. Experimental assessment of workstation failures and their impact on checkpointing systems. In *FTCS '98* (Munich, Germany, 23–25 June 1998), pp. 48–57.

[32] RICCIARDI, A., OGG, M., AND PREVIATO, F. Experience with distributed replicated objects: the Nile project. *Theory and Practice of Object Systems 4*, 2 (1998), 107–15.

[33] SCHIPER, A., AND SANDOZ, A. Uniform reliable multicast in a virtually synchronous environment. In *ICDCS-13* (Pittsburgh, PA, USA, 25–28 May 1993), pp. 561–8.

[34] SLYE, J. H., AND ELNOZAHY, E. N. Supporting nondeterministic execution in fault-tolerant systems. In *FTCS '96* (Sendai, Japan, 25–27 June 1996), pp. 250–9.

[35] VAN ROY, P., HARIDI, S., BRAND, P., SMOLKA, G., ET AL. Mobile objects in Distributed Oz. *ACM Trans. Program. Lang. Syst. 19*, 5 (Sept. 1997), 804–51.

[36] WOLFSON, O., JAJODIA, S., AND HUANG, Y. An adaptive data replication algorithm. *ACM Trans. Database Syst. 22*, 2 (1997), 255–314.

[37] XU, J., RANDELL, B., ROMANOVSKY, A., RUBIRA, C. M. F., STROUD, R. J., AND WU, Z. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS '95* (Pasadena, CA, USA, 27–30 June 1995), pp. 499–508.