Consistency of Partitionable Object Groups in a CORBA Framework*

P. Narasimhan, L. E. Moser, P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106
priya@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

The Eternal system provides a novel methodology for the consistent replication of objects in an adaptive, fault-tolerant, CORBA-compliant distributed system that is susceptible to partitioning. Passive and active replication schemes are supported in Eternal, and mechanisms are provided for the interaction of objects with different replication schemes. Nested operations for both passive and active objects are accommodated. Suppression of duplicate operations is ensured by unique message and operation identifiers. Continued operation is allowed in all components of a partitioned system. State transfer mechanisms and fulfillment operations restore the consistency of the states of replicas within the components of a partitioned system when communication is reestablished and the components remerge.

1 Introduction

Distributed systems are clusters of computers that operate autonomously and yet cooperate to perform an application task. Among the desirable properties that such clusters should provide are fault tolerance, reconfigurability, adaptability, and high performance. A truly adaptive distributed system should be able to reconfigure dynamically to respond to the addition and failure of processors, to partitioning and remerging of the network, and to modification and upgrading of the system.

An object-oriented framework is well-suited to building fault-tolerant distributed applications. Objects are distributed across the system, and interact to provide the necessary services of the application. The Common Object Request Broker Architecture (CORBA) standard [14, 15, 18] provides mechanisms for the definitions of interfaces to distributed objects, and the invocation of operations on objects via messages.

The Eternal system is CORBA-compliant middleware supported by a standard operating system. It enhances the CORBA standard by providing a simple and reliable framework for handling fault tolerance and adaptability, rendering these features as transparent to the application as possible.

To provide fault tolerance, objects are replicated across multiple processors within a distributed system. Messages communicate the operations that are to be performed on the objects, as well as the updated versions of the states of the objects after the operations have been performed. If the messages were to be received in different orders by the different replicas, the states of the replicas might become inconsistent. Maintaining consistency of the replicas is difficult in the presence of faults. The Eternal system uses the Totem multicast group communication system [1, 8, 13] to provide reliable totally ordered multicasting of messages, which simplifies the task of maintaining consistency.

2 The Totem System

The Totem system is a suite of group communication protocols that provide reliable totally ordered multicasting of messages within clusters of processors operating in single or multiple local-area networks interconnected by gateways. Each multicast message has a unique message sequence number assigned to it by the originator of the message. These message sequence numbers are used to deliver messages in a single system-wide total order that respects Lamport's causal order [7]. The Totem system also provides membership and topology change services to handle the addition of new and recovered processors and processes, deletion of faulty processors and processes, and network partitioning and remerging.

The virtual synchrony model of Isis [2] orders group membership changes along with the regular messages. It ensures that failures do not result in incomplete delivery of multicast messages or holes in the causal delivery order. It also ensures that, if two processors proceed together from one view of the group membership to the next, then they deliver the same messages in the first view. The extended virtual synchrony model of Totem [12] extends the model of virtual synchrony to systems in which clusters can partition and remerge and in which processors can fail and recover.

*Research supported in part by DARPA grant N00174-95-K-0083 and by Sun Microsystems and Rockwell International Science Center through the State of California Micro Program grants 96-051 and 96-052.

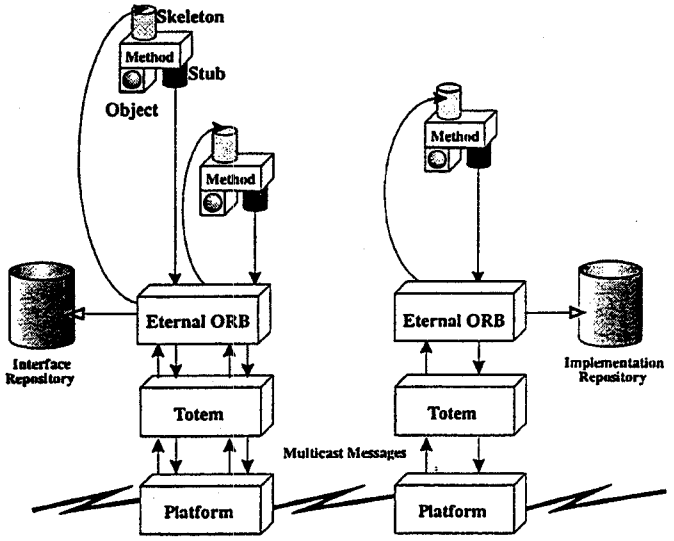

Figure 1: Relationship between Eternal and Totem.

Processors in different components of a partitioned network may deliver the same messages, and yet the order in which they deliver the messages is consistent. It is not easy to provide this guarantee but, with care, it can be done [1, 13], and even efficiently.

Typical applications consist of processes that cooperate or share information to perform a task. The process group layer [8], at the top of the Totem protocol stack, exploits the guarantees of the underlying layers to provide its own reliable ordered delivery guarantees and to maintain dynamic process group memberships. A processor may host multiple processes and multiple process groups, and maintains information about the current membership of the process groups that it supports. Processes can send messages to one or more process groups, of which they may or may not be members. These messages are ordered within each process group, and across all receiving process groups.

The Eternal system exploits the message ordering and membership services of Totem to ensure consistent and efficient replication.

3 The Eternal System

The CORBA standard does not address issues such as fault tolerance and adaptability. Eternal is a CORBA-compliant ORB of a rather specialized nature, extending CORBA to provide these services. In particular, the ORB is extended to describe the degree and type of replication of the objects. The Eternal system assumes responsibility for replicating objects, distributing objects across the system, maintaining the consistency of replicated objects, detecting and recovering from faults, and sustaining operation in all components when a cluster partitions.

Operations on objects are bound to stubs generated from the Interface Definition Language (IDL) specification of the object, as shown in Figure 1. The ORB intercepts this call from the stub, marshals the call and its parameters into a message, and multicasts it to the other processors using Totem. At the site of the object whose services have been requested, a skeleton, also constructed from the IDL specification, invokes the operation and returns the results via the ORB in a similar fashion. Thus, all operations on objects are communicated in messages and are visible to the ORB. The behavior of each object is assumed to be deterministic.

Eternal allows objects to be built hierarchically and compositionally from other objects, in keeping with the spirit of an object-oriented framework. It permits the development of a distributed application as if it were to be run on a single processor. The issues of replication, fault tolerance, and adaptability are transparent to the application.

The types of faults tolerated by Eternal, and the underlying Totem system, are communication faults, including message loss and network partitioning, and processor, process, and object faults. Processors, processes, and objects can crash and recover, and a partitioned network can remerge. Arbitrary faults are not tolerated.

Eternal provides fault tolerance by replicating objects at different sites within the distributed system. The total ordering of Totem multicast messages simplifies replica consistency and interaction in a dynamic framework of operation. In case a cluster partitions, the replicas within a component see the same membership changes at the same points in the message sequence. This is crucial for the consistent remerging of the components upon reestablishment of communication between them.

4 From Process Groups to Object Groups

The extension of process group communication to an object-oriented framework requires the notion of objects, rather than processes, that communicate and cooperate to perform a designated task.

An *object group* [3, 10] is a high-level abstraction for a collection of objects in a distributed object space. An object group may reside entirely on a single processor or may span several processors. Moreover, a processor may host multiple objects and multiple object groups. Just as a process can send messages to a process group, the object group abstraction enables an object to invoke the services of another object group in a transparent fashion so that the invoker of the operation need never be aware of the exact nature, location, membership, degree of replication, or type of replication of the objects. An object only needs to know the interface provided by the group, and invokes the object group as if it were a single object. Moreover, the object group mechanism of Eternal allows object groups to invoke

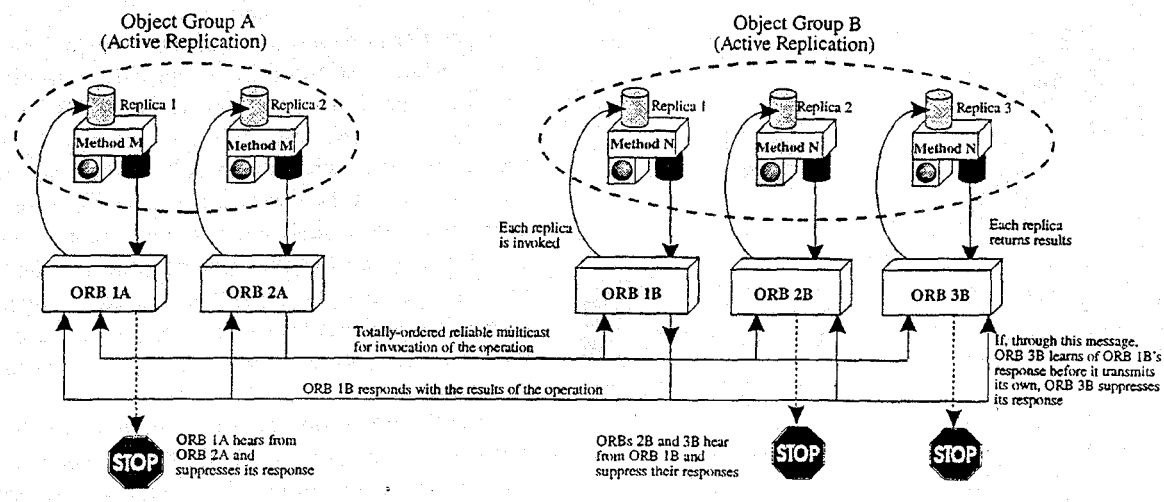

Figure 2: **Active replication.**

each other. Thus, objects in Eternal are fully replicated, *i.e.*, both client and servers objects can be replicated.

In a manner similar to that for process groups, the multicast messages of the Totem system are used to communicate operations to and from object groups. This ensures the reliable ordering of operations within each object group and across different object groups. The object group membership is maintained by the Eternal ORB.

5 Replica Consistency without Partitioning

Fault tolerance in an object-oriented framework is provided by replicating objects, ensuring the consistency of the states of the replicas, and detecting and recovering from faults. The *state* of an object replica is simply the current values of its data structures. The replicas of an object form a homogeneous object group whose purpose is to provide highly available, reliable, and efficient service.

Operations on replicas must maintain the consistency of the states of the replicas. Two approaches for achieving this are *active replication* and *passive replication*. The Eternal system provides mechanisms for nested operations under both replication schemes.

5.1 Active Replication

In active replication, as shown in Figure 2, all of the replicas of an object are considered to be participants in the operation. Here ORB 1A (associated with replica 1 of object group A) and ORB 2A (associated with replica 2 of object group A) communicate their invocations to ORBs 1B, 2B and 3B. Every replica of an object is required to execute the same operations in the same sequence, maintaining replica consistency.

The underlying totally ordered multicast mechanisms guarantee that all of the replicas of an object receive the same messages in the same order, and perform the operations in the same order. This ensures that the states of the replicas are consistent at the end of an operation.

For every operation invoked on a homogeneous object group, a multicast message is required to initiate the operation at each replica. Moreover, the same operation must be performed at each of the sites where the replicas are located. This can lead to increased usage of network bandwidth since each replica may generate further multicast messages. Eternal possesses mechanisms, based on message and operation identifiers, as described in Section 6.1, that detect and suppress duplicate invocations and responses, preventing inconsistencies that might otherwise arise.

Active replication also incurs the increased computational cost of performing the same operation at each of the replicas. The cost of using active replication is also dictated by application-dependent issues, such as the degree of replication and the depth of the nesting of operations. Clearly, active replication is favored if the cost of multicast messages and the cost of replicated operations is lower than the cost of transmitting the object's state to every replica at the end of the operation.

5.2 Passive Replication

In passive replication, as shown in Figure 3, each object is replicated but only a single replica, designated the *primary replica*, performs all of the operations requested. Here ORB 2A communicates its invocations to ORBs 1B, 2B and 3B. Only ORB 1B invokes its replica of object B; the other two ORBs for object B retain the message for use in the event of the failure of the primary replica. Once the primary replica completes the operation, it transfers its updated state to the secondary replicas. During the operation, the states of the secondary replicas may differ from that of the primary

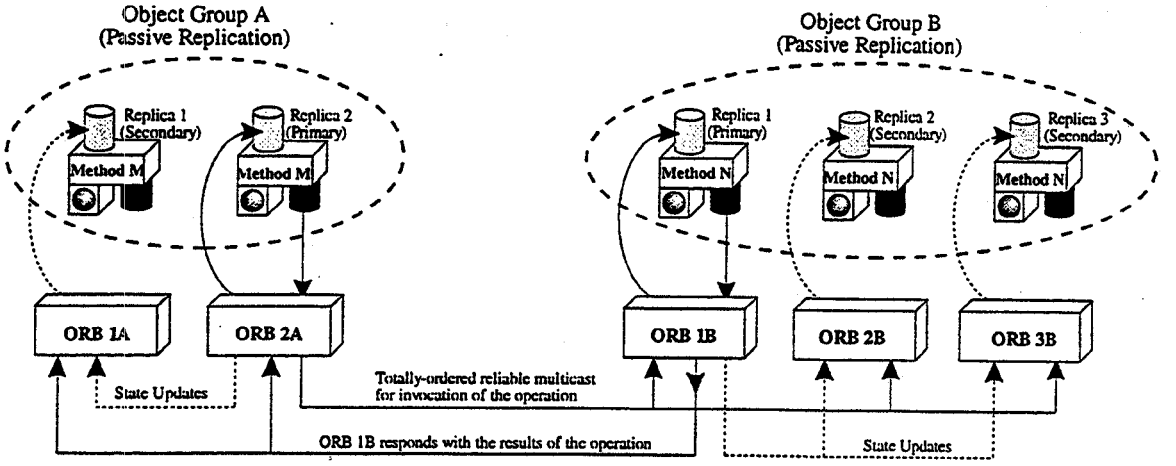

Figure 3: **Passive replication.**

replica; however, the update operations maintain replica consistency at the end of the operation.

A disadvantage of passive replication is that if the state of the object is large, transfer of the state from the primary replica to the secondary replicas can be quite expensive. An advantage is that it does not require the operation to be performed on each of the replicas. If the operation is computationally expensive, the cost of passive replication can be lower than that of active replication. Moreover, since only the primary replica responds with the results of the operation, passive replication may require fewer multicast messages.

5.3 State Transfer Mechanisms

State transfer mechanisms are required for both active and passive replication when a new replica is installed. They are also required in passive replication to update the states of the secondary replicas.

The simplest mechanism for the transfer of large states between replicas of an object is to suspend operations on the object, transfer the state, and then resume operations on the object. This solution is appropriate when the state is not too large and can be transferred quickly. A drawback of this scheme is the need to stop all operations on the object until the state transfer is accomplished. More refined, though more complex, schemes [11] allow operations to be performed on the object while a large state is being transferred. Such a scheme is described below.

For active replication, one of the replicas is designated to perform the transfer. This replica does not stop processing further operations while transferring the state. Rather, it logs a *preimage* (the values of the updated parts of the state before the update) of each update that it performs. First the existing state is transferred, and then the preimages are transferred. The state transferred to the new replica may be inconsistent, since it may have been partially updated, but the new replica can reconstruct the state by applying the preimages. During the transfer, the new replica performs no operations, but rather logs all of the operations. Once the state transfer is completed, the new replica processes the operations it has logged in order to bring its state into consistency with that of the other replicas.

For passive replication, the procedure is similar, except that postimages (the values of the updated parts of the state after the update) are logged and transferred, instead of preimages, and the new or secondary replicas do not log and process operations.

The advantages of this more complex scheme is that a replica does not have to stop processing its messages while transferring its state. However, extra load is imposed on the replica since it continues processing and transfers state simultaneously.

6 Interactions between Object Groups

Object groups serve as a useful abstraction for replication of distributed objects. The replicas within an object group might implement either active or passive replication, though this is transparent from outside the group. Group transparency implies that actively replicated objects and passively replicated objects must be invoked from outside the group in exactly the same manner, although these invocations are handled differently within the group in each case.

The most interesting intergroup interaction occurs between an actively replicated object and a passively replicated object, as shown in Figure 4. Here the replicas in object group A are only aware of addressing object group B as a whole, and never the individual replicas in object group B. Similarly, the replicas in object group B are only aware of responding to object group A as a single entity, and never the individual replicas in object group A. The Eternal ORB

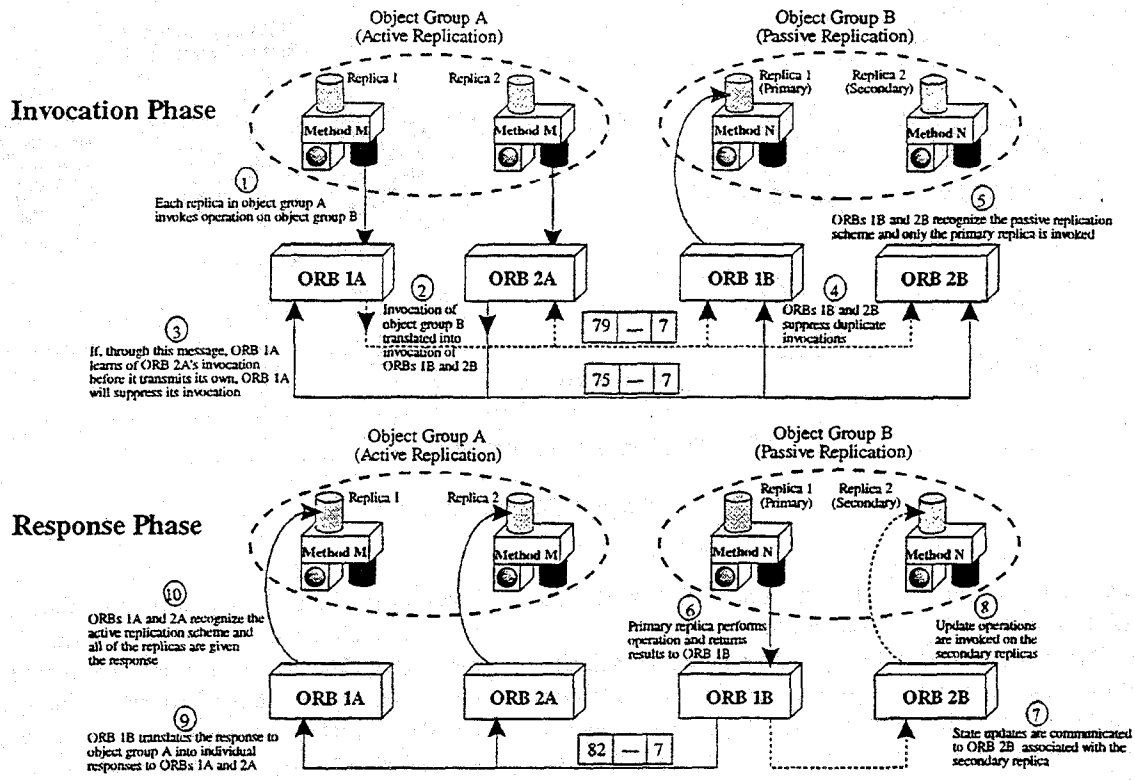

Figure 4: **Interactions between object groups with different replication schemes.**

translates object group invocations into individual replica invocations.

6.1 Operation Identifiers

In addition to the information that CORBA packages with an invocation, Eternal supplies unique operation identifiers that simplify the detection and suppression of duplicate invocations and responses. Consider, for example, an operation 1 on one object that invokes an operation 2 on another object. Operation 1 is invoked by a multicast message that is given a unique sequence number by Totem. Operation 1 may invoke a sequence of operations, one of which is operation 2; each such operation is given a unique sequence number by the ORB.

Figure 5 shows the invocation identifier for operation 1 invoking operation 2. The first field of the invocation identifier contains the sequence number of the message that invokes operation 2. This field may be different for redundant invocations. The second and third fields of the invocation identifier constitute the operation identifier, which is identical for duplicate invocations and is unique to the operation. The second field contains the sequence number of the message that invoked operation 1. The third field contains the sequence number assigned to operation 2 by the ORB.

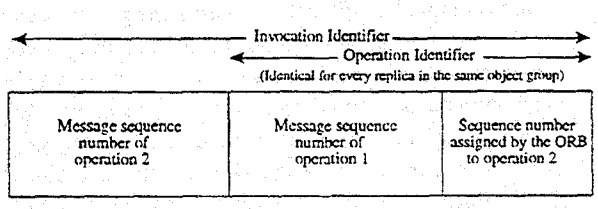

Figure 5: **Identifier for operation 1 invoking operation 2.**

The response identifier for operation 2 is tagged with the same operation identifier contained in the second and third fields as the invocation identifier, but the first field now contains the sequence number of the message containing the results of operation 2. Several examples are shown in Figure 6.

6.2 Suppression of Duplicate Operations

Duplicate invocations of an object are suppressed when active replicas of the object invoke an operation, as shown in Figure 4. This is done by ORBs 1A and 2A multicasting their invocations to each other, as well as to object group B. If either of the sender ORBs 1A or 2A receives the other's invocation before transmitting its own, that sender ORB suppresses its invocation. Here, ORB 1A's invocation is suppressed by the message from ORB 2A.

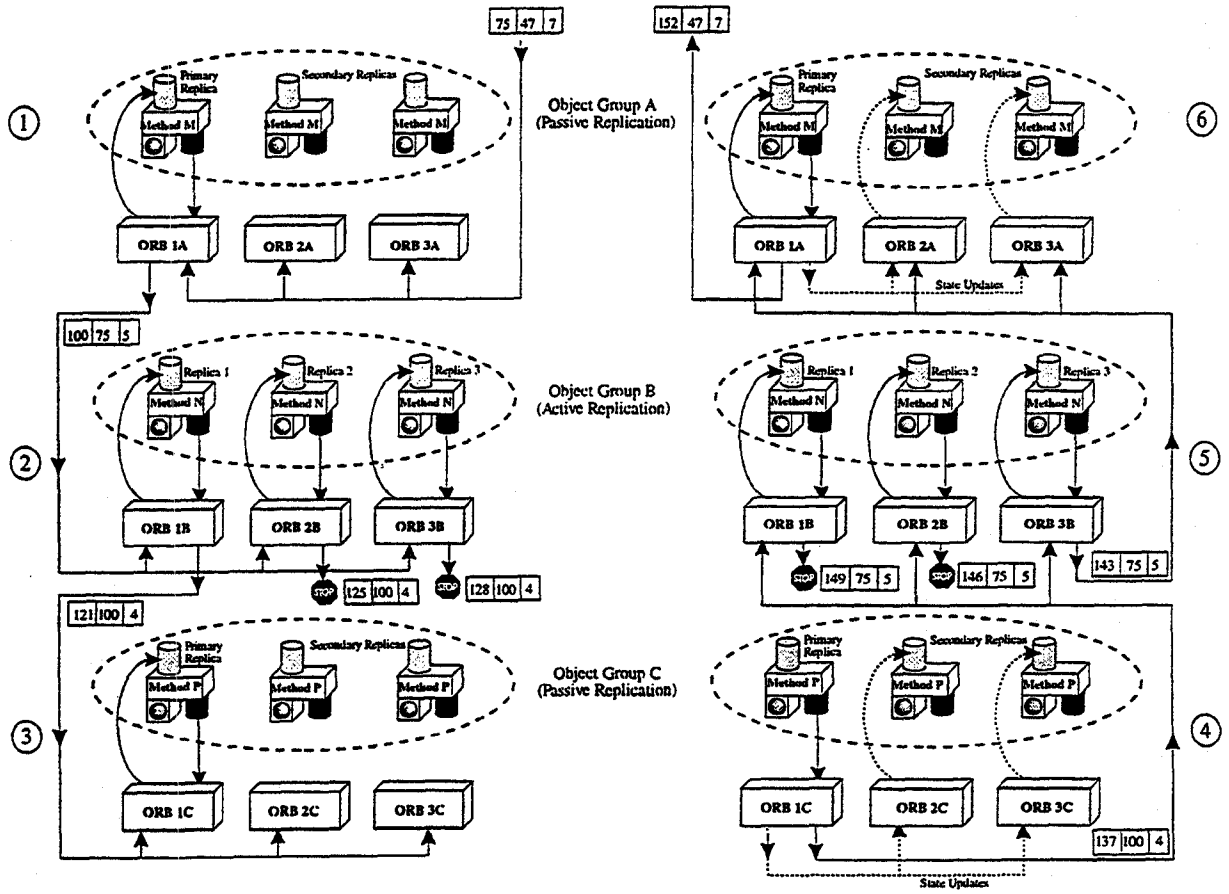

Figure 6: **Nested operations with mixed replication schemes.**

Even if these duplicate invocations are not suppressed at the sender ORBs, duplicate invocations at the receiving ORBs 1B and 2B are suppressed because their operation identifiers are equal. Thus, no operation is ever performed more than once, ensuring that the states of the objects are never corrupted by duplicate invocations.

Similarly, duplicate responses are suppressed when active replicas of an object respond to an operation invoked on them. Several examples are shown in Figure 6.

6.3 Nested Operations

Eternal also addresses the issue of nested operations. By a *nested operation*, we mean an operation that results in the invocation of yet another operation or, in the terminology of Eternal, the invocation of one object group leading to the invocation of another object group. Challenging problems arise when the chain of invocations involves replicas with different replication schemes, as shown in Figure 6.

For the active replicas, the figure indicates the suppression of duplicate invocations and duplicate responses. For the passive replicas, the figure indicates the state update operations that must be performed on the secondary replicas within the object group before the primary responds to the next object group in the chain.

The difficulties in performing a nested operation are that duplicate invocations and responses must be suppressed and that the states of all of the replicas of an object must be consistent after the entire operation, even in the presence of faults.

6.3.1 Failure of an Active Replica. For an actively replicated object, fault recovery is relatively simple. Totem's reliable multicast mechanisms ensure that a requested operation is known either to all of the remaining replicas of the object or to none of them. Consequently, the operation will be performed on all of the remaining replicas or on none of them.

If an active replica fails while performing the operation, the remaining active replicas in its object group continue to perform the operation and return the result. The failure is thereby rendered transparent to the other object groups involved in the nested operation. This replication scheme yields substantially more rapid recovery from faults.

6.3.2 Failure of a Passive Replica. For a passively replicated object, the effect of the failure of a replica depends on whether the failed replica is a primary or a secondary. If a secondary replica fails, it is simply removed from the group by the object group membership mechanism while the operation continues to be performed. Thus, the failure of a secondary replica is transparent to the other object groups in the nested operation.

Consider object group A in Figure 6. Before the operation is invoked on the primary replica via ORB 1A, both ORBs 2A and 3A receive the operation, but do not invoke it since they are associated with the secondary replicas. If the primary fails after invoking a nested operation on object group B, the object group membership mechanism is invoked and a new primary is determined. The new primary reinvokes the nested operation. The reinvocation carries the same operation identifier in the second and third fields, but a different message sequence number in the first field. Consequently, if the ORBs associated with object group B have already received this invocation, they will disregard it, but will transmit the response, even if it has already been sent.

Responses may be generated by both the original and new primary replicas. Such duplicate responses will be suppressed, since their operation identifiers are equal.

7 Replica Consistency with Partitioning

A distributed system of processors may partition into a finite number of *components*. The objects within any component can communicate with each other, while objects in different components are unable do so. In Eternal, all of the components of a partitioned system remain operational, and objects in all of the components continue to perform operations.

The underlying Totem system guarantees that all of the objects see a consistent global total order of operations, even if those objects are in different components; however, some of the operations may not be visible to some of the objects. The real problem arises when different components of a partitioned system merge to form a larger component. Each component may contain a subset of the replicas of an object, and different operations may be performed on the replicas in different components, leading to inconsistencies that must be resolved when the components remerge.

In Eternal, for each replicated object, at most one *primary component* is identified when an object group membership change occurs. Each of the other components is then a *secondary component* for that object. A component may be the primary component for one replicated object and a secondary component for a different replicated object.

Eternal supports *state transfer mechanisms* and *fulfillment operations* to restore consistency of replicas following remerging of the partitioned system. The state transfer mechanism transfers the state of the replicas in the primary component to those in the secondary components, while the fulfillment operations permit operations performed in a secondary component also to be performed in the merged component.

7.1 State Transfer Mechanisms

When the components of a partitioned system remerge, the replicas in the primary component must communicate their states to the replicas in the secondary. The mechanisms for the transfer of state between components are similar to those outlined in Section 5.3. Depending on whether the replicas involved are active or passive, the state transfer mechanisms use preimages or postimages, respectively. This ensures that all of the replicas of an object have consistent states; however, the operations performed in the secondary components are not yet reflected in that state.

7.2 Fulfillment Operations

If the operations in different components of a partitioned system are not disjoint, additional mechanisms are required to reconcile the states of the objects once communication is restored. These mechanisms must address the case in which both the primary and secondary components have performed operations on different replicas of the same object without being able to communicate with each other.

In Eternal, the replicas in a secondary component can continue to perform state updates while the system is partitioned, substantially as they would during normal unpartitioned operation. As the updates are performed on the replicas in the secondary component, they generate fulfillment operations. A queue of fulfillment operations is formed in the secondary component for each object whose replicas (in the secondary component) perform updates while the system is partitioned.

Once the state transfer phase is completed, the states of all of the replicas in the system are identical. The updates that were recorded as fulfillment operations by the previously disconnected secondary component are now applied to all of the replicas in the larger merged component. Normal operations continue to be applied in the merged component during the state transfer and during the application of fulfillment operations. The fulfillment operations ensure that the operations performed in a secondary component are performed in the merged component and that problems requiring manual resolution are reported. The fulfillment operations are, of course, application-specific but they are just operations; they require no special programming skills and they are applied only to the state of the replicas in the merged component.

Consider now a partitioned system consisting of two components, as shown in Figure 7. Some of the replicas of an object may no longer be able to communicate with the other replicas. Nevertheless, all of the replicas of an object,

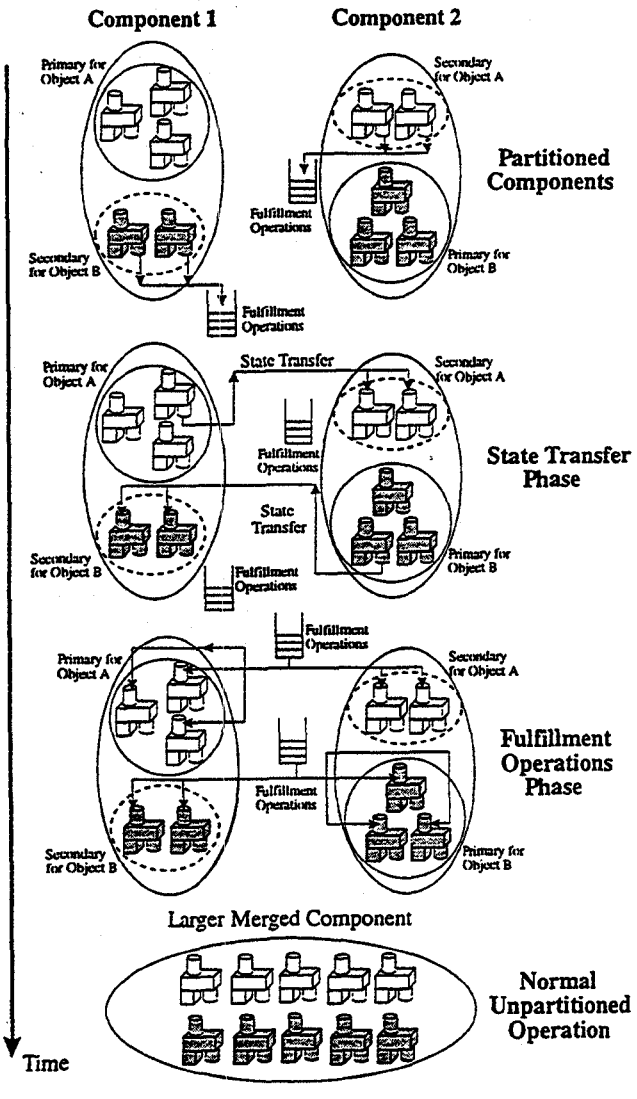

Figure 7: **Partitioning and remerging of object groups.**

both in the primary and secondary components, continue to perform operations and to have operations performed on them. Component 1 is the primary for object A and the secondary for object B, while component 2 is the primary for object B and the secondary for object A.

When communication between the components is restored, state transfers from the primaries for object A and object B to their respective secondaries ensure a consistent state of the objects after merging. However, if both the secondaries for object A and object B have updated their replicas while the system is partitioned, the effects of these updates are not reflected in the consistent state. Fortunately, such updates of object A and object B in the secondary components have generated fulfillment operations, which can now be applied to the consistent state of all the replicas of the respective objects in the merged component.

8 Example

Consider, for example, a system that coordinates the sale and manufacture of automobiles, as illustrated in Figure 8. An automobile sales showroom that sells an automobile updates the inventory object by decrementing the total number of automobiles that are available. This operation results in a shipping order being sent to the factory, and in subsequent shipment of the automobile to the customer. The automobile factory, for each new automobile that it manufactures, updates the inventory object by incrementing the total number of automobiles available.

The replicas in this system are the copies of the inventory object located at the factory and the two showrooms. Under normal unpartitioned operation, these contain identical information about the total number of automobiles available for purchase by customers. Every sale at the showroom, and every new automobile manufactured, results in an update operation on each of the three active replicas of the inventory object. Moreover, the factory and the showrooms update the inventory objects transparently, without being aware of the actual membership of the object group (consisting of the three replicas) or the location of all the replicas.

If one of the sales showrooms loses communication with the factory and the other showroom, the system becomes partitioned. The sales at the disconnected showroom, which forms the secondary component, will continue to update the replica at the showroom. The manufacture of cars at the factory and the sales at the showroom connected to the factory will update the replicas in that component, which forms the primary component.

The algorithm for performing updates in the two components is shown in Figure 8. The operations on the replica in the disconnected showroom are queued up as fulfillment operations which await reestablishment of communication with the other component of the partitioned system. The replica in the disconnected showroom component now has a state that differs from those of the two replicas in the primary component.

If communication is restored, the replicas in the primary component first transfer state to those in the secondary component. At the end of this phase, all of the replicas have the updated inventory of the replicas in the primary component. However, this is not sufficient since the operations performed on the replica in the secondary component are not reflected in the state. The fulfillment operations must thus be performed on the updated state of the replicas.

The fulfillment operations may need to handle special application-specific conditions. For example, if the showroom in the secondary component has sold a car that has also been sold by the showroom in the primary component, it will be necessary to generate a back order notification and a special rush manufacturing order to the factory to accelerate production.

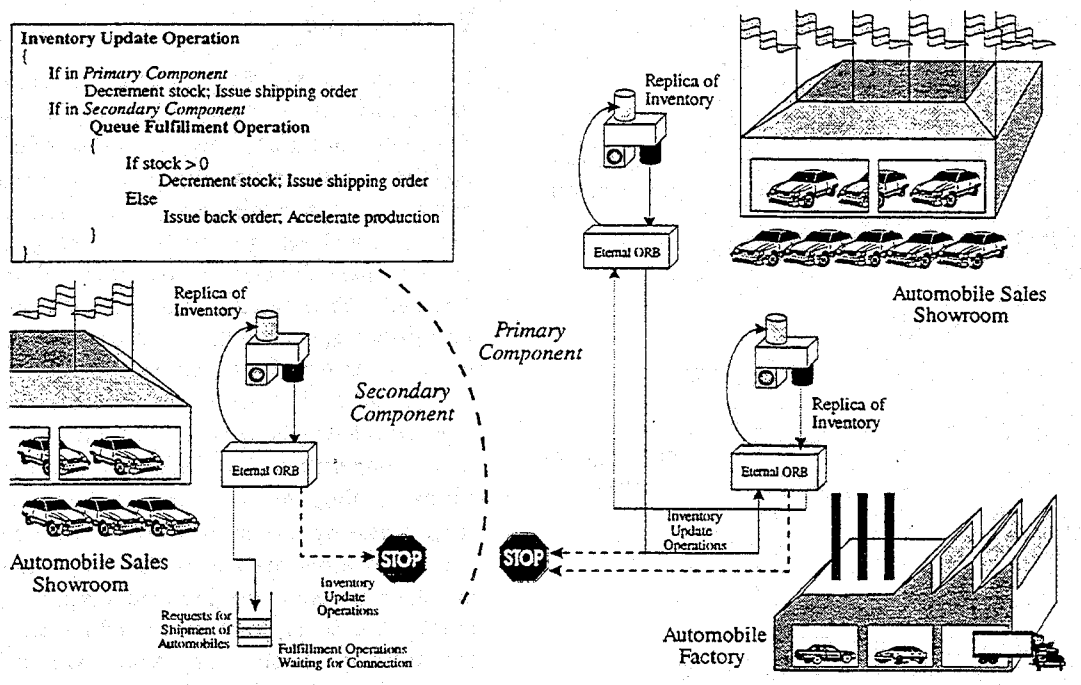

Figure 8: **Partitioning of one automobile sales showroom from the other automobile sales showroom and the automobile factory.**

9 Related Work

The Common Object Request Broker Architecture (CORBA) [14, 15, 18] standard established by the Object Management Group (OMG) is essential to our work, though the Eternal system goes beyond CORBA. CORBA provides support for location transparency, separation between implementation and specification, and the interworking of heterogeneous computer systems, all of which are essential to Eternal.

In [16] Sturman and Agha provide a reflective architecture for CORBA that, like Eternal, supports fault tolerance. The mechanisms underlying the Actor model are integrated with the CORBA model to yield a methodology for customizing fault tolerance without modification of the object request broker.

The Electra Toolkit [10] defines the concept of object groups in a CORBA-compliant ORB. The Electra ORB is built over the Horus system [17] and provides for fault tolerance, replication, and group communication in an object-oriented framework. Electra is restricted to a non-hierarchical object system and provides no mechanisms for dealing with partitionable systems since it is based on virtual synchrony instead of extended virtual synchrony.

Unlike the above systems, which are based on CORBA, several other distributed object systems have been built that do not utilize the facilities of CORBA. The Phoenix system [3] supports an object-oriented interface that provides group communication, group management, and thread management based on the concept of view synchrony to provide a toolkit for the development of large-scale fault-tolerant distributed systems.

The GARF system [4] is an object-oriented platform that supports the programming of fault-tolerant distributed applications. The Dependable Object Toolkit of GARF, built on top of Isis [2], provides object group management, with Smalltalk as the application programmer interface. Like Eternal, GARF allows nested operations when both client and server objects are replicated, and provides extensive support for active replication.

The Arjuna system [9] provides paradigms for the replication and management of object replicas and uses an atomic transaction strategy to provide fault tolerance. Support is provided for active replication, coordinator-cohort passive replication, and single copy passive replication. Strategies similar to checkpointing are used for state updates in passive replication.

The work of Higaki and Soneoka [5] provides insight into the problems associated with active and passive replication, and addresses the reduction of multicast overhead in active replication schemes.

To the best of our knowledge, none of the prior work deals with the issues of nested operations and intergroup interactions under different replication schemes. Moreover, the other systems have not addressed the challenging problems of sustaining operation in all components of a partitioned system and of recovery when the components of the system remerge.

10 Conclusion

The Eternal system is currently under construction, using the CORBA-compliant Inter-Language Unification (ILU) [6] from Xerox Palo Alto Research Center. Our focus in this paper has been on object replication, but many other aspects of CORBA systems, such as concurrency, remain to be addressed. We will be better able to assess the effectiveness and performance of our approach to object replication when our implementation is completed.

We are also currently investigating the use of replicated objects to achieve more than just fault tolerance. The ability to mask the failure of an object or processor can also be used to mask the deliberate removal of an object or processor and its replacement by an upgraded object or processor. For a processor, the replacement can be a different type of processor. It is also possible, in several steps, to replace an object by another object with a different interface specification, without stopping the system and without requiring great system programming skill from the application developer. Over time, both hardware and software components of the system can be replaced and upgraded without interrupting the service provided by the system. Thus, our objective is a system that can run forever, a system that is Eternal.

Acknowledgment

We wish to thank the anonymous referees for their constructive comments, which have greatly improved this paper.

References

[1] D. A. Agarwal, *Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks*, Ph.D. Dissertation, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[2] K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, IEEE Computer Society Press, Los Alamitos, CA (1994).

[3] P. Felber and R. Guerraoui, "Programming with object groups in PHOENIX," Broadcast Technical Report 93, Esprit Basic Research Project 6360, Ecole Polytechnique Federale de Lausanne, 1995.

[4] B. Garbinato, R. Guerraoui and K. R. Mazouni, "Implementation of the GARF replicated objects platform," *Distributed Systems Engineering Journal 2*, 1 (March 1995), pp. 14-27.

[5] H. Higaki and T. Soneoka, "Fault-tolerant object by group-to-group communications in distributed systems," *Proceedings of the Second International Workshop on Responsive Computer Systems*, Saitama, Japan (October 1992), pp. 62-71.

[6] B. Janssen, D. Severson, and M. Spreitzer, ILU 1.8 Reference Manual, Xerox Corporation (May 1995), ftp://ftp.parc.xerox.com/pub/ilu/ilu.html.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM* 21, 7 (July 1978), pp. 558-565.

[8] C. A. Lingley-Papadopoulos, *The Totem Process Group Membership and Interface*, M.S. Thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara (August 1994).

[9] M. C. Little and S. K. Shrivastava, "Object replication in Arjuna," Broadcast Technical Report 93, Esprit Basic Research Project 6360, University of Newcastle Upon Tyne, 1994.

[10] S. Maffeis, "Adding group communication and fault tolerance to CORBA," *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, Monterey, CA (June 1995), pp. 135-146.

[11] P. M. Melliar-Smith and L. E. Moser "Simplifying the development of fault-tolerant distributed applications," *Proceedings of the Workshop on Parallel/Distributed Platforms in Industrial Products, 7th IEEE Symposium on Parallel and Distributed Processing*, San Antonio, TX (October 1995).

[12] L. E. Moser, Y. Amir, P. M. Melliar-Smith and D. A. Agarwal, "Extended virtual synchrony," *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, Poznan, Poland (June 1994), pp. 56-65.

[13] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM* 39, 4 (April 1996), pp. 54-63.

[14] Object Management Group, *The Common Object Request Broker: Architecture and Specification* (1995), Revision 2.0.

[15] R. M. Soley, *Object Management Architecture Guide*, Object Management Group, OMG Document 92-11-1.

[16] D. C. Sturman and G. Agha, "Extending CORBA to customize fault tolerance," Technical Report, Department of Computer Science, University of Illinois (1996).

[17] R. van Renesse, K. P. Birman and S. Maffeis, "Horus: A flexible group communication system," *Communications of the ACM 39*, 4 (April 1996), pp. 76-83.

[18] S. Vinoski, "Distributed object computing with CORBA," *C++ Report 5*, 6 (July/August 1993), pp. 32-38.