The Eternal System:
An Architecture for Enterprise Applications

L. E. Moser, P. M. Melliar-Smith
P. Narasimhan, L. A. Tewksbury and V. Kalogeraki
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract
The Eternal system supports networked enterprise applications that must operate continuously 24 hours per day, 7 days per week. Based on the CORBA standard, Eternal provides object replication not only for fault tolerance but also for live software upgrades, as well as resource management facilities. Through the use of interceptors, Eternal renders the object replication transparent to the application, as well as to the ORB and to the operating system. Thus, Eternal works with commercial off-the-shelf CORBA ORBs and standard unmodified operating systems. Eternal handles the difficult issues of object replication, fault tolerance, live upgrades and resource management, thereby allowing the application programmers to focus on the applications.

1 Introduction
Future enterprise applications will be highly networked computer systems, will interact directly with other enterprise applications, and will operate continuously 24 hours per day, 7 days per week. They will be even more critical for the company, business or organization than they are today. Some applications, such as hospital administration, airport operation, public safety systems, manufacturing plants and retail supermarkets, must operate continuously because they provide vital local services. Other applications, such as e-commerce servers, supply chain networks and banking systems, must operate continuously because their business is global; such applications must be able to interact with customers and suppliers, and their computer systems, in other time zones.

Eternal will no longer be viewed within the traditional client/server model, with external clients and individual isolated servers. Consider a local retail supermarket checkout scanner that operates 24 hours per day, 7 days per week. Information must be communicated by the scanner to banks for credit or debit authorizations and transactions, and also to the warehouse for restocking, to the manufacturers or distributors of products and to their transportation agents for the purchase and delivery of supplies, to market research and advertising agencies, and to government taxation and welfare departments. Each manufacturer or distributor in the supply chain must itself respond to such information, based on its own production planning and scheduling, perhaps after further communication with its own suppliers.

Enterprise applications require a high degree of fault tolerance to provide the level of service that the customers, suppliers and partners have come to expect, and to reduce the risk of losing business, and goodwill and trust, in the event of a fault. Fault tolerance requires replication of information, so that the information is not lost or damaged when a fault occurs. Such replicated information must be kept consistent, even in the presence of faults, consistent between the several replicas maintained by a single organization, and consistent between copies of that information maintained by different organizations. Other important considerations are the time and difficulty required to repair damage and recover lost information, to “catch up” with what occurred while the system was down, and to restore a consistent state during system recovery. The distributed nature of enterprise applications that span multiple organizations over the Internet makes recovery difficult. Indeed, during recovery from a fault, it may be impossible to obtain information from another organization’s computer system.

Recent well-publicized examples of major computer system failures at Charles Schwab, E*Trade and eBay, with losses of thousands of dollars per minute of down time, demonstrate the need for robust fault tolerance and live upgrade technology. Interestingly enough, in these three examples, the down time was due not only to faults

* This research has been supported by DARPA in conjunction with ONR and AFOSR under Contracts N00174-95-K-0083 and F30602-97-1-0246.
in the computer system but also to fiascos while trying to upgrade the system. In a few years, every major enterprise will be as dependent on e-commerce and e-business as Schwab, E*Trade and eBay are today, and every major enterprise will need the high degree of fault tolerance and live upgrade capabilities that Schwab, E*Trade and eBay currently require.

Unfortunately, existing technologies that provide robust fault tolerance for complex networked applications are expensive to develop and require significant reworking of the enterprise’s infrastructure software to embed fault tolerance. Even if enterprises could afford the cost, there are not enough programmers skilled in fault tolerance to build all of the applications that will require fault tolerance. Moreover, enterprises must be able to deploy their new, modified and upgraded applications rapidly. Enterprises need the ability to take existing applications, programmed by typical application programmers, and to render them fault-tolerant without extensive modifications.

Enterprise applications that must operate continuously present another problem. They need the ability to have their hardware and software upgraded without stopping operation of the system. Live hardware upgrades are supported by several hardware vendors, although they are usually quite highly circumscribed. To our knowledge, no systematic and robust technology currently exists for live software upgrades. Existing practice is ad hoc, requires great skill, and is vulnerable to fiascos.

2 Strategies for Fault Tolerance

In the current state-of-the-art the two principal strategies available to support fault tolerance in enterprise applications are transaction processing and group communication.

2.1 Transaction Processing

Fault-tolerant transaction processing systems are well-understood, are easily programmed, are relatively inexpensive, provide excellent performance, and are readily available from several commercial vendors. Such systems typically employ a single individual server that is either duplicated (mirrored) or has access to duplicated disks within a cluster of computers. If a transaction is completed successfully, either both disks are updated or neither of them is updated. This provides high availability rather than high reliability because, following a fault, all active transactions must be aborted and restarted. No protection against faults is provided for the clients.

The fault-tolerant transaction processing strategy is effective for simple enterprise applications in which the servers operate independently and the clients are humans or are simple application programs for which recovery algorithms are easy to implement. For complex enterprise applications in which multiple servers interact, this approach is ineffective for maintaining consistency of replicated information. It is also inappropriate for applications in which even a momentary disruption of service is unacceptable, or in which both clients and servers must be replicated to be protected against faults.

2.2 Group Communication

Group communication systems have been employed in relatively few applications and are available commercially from relatively few vendors, but are potentially appropriate for a wide range of enterprise applications. In the group communication strategy, the application is structured into processes, which are distributed across multiple computers and which communicate with each other by message passing. The processes are replicated for fault tolerance, and the replicas of a process are grouped together to form a process group. Messages are multicast to the groups of processes for which they are relevant, using hardware multicast mechanisms of local-area networks to achieve high performance. Typical applications contain hundreds or thousands of processes, and tens or hundreds of process groups, distributed over dozens of computers.

The group communication paradigm is quite general and is applicable to many types of enterprise applications. It provides high availability and high reliability for both clients and servers, and is effective for enterprise applications involving multiple interacting servers. Unfortunately, the message-passing group communication paradigm exposes the distribution, replication and fault tolerance to the application programmer. Moreover, the program code required for replica consistency and fault recovery is difficult to develop, and is inextricably intertwined with the program code for the application.

Neither fault-tolerant transaction processing systems nor group communication systems fully address the fault-tolerance problems faced by future enterprise applications. Moreover, no effective solutions exist for live upgrades to operating software. The Eternal system provides effective inexpensive solutions in both of these areas.

3 The Eternal System

The Eternal system is based on the Common Object Request Broker Architecture (CORBA) defined by the Object Management Group (OMG) [16]. CORBA provides modular object-oriented programming, location transparency in distributed systems, portability of programs across platforms, and interoperability between diverse platforms—all of which are essential for future enterprise applications.

---

1 In principle, distributed transaction strategies are available that can safely handle complex interacting enterprise applications. In practice, no enterprise is willing to allow the resources of its own computer systems to be locked by transactions from other enterprises. Consequently, distributed transactions are seldom used between enterprises.
Eternal extends CORBA with capabilities for fault tolerance and live upgrades of application objects. Eternal replicates objects, distributes the object replicas across the system and maintains consistency of the object replicas. Both client and server objects can be replicated, and objects can act as both clients and servers.

In the Eternal system, the replicas of an object form an object group, a collection of objects typically located on different computers. An object can invoke methods on an object group (i.e., simultaneously on all of the members) in a transparent manner so that the invoker of the method is not aware of the nature, location, membership, degree of replication, or type of replication employed. A client object invokes the methods on a server object group as though the server were a single object.

With the Eternal system, an application programmer can write an enterprise application as a conventional CORBA program. The application programmer does not need to worry about the difficult issues of object replication, fault tolerance, live upgrades and resource management. By handling these difficult issues, Eternal frees up the application programmer to focus on the application.

Thus, Eternal provides:

- Simpler programming of fault-tolerant distributed enterprise applications
- Robust and consistent replication of objects across the distributed system
- Resource management to control the allocation of the replicas of the objects to the processors
- Continued operation, even in the presence of faults, without anomalies or inconsistencies
- Upgrade and evolution of software objects while the system is live.

The Eternal system consists of the Interceptor, Replication Manager, Resource Manager and Evolution Manager, as shown in Figure 1. The CORBA Object Request Broker (ORB) packages the method invocations and responses into messages formatted according to the Internet InterORB Protocol (IIOP), which are transmitted via TCP/IP. The Interceptor [13] transparently captures the IIOP messages, and diverts them to the Replication Manager, which in turn passes the messages to a group communication system that multicasts them to the object groups containing the object replicas.

The Replication Manager maintains the consistency of the states of the replicas of an object, and provides fault detection and recovery. It transparently translates method invocations on an object group into invocations on the individual object replicas. The Resource Manager allocates the replicas of the objects to the processors to satisfy fault tolerance and load balancing objectives.

The Evolution Manager performs live upgrades of objects, based on the IDL definitions of the objects in the
Interface Repository. The Evolution Manager invokes the Resource Manager to create replicas of new versions of objects and to remove replicas of old versions, while maintaining consistent operation.

The Resource Manager and the Evolution Manager are themselves implemented as CORBA objects and, consequently, can benefit from CORBA’s interoperability and Eternal’s fault tolerance and live upgrades.

4 Interception

The Eternal Interceptor monitors the operating system calls made by the objects to establish IIOP connections over TCP/IP, and to communicate IIOP messages over those connections. The Interceptor catches the IIOP messages before they reach TCP/IP, and diverts them instead to the Replication Manager. The Replication Manager multicasts the messages to the object groups using a group communication system that provides reliable, totally ordered message delivery. The current implementation of Eternal uses the Totem group communication system [11], but any group communication system that provides guarantees similar to those of Totem can be used instead.

The interception approach of Eternal requires no modifications to the ORB, the operating system, the application or the CORBA standard. The current prototype of Eternal operates with seven unmodified commercial CORBA ORBs and exhibits excellent performance. Fault-tolerant applications with replicated client and server objects experience no more than 10% increase in invocation and response times, compared to unreplicated objects using unmodified commercial ORBs.

5 Replication Management

The Eternal Replication Manager supports two types of object replication, active replication and passive replication, as shown in Figure 2.

In active replication, when a client object group invokes a method on a server object group, Eternal multicasts the invocation to the server object group via Totem, and each of the server replicas then executes the method and returns the response.

In passive replication, when a client object group invokes a method on a server object group, Eternal multicasts the invocation to the server object group via Totem, and a single designated server replica, known as the primary replica, executes the method. The Replication Manager at each nonprimary (backup) replica logs the message containing the invocation so that one of those replicas can invoke the method, if the primary replica fails. At the end of one or more invocation/response cycles, the Replication Manager transfers the updated state of the primary replica to the nonprimary replicas. This state transfer is transparent to the client that invoked the method because it is internal to the server object group.

The type of replication (active or passive) depends on the characteristics of the application and is determined for each application object individually. Both types of replication can be employed within the same application.

5.1 Fault Detection and Recovery

Eternal and Totem detect faults using fault detectors based on timeouts. On each processor, a fault detector periodically pings each of the objects on that processor, while
both Totem and a distributed fault-tolerant fault detector similarly monitor each processor.

For active replication, if any one of the replicas fails, or is removed for upgrading during the execution of a method, service is not interrupted (because the method is executed by the other replicas) and the results are returned to the invoking object. On recovery after a fault, Eternal transfers the state of an existing replica to a new or recovered replica.

For passive replication, the effect of a fault depends on whether the faulty replica is a primary replica or a nonprimary replica. If a nonprimary replica fails during the execution of a method, the object group membership mechanisms remove it from the group, while the primary replica continues to execute the method.

If the primary replica fails, the object group membership mechanisms of Eternal detect the fault and determine a new primary replica. The new primary replica retrieves the most recent state, recorded in a log by the prior primary replica, and restores its state. It then retrieves, from the log, the sequence of request messages that invoked methods of the prior primary replica, and repeats those invocations. Further invocations of other objects, as well as responses, may be generated by both the original and the new primary replicas. The Replication Manager detects and suppresses such duplicate invocations and duplicate responses, as described in Section 5.3.

5.2 Replica Consistency

To ensure that the replicas of an object are updated consistently, Eternal exploits the reliable, totally ordered message delivery of the underlying multicast group communication protocol. The invocations of methods on the object replicas, and the corresponding responses, are contained in multicast messages. Because the messages are delivered in the same order to each of the object replicas, the methods are executed by the replicas in the same order, thereby maintaining consistency of the states of the replicas.

For active replication, this ensures that a method invocation, and the corresponding response, is received by all of the replicas of an object or by none of them. Consequently, the method is executed by all of the replicas or by none of them.

For passive replication, this ensures that either all of the nonprimary replicas have the updated state of the primary or, alternatively, all of the nonprimary replicas know that none of them has the updated state.

However, a reliable totally ordered message delivery protocol alone does not suffice to maintain the consistency of the states of the replicas. The behavior of the replicas must be deterministic. In particular, Eternal restricts invocations so that only a single thread is active within an object, eliminating nondeterminism arising from concurrency [14]. In addition, Eternal detects and suppresses duplicate invocations and duplicate responses, and transfers state between the replicas, as described below. More details can be found in [12].

5.3 Duplicate Detection and Suppression

To detect duplicate invocations and duplicate responses, the Replication Manager attaches a unique operation identifier to each message conveying a method invocation or response. Duplicate invocations (responses) by the replicas in a client (server) object group have the same operation identifier and can be detected and suppressed.

Figure 2 shows an actively replicated client (object group A) invoking a method on a passively replicated server (object group B). When Eternal detects a duplicate invocation by a client replica in object group A, the Replication Manager suppresses the invocation from that replica.

For passive replication, suppression of duplicate invocations (responses) is not required during normal operation, but may be required during recovery from a fault. Thus, when Eternal detects a duplicate response from a server replica in object group B on recovery from a fault, the Replication Manager suppresses that replica’s response.

In case the duplicate invocations (responses) are not suppressed at the client (server) due to the asynchrony in the system, they will be detected and suppressed at the server (client), because the associated operation identifiers of the duplicate operations are identical to those of the original invocations (responses).

5.4 State Transfer

The Replication Manager uses a state transfer mechanism to update the states of the nonprimary passive replicas after one or more methods have been executed by the primary replica, as shown in Figure 2. The state transfer mechanisms are used to initialize both new and recovered replicas with the current state, to initialize modified replicas introduced into a system during an upgrade, and to migrate objects from one computer to another in order to balance the loads.

State transfer is performed using two methods, getstate() and setstate(). These methods can be programmed by hand or they can be synthesized by a preprocessor that employs pickling [2] to externalize the state for transmission across the network, and to internalize the state at the destination by reconstructing the state of the object.

A simple scheme for state transfer between the replicas of an object suspends the execution of the object, transmits its state, and then resumes execution of the object. This simple scheme is appropriate when the state is relatively small and can be transferred quickly. A disadvantage of this scheme for large states is that execution of the object is stopped until the state transfer is completed. More refined, though more complex, schemes [9] allow an object to execute while a large state is being transferred.
6 Resource Management

Typical enterprise applications are complex and incompletely understood, making it difficult for the application programmer to obtain accurate projections of resource requirements and behavioral characteristics. Consequently, resource management in Eternal [6, 10] is implemented as a Resource Manager for the system and a Profiler for each processor, as shown in Figure 3.

The Eternal Resource Manager is implemented as a set of CORBA objects and may itself be replicated. Logically, however, there is only a single copy of the Resource Manager. Each Profiler, however, is specific to an individual processor and is interfaced to both the CORBA ORB and the operating system.

6.1 Information Base

Initially, a configuration file is used to describe the physical configuration, including the processors and the network along with their specific characteristics (processing speed, memory size, network bandwidth, etc). Periodically, each Profiler provides the Resource Manager with updated information on the current load (processing load, available memory, communication load, etc).

For each application task, the Resource Manager maintains a list of the method invocations required for that task, a deadline for completion of the task, and an importance metric that is used to decide which tasks should be abandoned in the event of a system overload.

For each method of each object, the Resource Manager maintains estimates of the processing and communication times for invocations of that method. These estimates are used to determine the initial laxity when an application task starts to execute.

6.2 Resource Manager

The Resource Manager allocates the object replicas to the processors based on the current loads on the processors, and moves objects from one processor to another. As new tasks are introduced into the system, the Resource Manager decides whether the available resources can satisfy the requests and allocates resources to them accordingly. During operation, the Resource Manager might determine that a resource is overloaded or that a task is not meeting its deadlines, necessitating reallocation of the object replicas. If a processor is lost because of a fault, the Resource Manager might need to reallocate the object replicas to maintain a sufficient degree of replication to satisfy fault tolerance requirements.

The Resource Manager uses a least-laxity scheduler to schedule the invocations of the objects on the processors. The least laxity of a task represents a measure of urgency of the task and is defined as the difference of the deadline by which the task must complete and the sum of the current time and the estimated remaining completion time for the task. The Resource Manager also provides projected task latencies for use by the least-laxity scheduler.

The Resource Manager employs a three-level feedback loop. The tightest level (milliseconds) uses measurements of elapsed time to refine the estimated residual laxity of executing tasks, which are used by the least-laxity scheduler. The second level (fractions of a second) uses measurements of elapsed time and measurements of the resource load to refine the initial estimates of the laxity for the tasks as they start. The third level (several seconds) uses the measured resource load and residual laxities to revise the allocation of objects to processors.
6.3 Profilers
Each Profiler monitors the use of its processor’s resources to characterize the behavior of the application and to supply feedback to the Resource Manager. The Resource Manager allocates objects to processors; the objects execute and use resources, and the Profilers report resource usage to the Resource Manager, which then allocates additional objects or reallocates existing objects. The Profilers also report processing and communication times for the method invocations, from which the laxities of the tasks can be estimated.

7 Evolution Management
Without the ability to upgrade the software, and also the hardware, no enterprise application can claim to be able to operate continuously. Exploiting object replication, the Eternal Evolution Manager readily supports evolution of, and upgrades to, the hardware. Hardware components can fail, be repaired or replaced, and be reintegrated into the system without interruption of service. If the replacement hardware is of a different design (e.g., different byte order or data representation), the interoperability features of CORBA enable the application to adapt to the change.

With existing technologies, in a conventional enterprise application, the system must be stopped in order to upgrade the software (i.e., the application objects themselves). With Eternal, the system need not be stopped to upgrade the application objects. By exploiting object replication, the Evolution Manager accomplishes the overall change to the objects incrementally and systematically, while the application continues to operate.

The Evolution Manager, which comprises the Preparer and the Upgrader shown in Figure 4, performs the upgrade of a large program in a sequence of steps. Each step of the sequence is completed and demonstrated to operate satisfactorily before the next step is undertaken. A step in the sequence consists of three phases: a preparation phase in which the programmer prepares a new upgraded program that is to replace the existing program, a preliminary preprocessing phase that involves the Preparer with the assistance of the human, and a fully automatic upgrade phase that involves the Upgrader.

7.1 Preparer
To upgrade an object class, the programmer submits, to the Preparer, the code of both the existing object class and a new version of the class. The Preparer compares the two classes and determines the differences. With assistance from the programmer, the Preparer generates one or more intermediate classes to facilitate the upgrade, and compiles and deposits those classes into the Implementation Repository for use by the Upgrader. No special skill is required of the application programmer beyond that required to program the application program being upgraded.

7.2 Upgrader
The Upgrader upgrades an object of a class using a sequence of invisible upgrades, each of which moves closer to the desired overall upgrade. The actual upgrade is performed in a single atomic action when all program code and data structures are in place. Further invisible upgrades then
remove obsolete or transitional code. The Upgrader ensures that at least one replica of an object continues to provide service while another replica is being upgraded.

Particular care is required for upgrades that modify the attributes (local state variables) of an object. Code must be generated by the Preparer with the assistance of the programmer, and invoked by the Upgrader, to set appropriate values for new attributes. Even more care is required if the signatures (parameters and their types) of the methods of the objects must be changed. Several classes of objects may need to be upgraded together in a coordinated upgrade sequence.

If evolution is required but fault tolerance is not, then replication is necessary only while the system is being upgraded. Such systems operate normally with unreplicated objects, but additional replicas are introduced when needed to support live upgrade and evolution. Replication thus provides not only fault tolerance but also live upgrades that allow enterprise applications to grow and evolve without interruption of service.

8 Related Work

Several systems that extend CORBA with fault-tolerance have been developed.

The Electra toolkit implemented on top of Horus provides support for fault tolerance by replicating CORBA objects, as does Orbix+Isis on top of Isis [1, 8]. Unlike Eternal, Electra and Orbix+Isis are non-hierarchical object systems that support only active replication. Both Electra and Orbix+Isis use an integration approach in that the replication and group communication mechanisms are integrated into the ORB and require modification of the ORB. In contrast, Eternal uses the interception approach, which requires no modification to the ORB.

Another approach to fault tolerance and high availability, adopted by the OpenDREAMS toolkit [4], adds replication and group communication as services (implemented as CORBA objects) on top of the ORB, and requires no modification of the ORB. The service approach exposes the replication of objects to the application programmer and allows the programmer to modify the class library to construct customized replication and group communication services. In contrast, Eternal is transparent to the application and the application programmer.

The Maestro toolkit [19] aims to add reliability and high availability to CORBA applications in settings where it is not feasible to make modifications at the client side. It includes an IIOP-conformant ORB with an open architecture that supports multiple execution styles and request processing policies. The replicated updates execution style can be used to add reliability and high availability on the client side.

The Distributed Object-Oriented Reliable Service (DOORS) [17] adds support for fault tolerance to CORBA by providing replica management and fault detection as service objects above the ORB. DOORS supports passive but not active replication and is not based on multicast group communication. The DoorMan management interface monitors DOORS and the underlying system to fine-tune the functioning of DOORS and to take corrective action, if their hosts are suspected of being faulty.

The AQuA framework [3] employs the Ensemble/Maestro [18] toolkits, the Proteus dependability property manager, and the Quality of Service for CORBA Objects (QuO) runtime system. Proteus determines the type of faults to tolerate, the replication policy, the degree of replication, the type of voting to use and the location of the replicas. Using a Quality of Service Description Language (QDL) to specify an application's expected usage patterns and QoS requirements, QuO modifies the configuration to meet those requirements dynamically, and provides mechanisms for measuring and enforcing quality of service contracts and taking appropriate actions when those contracts are violated.

Another CORBA-based system, developed by Nett, Gerget and Mock [15], provides adaptation to dynamic and unpredictable changes in the computing environment. Like Eternal, their system uses integrated monitoring, dynamic execution time prediction, and scheduling to provide time-awareness for standard CORBA object invocations.

While the above systems provide support for fault tolerance and resource management, they do not provide support for live upgrade and evolution of objects.

The Simplex Architecture [5], which is intended for online upgrades of control systems, is based on two abstractions, the replaceable unit abstraction and the cell abstraction. The replaceable unit abstraction allows an existing software module to be replaced online by another module with similar or enhanced functionality, while the cell abstraction represents a protected module which cannot be affected by other modules. These abstractions have been implemented in a real-time POSIX testbed, based on publish/subscribe communication, which is quite different from the multicast group communication employed by the Eternal system.

Another system that supports upgrades of system software, hardware and application software has been developed by Kanevsky, Krupp and Wallace [7]. Their system shares many characteristics with the Simplex Architecture but differs in its application to the evolution of long life-cycle defense systems.

Neither of those two systems has addressed the general problem of live upgrades of object-oriented programs that the Eternal system addresses.
9 Conclusion

Networked enterprise applications of the future will be difficult enough to develop without also requiring the application programmers to be experts in fault tolerance, live upgrades and resource management. The Eternal system simplifies the programming of fault-tolerant, evolvable enterprise applications by exploiting the location transparency, portability and interoperability that CORBA provides.

By using object replication, the Eternal system not only provides fault tolerance for enterprise applications but also enables enterprise applications to be upgraded while they continue to operate. Eternal aims to support the building of networked enterprise applications that can grow and change without stopping applications that can become Eternal.

References


