# LIVE UPGRADE TECHNIQUES FOR CORBA APPLICATIONS*

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

**Abstract** The ability to perform *live software upgrades* is essential for long-running applications that provide critical services. Program modifications are necessary as programmer errors and new user requirements are uncovered. If software is to remain relevant, it must be upgradable. The Eternal Evolution Manager allows distributed CORBA applications to be upgraded while they continue to provide service. In addition to avoiding planned downtime, the Evolution Manager accomplishes the difficult tasks inherent to software evolution with minimal help from the application programmer. With our live upgrade techniques, and the underlying fault tolerance of the Eternal System, we can allow applications to run forever.

## 1. INTRODUCTION

Halting an executing application to modify its source code has traditionally involved difficult tradeoffs. The importance of the intended code improvement must be weighed against the revenue loss that is inevitable when an application incurs planned downtime. It might never be feasible to bring down critical computer systems, such as those that control the life support systems in spacecraft or hospitals. Oftentimes, software modifications to fix programming errors or to improve functionality are forsaken because an application must provide continuous service.

*The research in this paper has been supported by DARPA/ONR Contract N00174-95-K-0083, DARPA/AFOSR Contract F3602-97-1-0248 and MURI/AFOSR Contract F49620-00-1-0330.

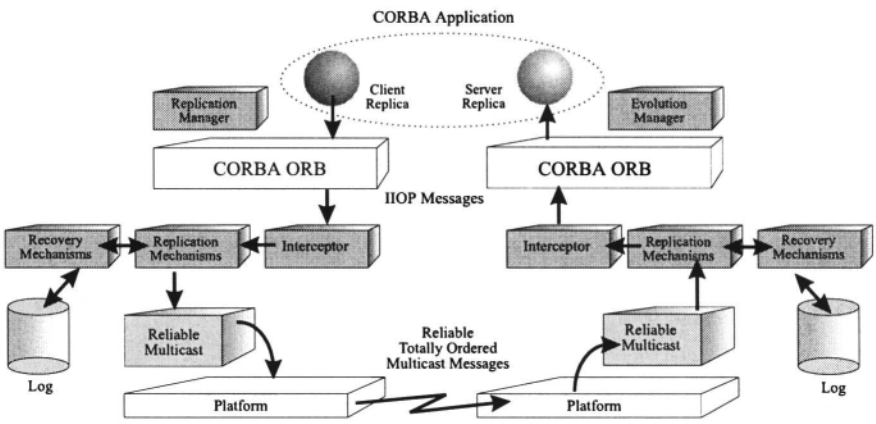


*Figure 1.* The Eternal System.

## 1.1. Fault Tolerant CORBA

The Common Object Request Broker Architecture [12] (CORBA) allows distributed client/server objects to interoperate, regardless of the operating system, platform or programming language being used. Clients invoke the methods defined in the server's Interface Definition Language (IDL) interface, through an Interoperable Object Reference (IOR) without knowing where the server resides in the network. The Object Request Broker (ORB) routes invocations and responses between distributed client and server objects.

The Eternal system, shown in Figure 1, provides CORBA applications with transparent fault tolerance. The Interceptor diverts the CORBA invocations and responses to the underlying Totem [10] protocol, which reliably multicasts the messages to all of the members of the recipient object group. The Logging-Recovery Mechanisms allows objects to recover from faults by initializing a new replica's state with the state of its object group. The Evolution Manager uses the object replication necessary for achieving fault tolerance to achieve live upgrades of CORBA application objects. [1]

The Replication Mechanisms place replicas into object groups. Assuming an active replication scheme, any invocation to the object group is received and responded to by all of the object replicas in the same order. Duplicate invocations and responses are suppressed so that replicas remain consistent.

[1] If evolution is required but fault tolerance is not, the objects are replicated only during the upgrade. Because the Replication Manager already provides replication to achieve fault tolerance, and because an application that requires no downtime from an upgrade will most likely require no downtime due to faults, replication for evolution effectively incurs no additional overhead beyond that required for fault tolerance.

Replicas can be added to (and removed from) object groups and the Replication Mechanisms ensure that the new replicas receive the proper messages. Thus, we kill and start individual replicas during an upgrade without interrupting the behavior of the object group as a whole.

# 2. THE ETERNAL EVOLUTION MANAGER

The Eternal Evolution Manager (composed of the Preparer and the Upgrader) upgrades CORBA applications without interrupting their execution. The Preparer, with application programmer assistance, performs static analysis of the original and new versions of an application. The Upgrader uses the results of this static analysis to perform a fully-automatic live upgrade. An application's performance is minimally impacted while an upgrade is underway, and entirely unaffected otherwise. The steps that must take effect during an upgrade are complicated, and would be tedious and error-prone for a human to perform. We automate most of these steps. The programmer is not required to maintain aspects of the old code (such as the old IDL interfaces) when writing the new version of the code. But objects must inherit from and implement the `Checkpointable` interface if they are intended to be upgradable, and, indeed, if they are to be recoverable by the fault tolerance part of the Eternal system. The `Checkpointable` interface contains a method *get_state( )* that retrieves the state of an object and a method *set_state( )* that initializes the state of an object. We aim to handle all *reasonable* code modifications, including those to an object's interface, method implementations, and renaming, removing or adding an instance variable, etc. Replacing an old version of an application with a completely unrelated new version is unreasonable as there is no way to transition between such "versions".

**The Eternal Preparer** compares the old version of the code *P_old* and the new version of the code *P_new* in preparation for the live upgrade. This offline analysis automatically generates *state transfer* code and, with application programmer assistance, generates *state conversion* code [15]. Whenever a replica is added to an object group, state is transferred from the object group to the new replica. The Preparer automatically generates the intermediate code *P_inter*, a superset of *P_old* and *P_new*, which executes during the transition between *P_old* and *P_new*. State conversion is needed during this transition if the structure of *P_old's* state differs from the structure of *P_new's* state. Additionally, the Preparer supplies the Upgrader with the necessary upgrade steps.

## 2.1. The Eternal Upgrader

The Upgrader performs a series of individual replacements which "nudge" the application towards an upgraded state but which neither affect its behav-

ior nor interrupt the executing application. The replicas being upgraded are replaced, one at a time [2], by their intermediate versions, which continue to execute the old methods. While one replica is down during this replacement, the other member(s) of the object group continue to provide service, masking the replacement from the application. Once all of the intermediate versions are executing, and once the object group is quiescent, we effect an *atomic switchover* after which only the new methods are executed. Because of the underlying reliable totally-ordered multicasts within the Eternal system, all of the replicas receive the switchover messages at the same logical time and, thus, the state of the upgraded replicas remains consistent. Then, to clean up the application, the intermediate object replicas are replaced, one at a time, with final versions containing only the new version of the code.

## 2.2. An Example Upgrade

The upgrade process is simplest when the new version of an object has an interface that is syntactically and semantically identical to the interface of the old version. *An interface-preserving upgrade* can be isolated from the other application objects. When an object's interface changes during an upgrade, the situation becomes more complex. Consider a *coordinated upgrade* in which the original server contains a method *B.m(void)* that is upgraded to *B.m(int)*. Upgrading the *B* servers before upgrading the *A* clients can result in invalid client invocations of *B.m(void)* on the new server.

We use *coordinated upgrade sets* to upgrade multiple objects concurrently. The clients that invoke a method undergoing an interface change must be upgraded at the same time as the modified server to provide the correct parameters for the invocation. The mechanics of a coordinated upgrade strongly resemble those of an interface-preserving upgrade. We therefore only describe the more complicated scenario, shown in Figure 2. The figure shows the sequence of invisible replacements performed during the coordinated upgrade of a simple application.

## 2.3. Quiescence

An object cannot be upgraded unless it is *quiescent* (not executing any of its methods). For example, consider a method *A.m( )* that invokes a method *B.n( )*. If *B.n( )* has been invoked but has not completed, then *B* is not quiescent, because it is executing, and *A* is not quiescent because it is stalled waiting for a response from *B*. Note that the invocations discussed here are synchronous. We cannot

[2] The replicas are replaced individually because we aim to preserve the fault tolerance of the application during the upgrade.

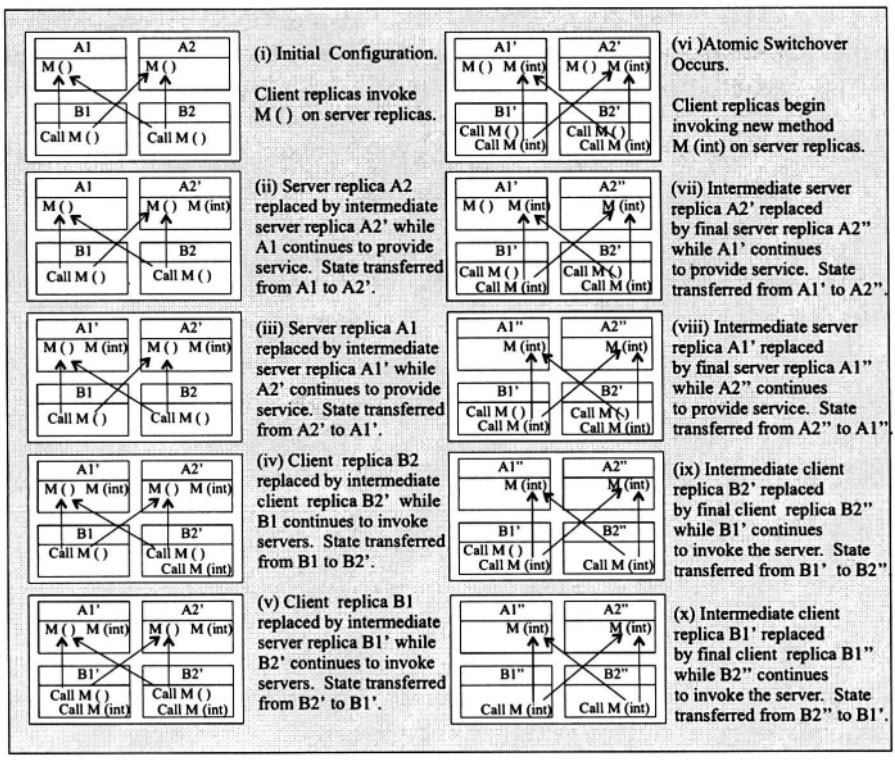


*Figure 2.* The invisible replacements performed during a coordinated upgrade.

| Replica A1 | Replica A2 | Replica A3 |
|---|---|---|
| aMethod() {<br>a++; ⇐ *state transfer*<br>b = a + b;<br>c = a*2+b;<br>} | aMethod() {<br>a++;<br>b = a + b; ⇐ *state*<br>c = a*2+b; *transfer*<br>} | aMethod() {<br>a++;<br>b = a + b;<br>c = a*2+b; ⇐ *state*<br>*transfer*<br>} |

*Figure 3.* Upgrading without waiting for quiesence yields indeterminate results.

completion of those operations is signaled to the Upgrader.

An object must be quiescent when it undergoes an invisible replacement and during the atomic switchover, both of which involve a transfer of state. To see why this condition is necessary, consider transferring the state of three replicas *A1*, *A2* and *A3* of the same object when they are executing *aMethod( )*. The replicas have finished executing the lines of code indicated by the arrows in Figure 3, and hence have different states. Eternal suppresses duplicate messages

original Count class

```
interface Count {
    void updateCount();
}
class Count {
    int countVal;
    int countDir;
    void updateCount() {
        if (countDir == 1)
            countVal++;
        else if (countDir == -1)
            countVal–;
    }
    void changeDirection(int dir) {
        countDir = dir;
    }
}
```

new Count class

```
interface Count_new {
    void updateCount(in long inc);
}
class Count_new {
    int countVal;
    int countDir;
    void updateCount(int inc) {
        if (countDir == 1)
            countVal += inc;
        else if (countDir == -1)
            countVal -= inc;
    }
    void changeDirection(int dir) {
        countDir = dir;
    }
}
```

*Figure 4.* The old and new versions of the Count class.

so we cannot predict which of the replicas' *get_state( )* operations will return, and a deterministic state cannot be returned. Similarly, receiving the state at a non-quiescent point would result in an inconsistent result.

## 3. UPGRADE CODE GENERATION

Our live upgrade methodology utilizes an intermediate period, during which the old code coexists with the new code within the automatically generated intermediate code. Before the *switchover*, the intermediate code behaves like the old version of the code and after the *switchover*, the intermediate code behaves like the new version of the code. By encapsulating the old and new functionality within a single object, switching between the executing code versions is expedited.

Consider a class `Count` whose *update Count( )* method increments or decrements `countVar`. The new version of the method increases or decreases `countVar` by `inc`. Both classes are shown in Figure 4.

The Preparer parses the two classes and automatically generates the intermediate class `Count_inter` shown in Figure 5. It contains a member variable `switchFlag`, indicating whether the switchover has occurred, and which determines whether incoming messages are "handled" by the old code or the new code. The methods defined in the intermediate object are a superset of those defined in both the old and the new code versions. The old state variables are kept distinct from the new state variables by appending the variables with either the *_old* or the *_new* suffix. The state of the old variables is transferred (and

```
interface Count_inter {
    void updateCount();
    void updateCount(in long inc);
}
class Count_inter {
    unsigned short switchFlag;
    int countVal_old, countDir_old;
    int countVal_new, countDir_new;
    void UpdateCount() {
        if (switchFlag == 0) {
            if (countDir_old == 1) countVal_old++;
            else if (countDir_old == -1) countVal_old--;
        }
    }
    void UpdateCount(int inc) {
        if (switchFlag == 1) {
            if (countDir_new == 1) countVal_new += inc;
            else if (countDir_new == -1) countVal_new -= inc;
        }
    }
    void ChangeDirection(int dir) {
        if (switchFlag == 0) countDir_old = dir;
        else countDir_new = dir;
    }
    void set_state() {
        countVal_new = countVal_old;
        countDir_new = countDir_old;
    }
}
```

*Figure 5.* Automatically generated intermediate code for the `Count` object.

| original IDL | intermediate IDL 1 | intermediate IDL 2 | new IDL |
|---|---|---|---|
| UpdateCount() | UpdateCount()<br>UpdateCount_(int) | UpdateCount_(int)<br>UpdateCount(int) | UpdateCount(int) |

*Figure 6.* The old, intermediate and new versions of the `Count` class.

converted, if necessary) to the new variables entirely within the intermediate object, avoiding an unnecessarily time-consuming encoding and decoding of state. By breaking the method implementation encapsulation of `Count` and `Count_new` within `Count_inter`, the *set_state( )* method can assign the (most likely) private state variables directly, without using their (possibly lacking) accessor functions.

Methods cannot be overloaded in IDL interfaces, so the `Count_inter` IDL code shown in Figure 5 (which contains both *UpdateCount( )* and *Update-*

*Count(int)*) will not compile. We have presented the sample intermediate code this way for the sake of simplicity. The Preparer actually generates two different intermediate objects (see Figure 6), temporarily renaming the *UpdateCount* method. The atomic switchover occurs when *intermediate IDL1* is executing.

# 4. WRAPPER FUNCTIONS

*Table 1.* Wrapper function feasibility.

| Original Method | New Method | Feasibility |
|---|---|---|
| aMethod(int i, int r) | aMethod(int i) | trivial |
| aMethod(float i) | aMethod(int i) | trivial |
| aMethod() | aMethod(int i) | questionable |

Wrapper functions translate between syntactic and semantic differences in methods and their synthesis requires significant programmer assistance. Some method signature upgrades cannot be masked with wrapper functions. The examples shown in Table 1 illustrate the feasibility of using wrapper functions for different types of method upgrades.

The first method upgrade in Table 1 eliminates a parameter from the new method. After receiving user verification that the *i* variables are equivalent, translation between the two methods is accomplished by not passing the second parameter to the new method. The wrapper function that is automatically generated is

```
aMethod(int i, int r) {
  wrappedObj → aMethod(i);
}
```

The second method upgrade in Table 1 changes the type of a method parameter. It is simple to convert from a float to an integer, and, with input from the application programmer, translation between less obvious types changes is tractable.

It is much more difficult to write a wrapper function for an upgrade that adds information to a method. The Preparer cannot automatically generate wrapper code that initializes *i* for the third upgrade in Table 1, although the application programmer might be able to provide an initialization routine.

## 4.1. Wrapper Functions to Handle Pure CORBA Clients

In Figure 7, `ClientObj` invokes *InvokeMiddleTier (int iDelay, int rDelay)* on `MiddleObj`. After waiting *iDelay* seconds, `MiddleObj` invokes *UpdateCount( )*

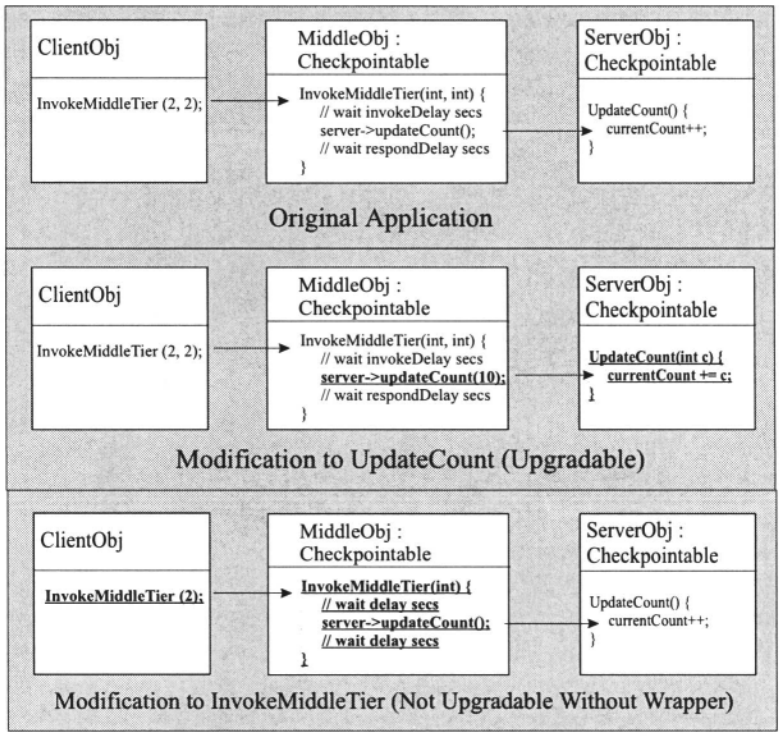


*Figure 7.* Example illustrating the use of wrapper functions during an upgrade.

on `ServerObj`, incrementing *currentCount*. `MiddleObj` then waits *rDelay* seconds before returning. One possible upgrade, the middle component in Figure 7, is for `MiddleObj` to pass an integer *c* to `ServerObj`, instructing `Serverobj` to increase *currentCount* by *c*. The coordinated upgrade set for this upgrade consists of the `ServerObj` replicas and the `MiddleObj` replicas, both of which are upgradable.

But how can the application programmer upgrade *InvokeMiddleTier (int, int)* to the method *InvokeMiddleTier (int)* (the lower component of Figure 7)? Because `ClientObj` invokes *InvokeMiddleTier*, it too must be upgraded. But `ClientObj` is a pure client; it does not implement the *Checkpointable* interface (it does not implement any interface), so it cannot undergo a live upgrade. `MiddleObj` can undergo an coordinated upgrade without disturbing `ClientObj` by using a wrapper function. A wrapper function is easy to generate for the *InvokeMiddleTier* upgrade, however, not all method upgrades can be masked by a wrapper function. The presence of pure clients in an application can make it impossible for otherwise upgradable objects to undergo certain upgrades, and the pure clients themselves cannot be upgraded. Even if the interfaces used by

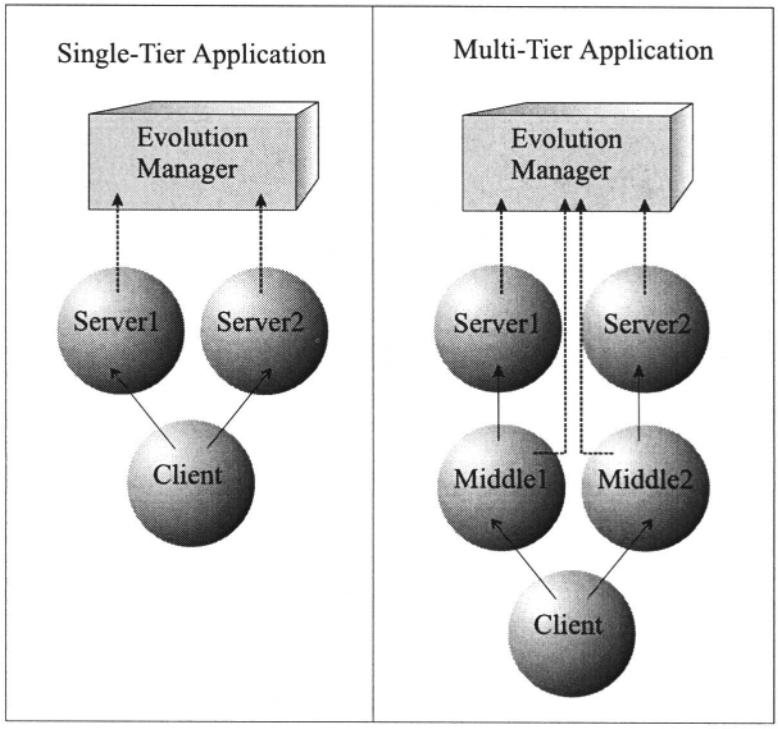


*Figure 8.* The single-tier and two-tiered applications undergoing upgrades.

pure clients can be modified, wrapper functions will permanently reside in the application.

Therefore, it is desirable to minimize the number of pure clients in an application. If the clients are made to implement the *Checkpointable* interface, the formerly pure client objects are effectively transformed into client/server objects. However, the server functionality is invoked only by the Eternal infrastructure. As far as the application is concerned, the object remains a client.

## 5. PERFORMANCE MEASUREMENTS

The Evolution Manager is implemented using Vertel's e*ORB2.1 and runs over a network of 360 MHz UltraSparc 5's running Solaris 8. Measurements were taken to ascertain the performance degradation experienced by an application during an upgrade.

We took measurements on an interface-preserving upgrade of a single-tier application, and an interface-changing upgrade of a two-tier application, both of which are shown in Figure 8. In the single-tier application, the client sends a constant stream of invocations to a doubly replicated server. In the two-

tier application, the client sends a constant stream of invocations to a doubly replicated middle object, which then invokes a doubly replicated server. Both upgrades change one of the server's methods, but the second upgrade changes a server's interface.

Five experiments were performed on each application. The first two experiments measure the throuhput of the application in an isolated configuration (not connected to the Evolution Manager). The third and fourth experiments measure the throuhput of the application in a non-isolated configuration without starting the upgrade. The non-isolated configuration includes all of the objects and connections shown in Figure 8. The isolated and non-isolated case were tested separately to determine how connection establishment alters application performance. We also wanted to determine how much killing and restarting replicas degrades performance. For the second and fourth measurements, we manually killed and restarted the application objects, mimicking the object group membership changes that occur during an upgrade's invisible replacements. The final throuhput measurements were performed while the live upgrade was underway.

*Table 2.* Application performance during an upgrade.

| Configuration being Measured | Roundtrip | Penalty |
|---|---|---|
| Interface-Preserving Isolated: | | |
| (1) Client invokes two server replicas (baseline) | 152 | – |
| (2) Both server replicas killed/restarted 2xs | 132 | 13% |
| Interface-Preserving Non-Isolated: | | |
| (3) Client invokes two server replicas | 135 | 11% |
| (4) Both server replicas killed/restarted 2xs | 115 | 24% |
| (5) Both servers upgraded (duration 8.9 s) | 98 | 35% |
| Interface-Changing Isolated: | | |
| (1) Client invokes two tiers (baseline) | 57 | – |
| (2) All server replicas killed/restarted 2xs | 52 | 9% |
| Interface-Changing Non-Isolated: | | |
| (3) Client invokes two tiers | 48 | 16% |
| (4) All server replicas killed/restarted 2xs | 43 | 25% |
| (5) All servers upgraded (duration 19.5 s) | 38 | 33% |

The experimental results are shown in Table 2. The single-tier (two-tier) application experienced four (eight) kill/restarts, and hence four (eight) transfers of state. To maintain replica consistency, replicas queue messages while state is transferred. Also, the underlying Totem relable multicast protocol [10] is slowed down during membership changes. These two effects decrease the throuhput by 13% (9%).

Both applications experience performance degradation while running in the non-isolated configuration. The extra Evolution Manager objects run on additional hosts, and because Totem is based on a logical token passing ring, extra hosts increase the round-trip time of the token. The extra hosts and the extra connections combine to decrease the throuhput of the single-tier (two-tier) application by 11% (16%).

The combined effects of killing and restarting replicas, and running the application in a non-isolated configuration result in a 24% (25%) decrease in throuhput. The performance penalty for performing a live upgrade is not substantially larger. The single-tier (two-tier) upgrade takes 8.9 (19.5) seconds and decreases throuhput by 35% (33%). The Evolution Manager competes with the application for computing resources, and the upgraded objects are briefly stalled during the switchover.

These are preliminary results. Neither the Evolution Manager, nor the underlying protocols has been performance tuned. In particular, the underlying protocols are not optimized for the frequent object group membership changes that live upgrades require. However, it is our assessment that these are extremely reasonable results, especially because these experiments represent a worst-case scenario in several ways. It is unlikely that the applications being upgraded will undergo the message invocation bombardment experienced by these example applications. Also, the upgrades were performed very rapidly for these experiments. The Evolution Manager performed one invisible replacement after the other, as quickly as possible. The upgrade's impact would be dampened by spreading the upgrade out over a larger timescale. Future experiments will investigate more reasonable upgrade durations and the impact of state size and achieving quiescence on the performance of an application undergoing an upgrade.[3]

## 6. RELATED WORK

Kramer and Magee's classic evolving philosophers paper [8] concerns itself with the structural issues involved in dynamic reconfiguration. All of the objects being upgraded are passivated, as are all of the objects that can invoke these objects. This *passive set* includes all of the objects that can initiate a nested operation that eventually invokes an object being upgraded. In a highly connected system, this technique can significantly hinder the application's availability during an upgrade. Goudarzi and Kramer [6] quiesce a group of objects being upgraded, the *BSet*, by only blocking the *non-BSet* objects that invoke the *BSet* during the quiescence algorithm. Bidan *et al* [1] have developed a

[3] The state in both of these examples was small, and the nature of these applications ensured that quiescence would be reached quickly.

Dynamic Reconfiguration Manager (DRM) for CORBA that passivates a group of objects undergoing reconfiguration. Only the links between the group being made passive and the objects external to that group are blocked, not the external objects themselves.

The Simplex architecture [5] executes multiple versions of a program simultaneously and shields the old code from the new version until its correctness is established. However, the types of upgrades allowed by Simplex are limited. Feiler [3] has developed an offline analysis tool that determines when the upgrade of one component will require the upgrade of additional components in the system. Syntactic, type and semantic incompatibilities in the component connections are considered.

Senivongse [16] has developed a mediator that makes the evolution of server objects transparent to their clients. A mapping operator (similar to Eternal's wrapper functions) transforms and forwards the client's old requests to the new server, and then transforms and forwards the new server's responses to the old client. The generation of these mapping operators is semi-automatic. One problem with this technique is that multiple upgrades to a server yield multiple mapping operators performing multiple forwards, degrading performance. The application programmer ensures that an upgrade can be masked by mapping operators.

Bloom [2] used the Argus system [9] to replace modules while preserving their state. The lack of subtyping in Argus limits the types of upgrades that can be performed. Adding a method to an object changes the object's type, and if an invocation returns an object of this changed type, type correctness is violated and the upgrade cannot be performed.

Hauptmann and Wasel [7] accomplish live upgrades by including the upgrade code within a separate application thread. This code must exist in any upgradable application object even when no upgrade is underway. Additionally, significant programmer assistance is required to generate the upgrade-specific code, although the authors state that a tool could be written to automate much of this work. The authors assume that as long as the interfaces remain the same, the upgrade of multiple components can be broken up into a sequence of independent upgrades, but do not explain how correct program semantics is maintained. Felber [4] includes a brief conceptual discussion a replication-based solution to on-line upgrades using the Object Group Service (OGS) in his dissertation, but does not provide much discussion of on-line upgrade issues.

Podus [14] is an example of a system that loads new code into memory and changes the binding of procedures to perform live upgrades. Methods are upgraded when they are not executing, and the method upgrades are ordered such that a new version of a method cannot invoke an old version of another method. The drawback of this approach is that the mechanisms used to achieve

the dynamic upgrade are complicated and require specialized compilers and linkers.

# 7. CONCLUSIONS AND FUTURE WORK

The Eternal Evolution Manager enables live upgrades of distributed CORBA applications by exploiting a reliable totally-ordered multicast protocol and object replication. The aim is to automate as much of the upgrade process as possible both to reduce programmer errors and to encourage previously infeasible application upgrades.

We have successfully performed interface-preserving and interface-changing live upgrades on simple applications and are convinced that our live upgrade mechanisms work properly. The next step is to perform an upgrade with alarge coordinated upgrade set. We plan to investigate ways to circumvent the anticipated coordinated quiescence bottleneck, including using wrapper functions to reduce the size of the coordinated upgrade set, and using semantic program knowledge (obtained from the application programmer) to break an application version change into multiple upgrades with smaller coordinated upgrade sets. We also plan to investigate programming styles and techniques that minimize the amount of time that the Upgrader must wait for a coordinated upgrade set to become quiescent.

# References

[1] C. Bidan, V. Issarny, T. Saridakisand, and A. Zarras, "A dynamic reconfiguration service for CORBA," *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 35-42.

[2] T. Bloom and M. Day, "Reconfiguration and module replacement in Argus: Theory and practice," *Software Engineering Journal* (March 1993), pp. 102-108.

[3] P. Feiler and J. Li, "Consistency in dynamic reconfiguration," *Proc. of the 4th Intl. Conf. on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 189-196.

[4] P. Felber, "The CORBA Object Group Service: A Service Approach to Object Groups in CORBA," Ph.D. Thesis, École Polytechnique Fédérale de Lausanne, (1998).

[5] M. Gagliardi, R. Rajkumar and L. Sha, "Designing for evolvability: Building blocks for evolvable real-time systems," *Proc. of the IEEE 1996 Real-Time Technology and Applications Symposium,* Brookline, MA (June 1996), pp 100-109.

[6] K. M. Goudarzi and J. Kramer, "Maintaining node consistency in the face of dynamic change," *Proc. of the 3rd Intl. Conf. on Configurable Distributed Systems*, Annapolis, MD (May 1996), pp. 62-69.

[7] S. Hauptmann and J. Wasel, "On-line maintenance with on-the-fly software replacement," *Proc. of the 3rd Intl. Conf. on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 70-80.

[8] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions On Software Engineering*, vol. 16, no. 11 (November 1990), pp. 1293-1306.

[9] B. Liskov, "Distributed programming in Argus," *Communications of the ACM* (March 1988), pp. 300-313.

[10] 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*, vol. 39 , no. 4 (April 1996), pp. 54-63.

[11] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems* vol. 4, no. 2, (1998), pp. 81-92.

[ 12] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Revision 2.4.2, OMG Technical Document Formal/2001-02-01, Object Management Group (February 2001).

[13] Object Management Group, "Online Upgrades Request for Information," OMG Technical Document realtime/2000-08-01 (August 2000).

[14] M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software* (March 1993), pp. 53-65.

[15] L. A. Tewksbury, L. E. Moser and P. M. Melliar-Smith, "Automatically generated state transfer and conversion code to facilitate software upgrades," *Maintenance and Reliability Conf. Proc*., Gatlinsburg, TN (May 2001).

[16] T. Senivongse, "Enabling flexible cross-version interoperability for distributed services," *Proc. of the Intl. Symposium on Distributed Objects and Applications*, Edinburgh, Scotland (September 1999), pp. 201-210.