Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications [*]

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

In CORBA-based applications that depend on object replication for fault tolerance, inconsistencies in the states of the replicas of an object can arise when concurrent threads within those replicas perform updates in different orders. By imposing a single logical thread of control on every replicated multithreaded CORBA client or server object, and by providing deterministic scheduling of threads and operations across the replicas of each object, the Eternal system achieves consistent object replication. The Eternal system does this transparently, with no modification to the application, the ORB, or the concurrency model employed by the ORB.

1 Introduction

Distributed object systems, based on standards such as the Object Management Group's Common Object Request Broker Architecture (CORBA) [9], provide applications with valuable features such as language transparency, interoperability and location transparency. Unfortunately, most distributed object standards, including CORBA, make no provision for other desirable features, such as fault tolerance, that current and future CORBA-based applications will need. The Object Management Group (OMG) has recognized the need to provide fault tolerance for CORBA applications by issuing a Request for Proposals (RFP) [10] for fault-tolerant CORBA.

A major requirement of the OMG's RFP is that strong replica consistency must be maintained as operations are performed that change the states of the replicas. However, strong replica consistency requires that the behavior of the application objects is deterministic. Unfortunately, many practical CORBA applications use nondeterministic mechanisms such as local timers, processor-specific functions and, in particular, multithreading.

The Eternal system [7] addresses this challenge in its provision of fault tolerance for CORBA applications, while ensuring low overheads, transparency and interoperability. To ensure strong replica consistency, the Eternal system exploits reliable totally ordered multicasts for the communication of operations between replicated objects, and implements mechanisms for the detection and suppression of duplicate operations and the transfer of state to new, recovering and backup replicas. This paper focuses on the mechanisms that the Eternal system employs to guarantee consistent replication in the face of one specific source of nondeterminism, namely, multithreading in the ORB or the application.

2 Replicating Multithreaded Applications

Many commercial ORBs are multithreaded, and multithreading can yield substantial performance advantages. Unfortunately, the specification of multithreading in the CORBA 2.2 standard [9] does not place any guarantee on the order of operations dispatched by a multithreaded ORB. In particular, the specification of the Portable Object Adapter (POA), which is a key component of the CORBA standard, provides no guarantee about how the ORB or the POA dispatches requests across threads. The ORB may dispatch several requests for the same object within multiple threads at the same time.

In addition to ORB-level threads, the CORBA application may itself be multithreaded, with the thread scheduling having been determined by the application programmer. The application programmer must ensure correct sequencing of operations and must prevent

[*]This research has been supported by the Defense Advanced Research Projects Agency in conjunction with the Office of Naval Research and the Air Force Research Laboratory, Rome, under Contracts N00174-95-K-0083 and F3602-97-1-0248, respectively.

thread hazards. Careful application programming can ensure thread-safe operations within a single replica of an object; however, it does not guarantee that threads and operations are dispatched in the same order across all of the replicas of the object. Making the application programmer responsible for concurrency control and ordering of dispatched operations in replicated objects is unacceptable for maintaining strong replica consistency in a fault-tolerant system.

Several different concurrency models [13] are supported by current commercial ORBs. These include thread-per-request, where one thread is spawned for each new invocation on an object, and thread-per-object, where a single thread executes all invocations on an object. Most practical CORBA applications consist of servers that contain multiple objects, each having possibly multiple threads. Objects that are co-located within the same server process may access and update shared data; thus, irrespective of the threading model used by the ORB, multiple threads may exist within each server. Because a server may itself assume the role of a client, we do not distinguish between the problem of multithreading for clients and servers.

We use the term *MT-domain* to refer to any CORBA client or server that supports multiple (application-level or ORB-level) threads that may access shared data and that contains one or more CORBA objects. The MT-domain abstraction is independent of the concurrency model of the ORB, the role of the MT-domain as a client or server, and the commercial ORB that hosts the MT-domain.

In Sections 2.1 and 2.2, we provide examples that illustrate how replica inconsistency can arise for active and passive replication of MT-domains. We assume that all of the mechanisms that guarantee replica consistency for single-threaded objects are present, *i.e.*, messages are delivered to the ORB in a reliable totally ordered sequence, duplicate operations are detected and suppressed, and state transfers are provided for recovering replicas. While these mechanisms suffice to guarantee replica consistency, they serve only to facilitate, but not to guarantee, replica consistency when either the ORB or the application is multithreaded. In particular, reliable totally ordered multicasts ensure only that the ORBs that host the various replicas receive the same messages in the same order. They do not guarantee that the ORBs will dispatch these incoming messages onto the threads of the replicas in the same order.

We assume further that other sources of nondeterminism, *e.g.*, system calls (such as local timers) that return processor-specific information, are handled by other mechanisms. We also assume that all replicas of the application are located on the same type of platform, i.e., same vendor's ORB, same operating system, same type of workstation (same processing speed, same amount of memory, etc). In addition, we assume deterministic behavior of the operating system. Thus, any replica inconsistency that arises can be attributed to the multithreading of the ORB or the application.

2.1 Active Replication

For active replication, strong replica consistency means that, at the end of each operation invoked on the replicas, each of the replicas of an object have the same state. The example shown in Figure 1 illustrates the problem of replica inconsistency when the only support for consistent replication is that provided for single-threaded ORBs.

In the figure R_1 and R_2 are active replicas of the same MT-domain. The ORB at each replica receives messages in the same order, but dispatches two threads T_1 and T_2 to perform operations on the replicas. The two threads are simultaneously active within each replica, and can access and update shared data. Suppose that threads T_1 and T_2 issue update operations A and B, respectively, on the shared data and that operation B (A) is executed before operation A (B) in replica R_1 (R_2). Even if the MT-domain is programmed to avoid race conditions and other thread hazards, the order of operations in the two replicas of the MT-domain may differ and, thus, their states may be inconsistent at the end of the update operations.

Replica inconsistency can also arise when the ORB initially dispatches only a single thread, which later spawns other threads within the same replica.

2.2 Passive Replication

For passive (primary-backup) replication, strong replica consistency means that, at the end of each state transfer, each of the replicas of an object have the same state. The example shown in Figure 2 illustrates the problem of replica inconsistency for passive replication.

In the figure MT-domain C is passively replicated, with primary replica C_2 and backup replica C_1. Objects A, B, D and E (not shown) with which C interacts are not necessarily multithreaded or replicated. The example focusses on the passive replication of MT-domain C, and how C's multithreading results in the inconsistency of its replicas.

The invocation I_{AC} (I_{BC}) of object A (B) on MT-domain C requires thread T_1 (T_2) to be dispatched. If thread T_1 (T_2) is dispatched, MT-domain C will issue the nested invocation I_{CE} (I_{CD}) to object E (D). Thus, MT-domain C invokes two different nested operations through its two threads, and must obtain results

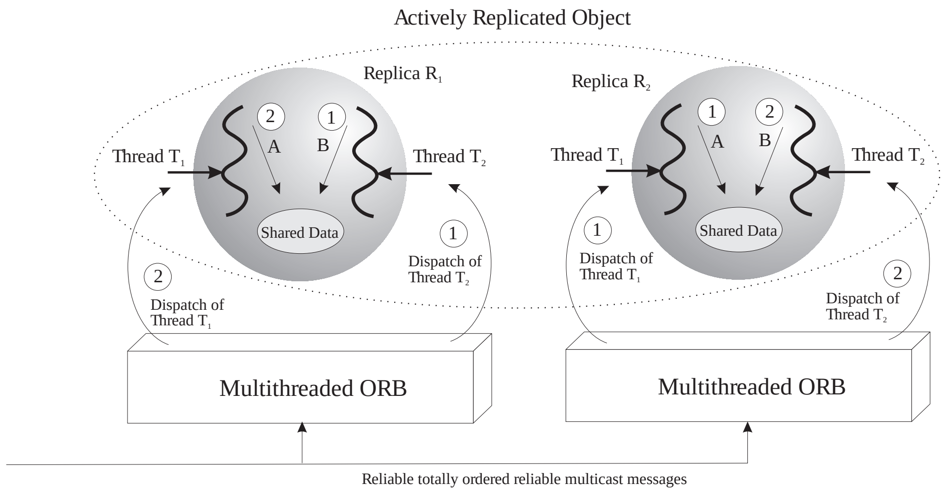


Figure 1: Inconsistency with active replication of multithreaded objects.

from both operations. Consider the following sequence shown in Figure 2:

1. The primary replica C_2 is initially operational. The ORB hosting C_2 dispatches thread T_2 to handle invocation I_{BC} first.
2. Thread T_2 issues a nested invocation I_{CD} on object D.
3. The primary replica fails before handling invocation I_{AC}. The backup replica C_1 becomes the new primary replica for MT-domain C.
4. The ORB hosting the new primary replica C_1 dispatches thread T_1 to handle invocation I_{AC} first.
5. Thread T_1 issues a nested invocation I_{CE} on object E.
6. Before the new primary's ORB handles invocation I_{BC}, object D returns the response R_{CD} to the old primary's nested invocation I_{CD}. The receiving ORB delivers this response to C_1, which is unable to handle this response to the nested invocation (I_{CD}) that it has no knowledge of ever having issued.

The inconsistency arises precisely because of the nondeterministic behavior of multithreaded ORBs, and the lack of specification of the order of dispatch of the operations that such ORBs receive.

3 Enforcing Determinism

A MT-domain is the basic unit of replication for multithreaded applications in the Eternal system. To preserve replica consistency for MT-domains, the Eternal system provides mechanisms that govern the order in which the threads (and operations) are dispatched within each replica of a MT-domain, over and above the total order in which the messages containing the operations are delivered to the ORB.

The Eternal system *enforces* deterministic behavior within a MT-domain by allowing only a *single logical thread of control*, at any point in time, within each replica of the MT-domain. Although multiple threads may exist in a MT-domain, all of them must be related to (and required for the completion of) the single operation that "holds" the logical thread-of-control. Furthermore, at most one of these threads can be actively executing; all of the other threads must be suspended or awaiting a response.

The Eternal system controls the dispatching of threads and operations within every replicated MT-domain, transparently both to the objects and threads within the MT-domain, and to the multithreaded ORB that hosts the MT-domain. To achieve this, the Eternal system employs a deterministic *operation scheduler*[*] that is inserted into the address space of every replica of a MT-domain, and that maps incoming invocations to the thread-of-control within the replica, or enqueues unrelated invocations for later dispatch.

The scheduler dictates the creation, activation, deactivation and destruction of threads, within the replica of a MT-domain, as required for the execu-

[*]The scheduling of operations for replica consistency is orthogonal to the real-time scheduling of operations. The operation scheduler described here does not factor in any considerations for real-time operation.

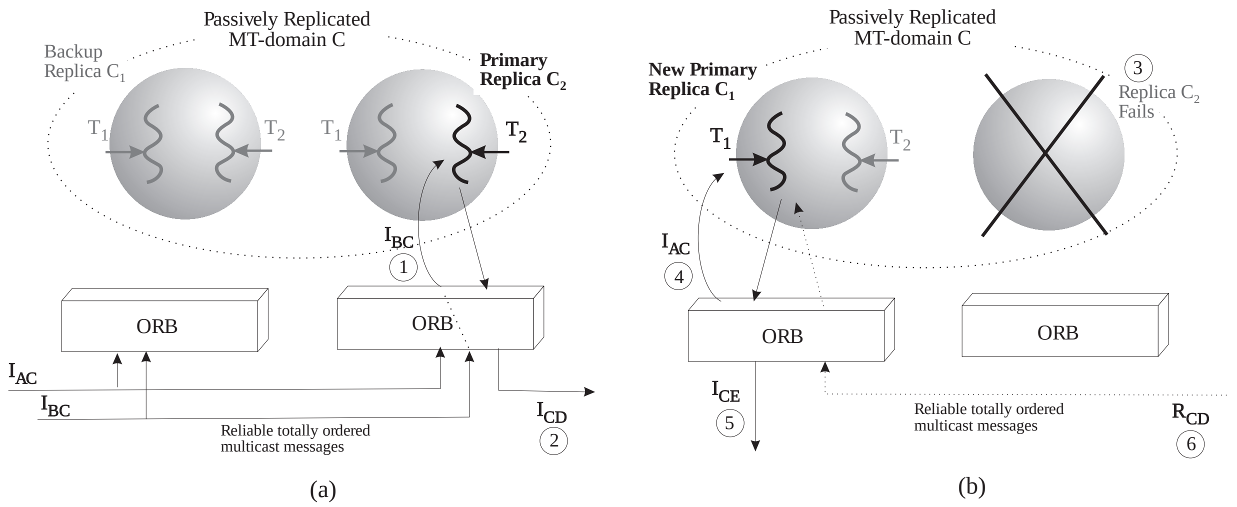


Figure 2: Inconsistency with passive replication of multithreaded objects when (a) the primary replica is initially operational, and (b) the primary later fails and the backup replica becomes the new primary replica.

tion of the current operation "holding" the logical thread-of-control. The scheduler is inserted into a position such that it can override any thread or operation scheduling performed by either the nondeterministic multithreaded ORB within the replica, or by the replica itself.

Operations are mapped identically onto the logical thread-of-control at all of the replicas of a MT-domain, thereby ensuring that the same operation "holds" the thread-of-control at each replica at any logical point in time. To enable this, the Eternal system ensures that the scheduler at each replica of a MT-domain receives the same sequence of totally ordered messages containing invocations and responses destined for the MT-domain. Based on this incoming sequence of messages, the scheduler at each replica decides on the immediate delivery, or the delayed delivery, of the messages to that replica. At each replica of a MT-domain, the scheduler's decisions are identical and, thus, operations and threads within the MT-domain are dispatched identically at each replica.

4 Scheduling for Consistency

While the MT-domain model may seem somewhat restrictive in terms of the effective concurrency achieved in the application, those restrictions are necessary to achieve replica consistency for replicated multithreaded CORBA applications. To ensure a single logical thread-of-control within the MT-domain, the scheduler may delay or reschedule invocations and responses on a MT-domain. This is necessary because another operation can assume the MT-domain's logical thread-of-control only when the current operation within the MT-domain completes.

Thus, the scheduler enqueues, in the order of their arrival, all incoming invocations and responses that are unrelated to the current operation or the logical thread-of-control. The next operation to be scheduled on the MT-domain, upon release of the thread-of-control, is the first operation that has been enqueued or, in the absence of enqueued operations, the next operation that the scheduler receives.

Figure 3 shows a replica of a MT-domain, along with its operation scheduler, and the sequence of actions of the MT-domain's scheduler for the given totally ordered messages. All of the replicas of the MT-domain are forced to behave identically, as the example illustrates.

The MT-domain in this example consists of two objects A and B, each capable of supporting a thread. Invocations A_i, B_i and C_i are destined for objects A, B and C, respectively (object C is in some other MT-domain not shown in the figure). The invocation A_i gives rise to a nested invocation C_i; the invocation B_i is independent of both A_i and C_i.

At the start of the sequence of actions in Figure 3(a), there is no operation executing in the replica of the MT-domain, and the thread-of-control is free to be assumed by the next operation. Thus, the operation scheduler delivers the invocation A_i (which occurs first in the total order of messages), leading to a thread executing within object A. The scheduler assigns the MT-domain's thread-of-control to the logical

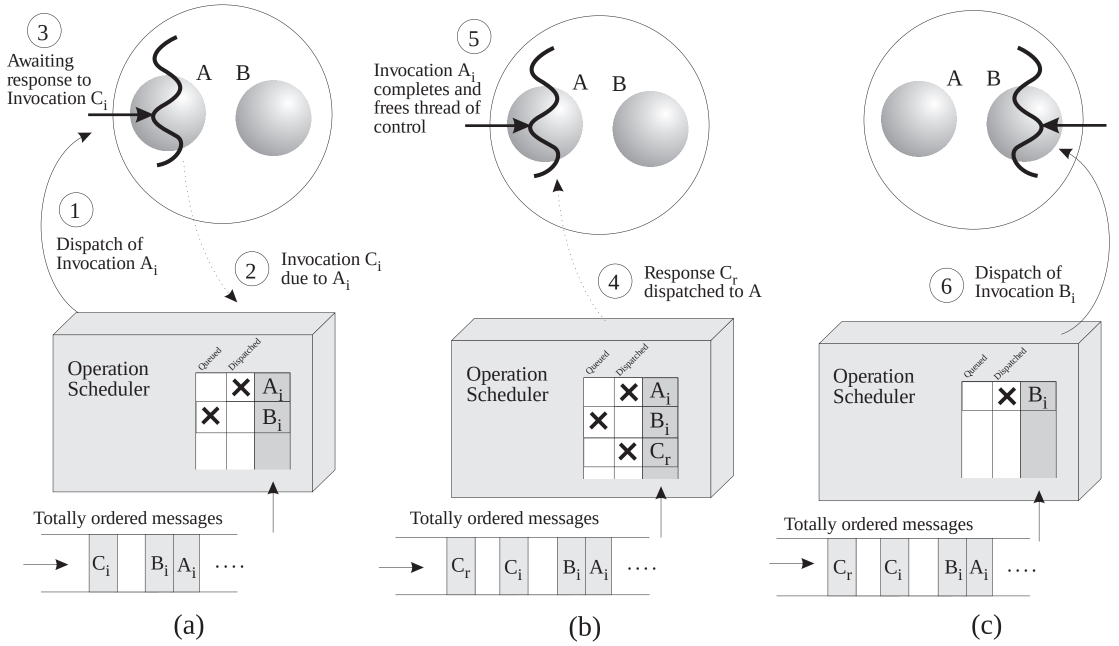


Figure 3: Sequence of actions of the operation scheduler at a replica of a MT-domain.

operation represented by the invocation A_i until A_i completes.

The invocation A_i on object A leads to the subsequent invocation C_i on object C, and A_i can complete only when the response C_r to the invocation C_i is returned to object A. Because the thread-of-control has been assigned to A_i, the scheduler delivers to the MT-domain only those incoming invocations and responses that correspond to A_i. In this case, the only message in the total order that is related to A_i is C_r. Note that the invocation B_i is independent of A_i. Thus, although B_i has been received by the scheduler ahead of C_r in the total order of messages, it is not delivered to its target object B until the thread-of-control is released by A_i. To deliver only those invocations related to the thread-of-control, the scheduler requires some means of recognizing, and relating, the operations contained in the totally ordered messages. Identifiers for associating operations with the thread-of-control are discussed in Section 4.2.

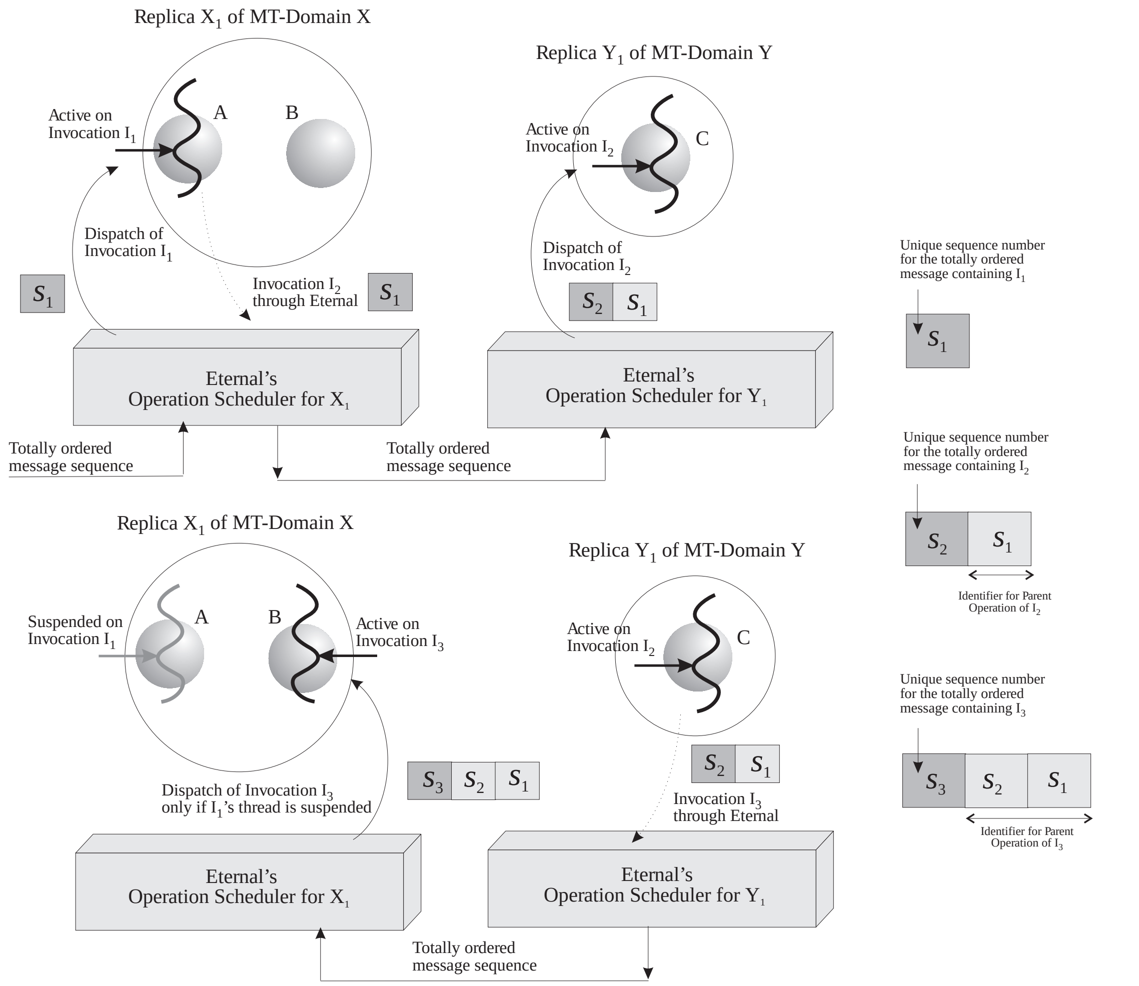
Figure 3(b) shows the receipt of the response C_r by the MT-domain, and its delivery to object A, when the scheduler determines that its delivery is appropriate.

After processing the response C_r, object A completes the invocation A_i and the thread-of-control again becomes available. The scheduler also performs garbage collection of the threads that were used by the thread-of-control. The MT-domain's scheduler then reassigns the thread-of-control to the next operation, which is the invocation B_i. The MT-domain's scheduler then delivers the invocation B_i, leading to a new thread of execution within the target object B, as shown in Figure 3(c).

This delaying of invocation B_i in favor of the response C_r (although B_i precedes C_r in the total order of operations) does not itself introduce any inconsistency between the replicas of the MT-domain. The reason is that the operation scheduler at each of the replicas of the MT-domain arrives at the same scheduling decision regarding the delivery of C_r before B_i.

Replica consistency is thus maintained as a result of the deterministic behavior *across all* of the replicas of a MT-domain through the totally ordered messages that they receive, as well as the deterministic behavior *within each* replica of the MT-domain through the identical scheduling of distinct operations onto a single thread-of-control.

Replica determinism, unfortunately but inevitably, reduces the degree of concurrency within the application. However, if objects are indeed independent of each other, and do not share data, they can be assigned to different processes and Eternal's operation scheduler will schedule them concurrently without restriction. The memory protection between processes, provided by the operating system, ensures that the objects do not share data and are indeed independent.



Replica X1 of MT-Domain X
Active on Invocation I1
A
B
Dispatch of Invocation I1
S1
Invocation I2 through Eternal
S1
Eternal's Operation Scheduler for X1
Totally ordered message sequence
Replica Y1 of MT-Domain Y
Active on Invocation I2
C
Dispatch of Invocation I2
S2 S1
Eternal's Operation Scheduler for Y1
Totally ordered message sequence
Unique sequence number for the totally ordered message containing I1
S1
Unique sequence number for the totally ordered message containing I2
S2 S1
Identifier for Parent Operation of I2
Replica X1 of MT-Domain X
Suspended on Invocation I1
A
B
Active on Invocation I3
Dispatch of Invocation I3 only if I1's thread is suspended
S3 S2 S1
Eternal's Operation Scheduler for X1
Replica Y1 of MT-Domain Y
Active on Invocation I2
C
S2 S1
Invocation I3 through Eternal
Eternal's Operation Scheduler for Y1
Totally ordered message sequence
Unique sequence number for the totally ordered message containing I3
S3 S2 S1
Identifier for Parent Operation of I3

Figure 4: Nested reentrant operations on a MT-domain. Scheduling identifiers assigned by the operation scheduler enable the detection of reentrant operations.

4.1 Reentrant Operations

In the example of Figure 3, the operation A_i was not reentrant on the MT-domain because the execution of A_i did not lead to further invocations on the *same* MT-domain. A reentrant operation may lead to multiple nested invocations on the same MT-domain, each of which must be delivered and executed in order for the operation to complete. Thus, for such an operation, the single logical thread-of-control is realized through multiple threads within the MT-domain, at most one of which is actively executing, while the others are suspended or awaiting a response.

Figure 4 shows the interaction between a replicated MT-domain X and a replicated MT-domain Y. Here, invocation I_1 on object A, when dispatched to the logical thread-of-control in X, results in the invocation I_2 of object C in the MT-domain Y. The invocation I_2 leads to a further nested invocation I_3 on object B within the MT-domain X. I_1 requires that I_2 completes, and I_2 requires that I_3 completes. Thus, the invocation I_3 must be allowed to proceed inside every replica of X and return a response to object C within every replica of Y. Because the scheduler ensures identical behavior at all of the replicas of a MT-domain, it suffices to consider replicas X_1 and Y_1 (shown in the figure) of the MT-domains X and Y, respectively.

To permit a second invocation I_3 on a MT-domain X_1 that already has the invocation I_1 pending a response, the scheduler for X_1 needs to verify that the second invocation is a *descendant* of the first (*parent*) operation and that the parent operation is suspended. A descendant of a particular parent operation is an invocation that arises from the execution of the parent operation, and that must be allowed to execute for the parent operation to complete. A descendant invocation may be reentrant. In this example, invocations

I_2 and I_3 are descendants of the parent invocation I_1, with I_3 being reentrant on the MT-domain X_1.

After the scheduler for X_1 determines that I_3 is a descendant of I_1 and that the thread for I_1 is suspended, awaiting a response (in this case from object C in Y_1), it proceeds to activate a thread to handle invocation I_3. If the thread executing I_1 is not suspended before I_3 is allowed to start executing, the states of the objects within X_1 may become inconsistent due to the multiple threads being active. The logical thread-of-control within X_1 is still associated with the first invocation I_1 because all other operations within the concurrency domain are direct descendants of I_1. Once the invocation I_3 completes and B returns a response to the invoking object C, the scheduler for X_1 disposes of the thread for I_3.

A reentrant descendant invocation on a MT-domain can generate further reentrant descendant invocations that are also reentrant on the same MT-domain. Thus, the MT-domain scheduler must maintain a stack of invocations dispatched in the MT-domain. Every reentrant descendant invocation is pushed onto the stack when it is dispatched onto the MT-domain, and popped off the stack when it completes.

4.2 Scheduling Identifiers

To handle nested reentrant operations, the operation scheduler uses *scheduling identifiers* that allow the scheduler to associate parent and descendant operations at the MT-domain level. These scheduling identifiers are internal to, and examined by, Eternal's operation schedulers, and are never seen by the CORBA application or the ORB.

At the point that it dispatches an invocation I_q onto the replica of the MT-domain that it controls, the operation scheduler assigns I_q the scheduling identifier $s_q s_p$, where s_q is the sequence number of the message containing I_q, and s_p is the scheduling identifier of the parent, if any, of I_q.

For the example of Figure 4, the MT-domain schedulers assign the identifiers s_1, $s_2 s_1$ and $s_3 s_2 s_1$ to the invocations I_1, I_2 and I_3, respectively. In this case, s_1 and s_2 (s_3) are (is) uniquely assigned by the scheduler at every replica of the MT-domain X_1 (Y_1). Furthermore, the same unique identifiers are generated within every replica of the MT-domain X_1 (Y_1) because these identifiers are derived, in an identical manner, from the totally ordered messages that the operation scheduler at each replica receives.

To detect a reentrant incoming invocation, the operation scheduler uses the scheduling identifier to determine if the invocation is a descendant of any operation that has been invoked, and is awaiting a response within the MT-domain. For instance, the scheduler at X_1 detects that invocation I_3 is a descendant of I_1 due to the presence of I_1's identifier s_1 in I_3's identifier $s_3 s_2 s_1$. Once the scheduler verifies that an operation is indeed a descendant, it waits for the currently executing thread-of-control within the MT-domain to suspend itself, and then dispatches the descendant operation.

4.3 Scheduling Algorithm

To dispatch an operation to the thread-of-control, or to delay an operation that may lead to inconsistency, each operation scheduler for a MT-domain replica maintains:

- The scheduling identifier s_D, and semantics (synchronous or asynchronous), of the current operation I_D being executed by the logical thread-of-control T_D within the MT-domain. The scheduling identifier s_D is used by the operation scheduler to detect reentrant invocations. The scheduler compares the scheduling identifer of every incoming operation with s_D to determine any descendants of I_D. If an incoming message is a descendant of I_D, or a response to an invocation issued by the MT-domain, the operation scheduler dispatches it when all of the threads within the MT-domain are suspended, or awaiting a response.
- A dispatch queue Q_{op} of operations (invocations, responses and state transfer messages) waiting to be assigned to the thread of control when it becomes available. When the current operation I_D completes, the operation scheduler can dispatch a new operation from Q_{op}. Operations that are not related to the thread-of-control in the MT-domain are enqueued in the total order in which the operation scheduler receives them. Operations that are descendants of I_D are scheduled for execution, in the order of their arrival, ahead of all operations that are not descendants of I_D.
- A stack of the reentrant descendant operations that have already been dispatched onto threads within the MT-domain, and are awaiting responses. When the thread-of-control becomes available, the stack is empty. The first operation I_D to be pushed onto the stack is the one that assumes the thread-of-control T_D. Subsequently, descendants of I_D that are reentrant on the MT-domain are also pushed onto the stack. The invocation on top of the stack is removed from the stack as soon as it completes, and an invocation is pushed onto the stack as soon as it is dispatched to a thread within the MT-domain.

```
switch (Reason for Activation)

 // The thread-of-control for the MT-domain is
 // available to be assigned to a new operation.

 case THREAD_OF_CONTROL_RELEASED:
  if (operation queue $Q_{op}$ is not empty)
   $I_D$ = operation at the head of $Q_{op}$
   $s_D$ = scheduling identifier for $I_D$
   if (thread pool $Q_{thr}$ is empty)
    Create new threads into $Q_{thr}$
   endif
   $T_D$ = first available thread in $Q_{thr}$
   Dispatch operation $I_D$ onto the thread $T_D$
   Push $I_D$ onto stack of reentrant descendant invocations
  endif
  return
 endcase

 // A new operation intended for the MT-domain is
 // delivered in the totally ordered messages.

 case INCOMING_INVOCATION_OR_RESPONSE:
  Insert incoming message at the end of $Q_{op}$
  return
 endcase

 // The thread-of-control for the MT-domain is
 // suspended on the operation $I_D$. In addition to $T_D$,
 // numDesc threads could be suspended due to incomplete
 // reentrant descendant operations of $I_D$. All of these
 // threads are awaiting responses.

 case THREAD_OF_CONTROL_SUSPENDED:
  if (dispatch queue $Q_{op}$ is not empty)
   if (dispatch queue $Q_{op}$ has descendants of $I_D$)
    $I_{Top}$ = reentrant descendant at the top of the stack
    Increment numDesc
    $I_{numDesc}$ = first enqueued descendant of $I_{Top}$ in $Q_{op}$
    if ($I_{numDesc}$ completes operation $I_{Top}$)
     Remove $I_{Top}$ from the stack of operations
    else
     Push $I_{numDesc}$ onto the stack of operations
    endif
    $i_{numDesc}$ = scheduling identifier for $I_{numDesc}$
    if (thread pool $Q_{thr}$ is empty)
     Create new threads into $Q_{thr}$
    endif
    $T_{numDesc}$ = first available thread in $Q_{thr}$
    Dispatch operation $I_{numDesc}$ onto thread $T_{numDesc}$
   endif
  endif
  return
 endcase

endswitch
```

Figure 5: Algorithm executed by the operation scheduler each time it is activated. The operation scheduler is associated with a MT-domain D, whose logical thread-of-control T_D executes the operation I_D with scheduling identifier s_D.

- A thread pool Q_{thr} of pre-spawned threads to avoid the overhead of thread creation with every new dispatch of an operation to a thread. This thread pool is used purely for efficiency, rather than for correctness.

The operation scheduler at every replica of a MT-domain executes the deterministic algorithm, shown in Figure 5, to schedule operations within the replica that it controls. The execution of this algorithm is triggered by the occurrence of any of the following events:

- The release of the MT-domain's thread-of-control when the current operation completes, allowing the next operation to be dispatched
- The suspension of all threads within the MT-domain, in anticipation of receiving a response, allowing the delivery of a received response, or a new reentrant descendant invocation
- The delivery of a totally ordered invocation or response message to the operation scheduler, requiring the scheduler to decide if the message should be enqueued, scheduled or dispatched.

The scheduling algorithm, as well as the enforcement of the thread-of-control, are independent of the specific concurrency model (thread-per-request, thread-per-object, etc) adopted by the ORB that hosts the MT-domain.

5 Implementation in Eternal

As shown in Figure 6, the Eternal system transparently replicates the objects, and the MT-domains, of the application. For every replica of a MT-domain or an object, the Interceptor transparently captures its IIOP invocation and response messages, which were originally destined for TCP/IP, and diverts them instead to the Replication Manager. The Replication Manager performs the encapsulation (retrieval) of IIOP messages to (from) the messages of the underlying reliable totally ordered multicast group communication system. In addition, the Replication Manager implements mechanisms for the detection and suppression of duplicate invocations and duplicate responses, and for state transfer and recovery.

The Eternal system provides an *operation scheduler* within the address space of each replica of a MT-domain. Although each replica has its own scheduler, all of the schedulers for the replicas of a MT-domain reach identical scheduling decisions. This deterministic behavior of the schedulers ensures replica consistency for every MT-domain within the application.

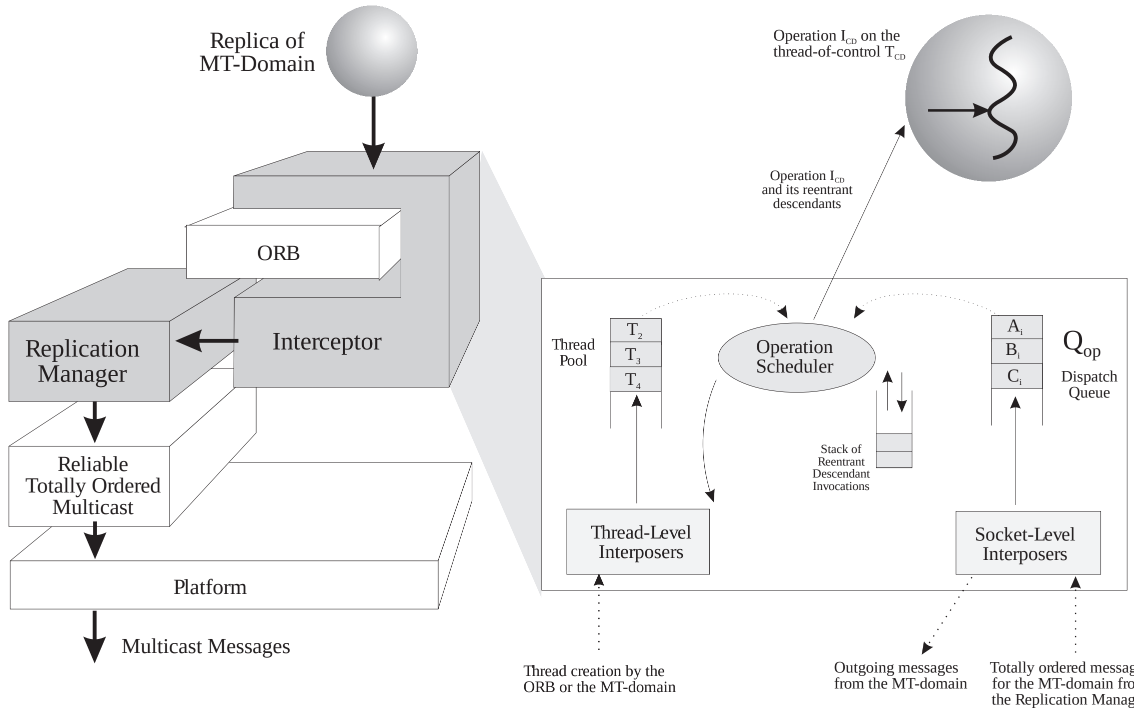

Replica of MT-Domain
ORB
Replication Manager
Interceptor
Reliable Totally Ordered Multicast
Platform
Multicast Messages
Operation I_{CD} on the thread-of-control T_{CD}
Operation I_{CD} and its reentrant descendants
Thread Pool
T_2
T_3
T_4
Operation Scheduler
A_i
B_i
C_i
Q_{op}
Dispatch Queue
Stack of Reentrant Descendant Invocations
Thread-Level Interposers
Socket-Level Interposers
Thread creation by the ORB or the MT-domain
Outgoing messages from the MT-domain
Totally ordered messages for the MT-domain from the Replication Manager

Figure 6: Implementation of the MT-domain operation scheduler in the Eternal system, using the Interceptor which is transparently co-located with the replica of the MT-domain.

The operation scheduler must operate at a level that allows it to govern the concurrency within each object of the MT-domain, irrespective of the ORB's multithreading policies. The operation scheduler must receive all of the totally ordered operations, and decide on their delivery to the application. In the Eternal system, the operation scheduler is implemented at the level of the Interceptor, as shown in Figure 6.

The Interceptor of the Eternal system exploits "hooks" [8] provided by standard operating systems to allow for the runtime modification of process behavior. A library interpositioning-based implementation allows the Interceptor to be mapped transparently into the address space of a MT-domain at runtime, which allows the Interceptor easy access to the sockets and threads that the MT-domain creates.

The transparency of the Interceptor has the added advantage that the operation scheduler can perform its function without modification of the ORB or the application. The interception mechanisms also enable the scheduler to overide any dispatching or threading performed by the ORB or the application, without either of them being aware of the scheduler's existence. The operation scheduler exploits a number of interposers within the Interceptor, with each interposer overriding some specific behavior of the ORB or the application.

- **Socket-level interposers** "replace" the socket library routines that CORBA objects use to connect and communicate over TCP/IP. To ensure that the inter-MT-domain communication occurs over a reliable totally ordered multicast protocol (instead of over TCP/IP), the communicating MT-domains must establish a path of communication through the Replication Manager. The Interceptor's socket-level interposer serves to redirect all TCP/IP communication to the Replication Manager. The Replication Manager then conveys them over the underlying multicast protocol. By exploiting the socket-level interposer, the operation scheduler can receive, transparently, all of the operations destined for the replica of the MT-domain.

- **Thread-level interposers** "replace" the thread library routines that multithreaded ORBs and CORBA objects use to create and control threads. The thread library interposer does not need to replace all of the symbol definitions, for instance, within the Solaris thread library, `libthread.so`, or the POSIX thread library, `libpthread.so` [6] but to interpose only on the symbols of interest. The operation scheduler employs a thread-level interposer to provide alternative implementations of some of the thread library routines in order to enable the MT-domain's operaton scheduler to determine the status of the MT-domain's thread-of-control, as well as to control the creation, dispatch and destruction of threads spawned within the MT-domain.

Of course, not all of the threads that the MT-domain or the ORB creates need to be controlled. The listening thread that an MT-domain server first spawns must not be prevented from running because, otherwise, the MT-domain would not be able to function in its role as a server. However, the additional threads that are dispatched to handle client invocations must be controlled because they may modify the state of the MT-domain.

Thus, the operation scheduler must examine every IIOP message, that it receives through the totally ordered messages from the Replication Manager, to determine if it contains a method invocation or response. The combination of the socket-level and the thread-level interposers ensures that the dispatch of threads that execute operations within a MT-domain is dictated solely by the operation scheduler, rather than by the MT-domain or by the ORB.

Operation scheduling does not significantly increase the overheads in our implementation. All of the mechanisms of Eternal, including not only operation scheduling but also interception, replication, duplicate detection, message ordering and multicasting, increase the response time by about 10% over an unreplicated multithreaded application.

6 Related Work

Considerable research efforts have been expended in designing and implementing practical systems that employ strategies to enforce replica determinism, or to circumvent specific sources of nondeterminism.

The use of replication for fault tolerance requires replica determinism to ensure that no undesirable or unforeseen side-effects cause the states of the replicas of an object to become inconsistent. The Delta-4 project [12] employs a primary-backup, or passive, replication approach to overcome the problems associated with nondeterministic replicas. The restrictions required are rather severe.

For systems that must meet real-time requirements in addition to fault tolerance, the replicated data must be both consistent and timely. The fault-tolerant real-time MARS system [11] requires deterministic behavior in highly responsive applications, such as automotive applications, which exhibit nondeterminism due to time-triggered event activation and preemptive scheduling. Replica determinism is enforced using a combination of timed messages and a communication protocol for agreement on external events.

In the SCEPTRE 2 real-time system [1], nondeterministic behavior of the replicas also arises from preemptive scheduling. The developers of SCEPTRE 2 acknowledge the limitations of both active and passive replication of nondeterministic "capsules" for the purposes of ensuring replica consistency. Semi-active replication is used, with deterministic behavior enforced through the transmission of messages from a coordination entity to the multiple backup replicas for every nondeterministic decision taken by a designated primary replica. The messages force the backup replicas to override their own decisions.

Considerable research efforts have been directed towards building systems that augment CORBA with fault tolerance. While most of these systems exploit group communication to ensure replica consistency, to the best of our knowledge, none of them addresses the support that is required to ensure consistent object replication in the presence of nondeterminism such as multithreading in the ORB or the application.

The AQuA architecture [3] is a CORBA-based framework that provides fault tolerance to CORBA applications through object replication. The AQuA architecture exploits the group communication facilities and the ordering guarantees of the underlying Ensemble and Maestro toolkits [15] to ensure replica consistency for the application. The AQuA gateway translates CORBA object invocations into messages that are transmitted via Ensemble, and detects and filters duplicate invocations (responses).

The Object Group Service (OGS) [4] and GARF [5] provide fault tolerance, the former for CORBA applications. Replica consistency is enforced to some extent through the use of a consensus algorithm, which operates on the TCP/IP-based IIOP messages, to provide ordered message delivery to the application. However, the consensus algorithm controls the concurrency only within the protocol stack and the underlying layers; concurrency within the ORB and the application are not controlled.

Some of the issues surrounding replica consistency and multithreading have been addressed for fault-tolerant systems that are not based on CORBA. In [14], a technique is employed to track and record the nondeterminism due to asynchronous events and multithreading. While the nondeterminism is not eliminated, the nondeterministic executions are recorded so that they can be replayed to restore replica consistency in the event of rollback.

The Transparent Fault Tolerance (TFT) system [2] enforces deterministic computation on replicas at the level of the operating system interface. TFT sanitizes nondeterministic system calls by interposing a software layer between the application and the operating system. The object code of the application binaries is edited to insert code that redirects all nondeterministic system calls to a layer that returns identical results at all replicas of an object.

7 Conclusion

The Eternal system provides consistent object replication for multithreaded CORBA applications and ORBs. The basic unit of replication is an object or a MT-domain which is a CORBA client or server that can support concurrent ORB- or application-level threads. An Interceptor, implemented using thread- and socket-level interposers, allows the Eternal system to schedule and dispatch threads and operations onto replicas of MT-domains transparently, thereby overriding the scheduling of operations by the ORB.

Each replica of a MT-domain is equipped with an operation scheduler, which enforces a single logical thread-of-control within the replica. The operation scheduler dispatches operations onto the thread-of-control, and enqueues operations not related to the thread-of-control for later dispatch. Replica consistency is maintained as a result of the deterministic behavior across the replicas of a MT-domain through the totally ordered messages that they receive, as well as the deterministic behavior within each replica of the MT-domain through the identical scheduling of operations onto a single thread-of-control.

The Eternal system maintains replica consistency, irrespective of the multithreading model (thread-per-request, thread-per-object, etc) adopted by the ORB. The transparency of the operation scheduling and the replica consistency mechanisms enables Eternal to provide fault tolerance to unmodified multithreaded CORBA applications, without requiring the modification of either the ORB or its concurrency model.

References

[1] S. Bestaoui. One solution for the nondeterminism problem in the SCEPTRE 2 fault tolerance technique. In *Proceedings of the Euromicro 7th Workshop on Real-Time Systems*, pages 352–358, Odense, Denmark, June 1995.

[2] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the IEEE 28th International Conference on Fault-Tolerant Computing*, pages 128–137, Munich, Germany, June 1998.

[3] M. Cukier, J. Ren, C. Sabnis, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the IEEE 17th Symposium on Reliable Distributed Systems*, pages 245–253, West Lafayette, IN, October 1998.

[4] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[5] R. Guerraoui, P. Felber, B. Garbinato, and K. Mazouni. System support for object groups. *SIGPLAN Notices, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 33(10):244–258, October 1998.

[6] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. Prentice Hall, Mountain View, 1996.

[7] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998.

[8] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, pages 62–68, July 1999.

[9] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.2 edition. OMG Technical Committee Document formal/98-07-01, February 1998.

[10] Object Management Group. Fault tolerant CORBA using entity redundancy: Request for proposals. OMG Technical Committee Document orbos/98-04-01, April 1998.

[11] S. Poledna. *Replica Determinism in Fault-Tolerant Real-Time Systems*. PhD thesis, Technical University of Vienna, Vienna, Austria, April 1994.

[12] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[13] D. C. Schmidt. Evaluating architectures for multithreaded Object Request Brokers. *Communications of the ACM*, 41(10):54–60, October 1998.

[14] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the IEEE 26th International Symposium on Fault-Tolerant Computing*, pages 250–259, Sendai, Japan, June 1996.

[15] A. Vaysburd and K. Birman. The Maestro approach to building reliable interoperable distributed applications with multiple execution styles. *Theory and Practice of Object Systems*, 4(2):73–80, 1998.