Towards Dependable Real-time and Embedded CORBA Systems

Balachandran Natarajan, Chris D. Gill
{bala, cdgill}@cs.wustl.edu
Dept. of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130

Aniruddha S. Gokhale
a.gokhale@vanderbilt.edu
ISIS
Vanderbilt University
P O Box 36, Peabody
Nashville, TN 37203

Douglas C. Schmidt
schmidt@uci.edu
Dept. of Electrical and Computer Engineering
University of California
616E Engineering Tower
Irvine, CA 92697

And

Joseph K. Cross, Christopher Andrews, Sylvester J. Fernandez
{joseph.k.cross, christopher.andrews, sylvester.j.fernandez}@lmco.com
Lockheed Martin Tactical Systems
PO Box 64525, M S U2X26
St. Paul, MN 55164-0525

Abstract

Commercial off-the-shelf components (COTS) based on distributed object computing (DOC) middleware, such as CORBA, are increasingly being used to develop and deploy distributed applications rapidly and cost effectively. Conventional COTS middleware has been considered less suitable for mission-critical distributed real-time and embedded (DRE) applications that require support for multiple quality of service (QoS) properties, such as dependability, efficiency, and predictability. The CORBA Real-time and Fault-tolerance specifications individually address the issues of predictability and dependability, respectively. However, implementations of these specifications do not yet support DRE applications with stringent simultaneous dependability and predictability requirements.

This paper provides three contributions to the development of middleware services that simultaneously address dependability and predictability requirements of key classes of DRE applications, such as commercial or military avionics systems, that require a high degree of reliability and bounded latency even in the case of faults. First, we outline the QoS requirements of an important class of DRE applications that possess both stringent time/space constraints and high dependability needs. Second, we show that meeting DRE application dependability and timing requirements by naively applying the strategies in the existing CORBA specification is replete with contradictions and pitfalls. Finally, we propose and empirically evaluate a new strategy that enables the composition of semantically compatible strategies from the Real-time and Fault-tolerant CORBA specifications to support DRE applications more effectively.

Keywords: Fault-tolerant CORBA, Real-time CORBA, DRE systems, Dependability, Middleware protocols.

1 Introduction

The vast majority of computational cycles today are expended to control distributed real-time and embedded (DRE) systems, including commercial and military aircraft and satellites, automobile engines, chemical and manufacturing plants, and hospital patient monitoring equipment. Mechanical and human controls in these systems are being replaced by software controllers at a growing rate, which means that essential features of our lives and economy depend upon the quality and cost of DRE software. We therefore need technologies that can ensure DRE systems will work predictably, and will be diagnosable and repairable when they fail.

Due to constraints on weight, power consumption, memory footprint, and performance, DRE systems are harder to develop, maintain, and evolve than mainstream desktop and enterprise software. Moreover, the tools and techniques traditionally used to develop DRE software are often highly specialized. For example, DRE systems are commonly designed around fixed task schedules, where time is divided into a sequence of fixed-length frames at each processor, and the processor executes each of its tasks for a fixed interval within each frame. Such systems often use frame-based interconnect among the processors, so that the traffic on the interconnect is also entirely scheduled at system design time. Highly specialized software design tools have been built to support the development of such systems [1]. Such specializations make it hard to adapt traditional DRE software to meet new functional or QoS requirements, hardware/software technology innovations, or emerging market opportunities.

During the past decade, a substantial amount of R&D effort has focused on developing distributed object computing (DOC) *middleware* as a means to simplify the development and reuse of successful DRE systems. DOC middleware is systems software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware [2]. It offers clients portable language-independent and location-transparent invocation of methods on target object implementations [3]. DOC middleware simplifies DRE system development by off-loading the tedious and error-prone aspects of distributed computing from application developers to middleware developers.

During the past several years, DRE systems with hard real-time requirements [4, 5] have increasingly been developed with the OMG Common Object Request Broker Architecture (CORBA) [6]. CORBA is DOC middleware that provides run-time support to automate many distributed computing tasks, such as connection management, object (de)marshaling, object demultiplexing, language and OS independence, load balancing, fault-tolerance, and security. Certain QoS requirements of DRE systems, particularly dependability and predictability, are addressed individually by the OMG's Fault-tolerant [7] and Real-time CORBA [8] specifications, respectively.

Unfortunately, implementations of these specifications do not yet support mission-critical DRE systems, such as shipboard combat control systems and avionics mission computing systems, that require support for these QoS properties simultaneously. These types of DRE systems are typified by the following characteristics:

- **Stable applications** – Most DRE systems have a longer life than their commercial counterparts, which requires that the infrastructure for DRE systems provide stable interfaces [9]. This in turn provides DRE systems the flexibility to modify the underlying infrastructure as long as the interfaces remain compatible.
- **End-to-end timeliness and dependability requirements** – DRE systems have stringent latency and dependability requirements. The latency bounds are commonly expressed in response to external events, whereas dependability requirements are often expressed as a probabilistic guarantee that the requirements will be met.
- **Heterogeneity** – DRE systems often run on a wide variety of computing platforms that are interconnected by different types of networking technologies. The efficiency of execution of the different infrastructure components on which the DRE systems operate varies as the type of computing platform and interconnection technology.

Simultaneously providing dependability and predictability properties for the class of DRE systems outlined above is hard since the combination of these properties is often in conflict. For example, any CORBA middleware infrastructure that offers dependability could spend a non-deterministic amount of time detecting and recovering from faults. This in turn conflicts with the bounds on latency for message invocation since the CORBA infrastructure must account for the time spent on fault detection and recovery. Consequently, providing both these QoS requirements simultaneously requires a careful blend of protocols, patterns, and design constraints and tradeoffs, which transcends the present capabilities of commercial off-the-shelf (COTS) middleware.

Our prior research on CORBA middleware has explored the efficiency, predictability, scalability and dependability aspects of ORB endsystem design, including static [10] and dynamic [11] scheduling, event processing [12], I/O subsystem [13] and pluggable protocol [14] integration, synchronous [15] and asynchronous [16] ORB Core architectures, systematic benchmarking of multiple ORBs [17], optimization principle patterns for ORB performance [18] and high-performance architectures for Fault-tolerant CORBA [19, 20]. This paper focuses on another dimension in the ORB endsystem design space: *providing dependability using Fault-tolerant CORBA (FT-CORBA) to Real-time CORBA (RT-CORBA)-based DRE systems.*

This paper is organized as follows: Section 2 provides a brief overview of the RT-CORBA and FT-CORBA specifications; Section 3 describes key challenges that must be resolved when designing dependable DRE systems using CORBA; Section 4 proposes and empirically evaluates a new technique that addresses the key challenges outlined in Section 3 when designing dependable DRE systems; Section 5 discusses open issues and future search directions; and Section 6 presents concluding remarks.

2 Overview of Real-time and Fault-tolerant CORBA specifications

This section provides a brief overview of the RT-CORBA and FT-CORBA specifications – detailed descriptions of these standards appear in [21] and [19], respectively.

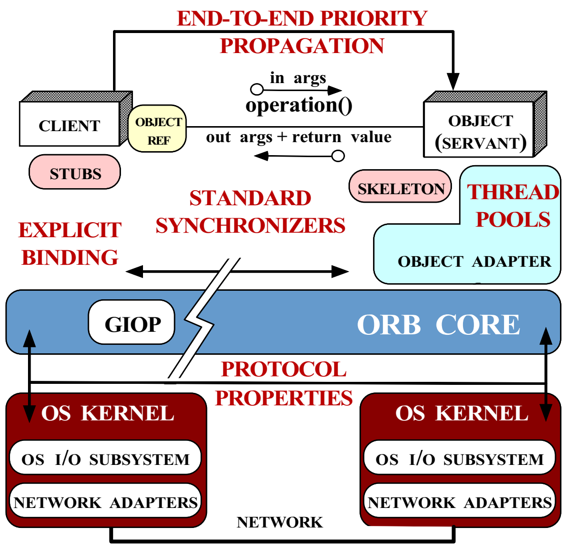

Figure 1: ORB Endsystem Features for Real-Time CORBA

Overview of Real-time CORBA: Figure 1 depicts an ORB endsystem [10] comprising network interfaces, operating system I/O subsystems, and communication protocols, and CORBA-compliant middleware components and services. The RT-CORBA specifications identify capabilities that must be *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, peer-to-peer) integrated and managed by ORB endsystems to ensure end-to-end predictable behavior for *distributable threads*[1] that traverse from one object to another to complete operations.

To manage these capabilities, vertically and horizontally, RT-CORBA defines standard interfaces and QoS policies that allow applications to configure and control the following resources:

- *Processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service
- *Communication resources* via protocol properties and explicit bindings and
- *Memory resources* via buffering requests in queues and bounding the size of thread pools.

Applications typically specify these real-time QoS policies along with other policies when they call standard ORB operations, such as `create_POA` or `_validate_connection`. For instance, when an object reference is created using a QoS-enabled POA, the POA ensures that any server-side policies that affect client-side requests are embedded within a *tagged component*[2] in the object reference. This enables clients that invoke operations on such object references to honor the policies required by the target object.

Overview of the Fault-Tolerant CORBA Specification: The FT-CORBA [6] defines a standard set of interfaces, policies, and services that provide robust support for applications requiring high reliability. The

[1]A distributable thread is a programming model abstraction, that can execute operations on objects without any regard for any physical node boundaries. It is a schedulable entity having its own scheduling parameters such as priorities and deadlines, which specify an acceptable end-to-end timeliness guarantees for completing the sequential execution of operations in multiple object instances residing on multiple physical nodes.

[2]Tagged components are name/value pairs that can be used to export attributes, such as security or QoS values, from a server to its clients within object references [22].

fault tolerance mechanism used to detect and recover from failures is based on *entity redundancy*. Consequently, in FT-CORBA the redundant entities are replicated CORBA objects.

Replicas of a CORBA object are created and managed as a "logical singleton" [23] composite object. Figure 2 illustrates the key components in the FT-CORBA architecture. All components shown in the figure are implemented as standard CORBA objects, *i.e.*, they are defined using CORBA IDL interfaces and implemented using servants that can be written in standard programming languages, such as Java, C++, C, or Ada.

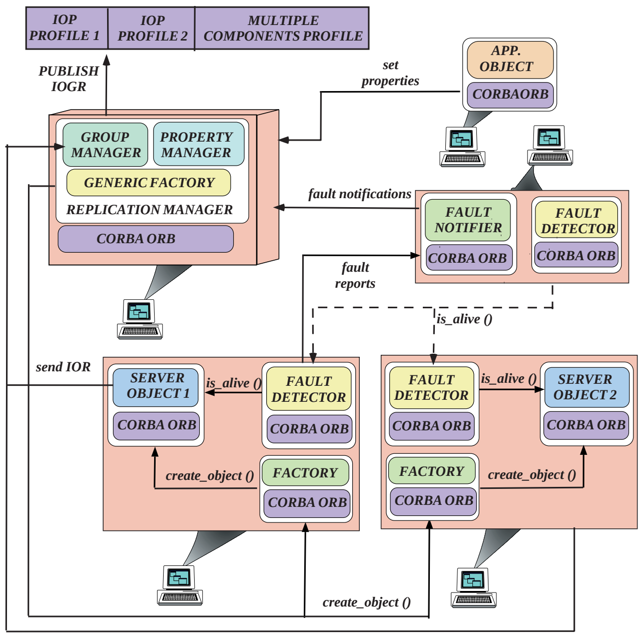

Figure 2: The Architecture of Fault Tolerant CORBA

3 Challenges Designing Fault-Tolerant and Real-time CORBA-based DRE Systems

The DRE system characteristics described in Section 1 motivate the integration of implementations of RT-CORBA and FT-CORBA as the infrastructure for DRE systems. This middleware provides open standard interfaces that simplify the development of DRE systems requiring dependability and predictability. As we discuss below, however, their combined use in today's ORBs lacks certain features and have semantically incompatible strategies that make them unsuitable for important classes of DRE systems. The remainder of this section describes the challenges associated with integrating the RT-CORBA and FT-CORBA specifications to deliver QoS requirements to DRE systems.

3.1 Challenge 1: Non-determinism and Expensive Replication Strategies

Context: Application objects use replication to achieve transparent fault tolerance. The FT-CORBA specification specifies the COLD_PASSIVE, WARM_PASSIVE, ACTIVE, and ACTIVE_WITH_VOTING replication styles to tolerate faults transparently. With the exception of COLD_PASSIVE, these replication styles require that replicas maintain consistent state.

Problem: In COLD and WARM_PASSIVE systems, the recovery time needed to switch to a backup replica can be unacceptably high for DRE systems with stringent timing constraints. Likewise, in an ACTIVE system the cost associated with providing totally ordered reliable multicast and the time needed to synchronize via proprietary or group communication mechanisms can be unacceptable.

Using an ACTIVE replication for applications based on "push-pull" architectures, such as the CORBA Event Service [12], can introduce non-determinism when trying to handle multiple events that must be managed. Moreover, the replicas in an ACTIVE system must behave identically and deterministically if they are used in DRE systems. Special negotiations (which impose high overheads) are needed to enforce the order of execution of messages among all the replicas. Preemption of requests, such as responding to an alarm condition, in an ACTIVE configuration can make the system non-deterministic, which can be avoided only via complex protocols that consume significant system resources.

The FT-CORBA specification also requires strong replica consistency, which for ACTIVE replication objects means that all members of the group must have the same state at the end of each method invocation. It is conceivable, however, that application objects required to be fault tolerant will also make outcalls, such as reading the real-time clock or responding to arbitrary external events. Under these circumstances, it is unlikely that consistent state can be maintained across the replicated objects, which suggests that a conforming implementation must severely constrain the ways in which objects can interact with their environment.

In Section 4 we propose a new replication strategy that provides the following benefits

- Fast and deterministic failure detection times
- Deterministic state synchronization strategy that eliminates the need for protocols with high overhead and
- No restriction placed on the application on ways that it can interact with the environment.

3.2 Challenge 2: Dealing with Semantic Incompatibilities Between RT-CORBA and FT-CORBA Features

Context: Requirements on DRE systems are commonly expressed in terms of external stimuli and responses. For example, consider a tactical display system that uses radars as sensors and that presents an operator with a graphical representation of the geographical area, including the present locations of moving objects such as missiles. In such systems, a common requirement is *radar to glass in one second*. This requirement means that at most one second may elapse between a radar pulse bouncing off the surface of a missile and the corresponding observation being displayed to the operator.[3]

Similarly, dependability requirements are often expressed in terms of the probability that one or few of the many requirements will fail to hold over a specific period of operation. DRE systems like the one outlined above require simultaneous stringent QoS properties, including predictability and dependability, for their correct operation.

Problem: Although combining the power of RT- and FT-CORBA seems a promising approach, the requirements on DRE systems illustrate a significant semantic gap between end-to-end predictability and dependability requirements and capabilities, such as propagating priorities and replicating server objects. The ability to engineer a good fault tolerant solution requires tradeoffs that may compromise a DRE system's ability to support real-time behavior, and vice versa. For example, a non-trivial amount of communication and processing costs are incurred to accomplish failover from one replica to another, and this presents a hard choice between accounting for failover in every message transmission versus preparing for time budgets to be violated during failover.

Fundamentally, the variation in performance and latency inherent in an elaborate FT solution is antithetical to the predictable behavior required in a DRE system. The effect of choices made to support a requirement in one area can have complex and unforeseen consequences in the other. For DRE systems to leverage the advantages of open, standard interfaces, therefore, a solution that can mask the semantic incompatibility between FT-CORBA and RT-CORBA solutions is needed.

Schemes like [24, 25, 26] have been developed and deployed that resolve the conflict between real-time and fault-tolerance capabilities of the system. We propose and evaluate a strategy in Section 4 that provides bounded fault-tolerance capabilities while trying to maintain the real-time properties, which is an essential condition for the development for dependable DRE systems.

3.3 Challenge 3: Lack of Standards to Handle Byzantine and Partial Failures

Context: A system is considered reliable if it does not fail to perform its designated function. However, failure is often measured in ways that generally allow for a non-zero probability that the system will in fact fail over some long enough period of time. Moreover, partial failures are also a common occurrence in distributed systems. For example, a component interacting with another component cannot distinguish whether a delay in response is due to failure of the remote component or due to a slow or partitioned network.

Problem: The FT-CORBA model for failure detection and recovery emphasizes a certain type of failure, namely component failure, which is also called a "crash failure." In this type of failure the individual component ceases

[3]The above requirement is still a real-time requirement, even though the timeliness requirements are expressed in seconds.

all interactions with its environment. The policies and detection mechanisms in FT-CORBA, such as the use of heartbeats and timeouts, implicitly acknowledge only this limited view.

A more subtle form of failure is one where interactions among components cause the system to fail. A case in point is where a corrupted component requests services more frequently than allowed for in the design, denying other components access to critical resources. Heartbeats and time-outs cannot protect against this type of failure. Given the inability to detect this difference, the requirement that objects using ACTIVE replication maintain consistent state in a bounded time between method invocations may, in the general case, be infeasible to achieve.

We discuss the future work that is needed to resolve this challenge in Section 5.

3.4 Challenge 4: Lack of QoS Semantics

Context: For designers of DRE systems, the primary benefit of an open standard is the promise of a stable interface between the application and the services provided by the middleware since this simplifies porting to a different service implementation. In particular, when the service being provided is part of the infrastructure—and the infrastructure is built with COTS products—system designers are highly motivated to consider both the cost of upgrades and penalties associated with COTS obsolescence [9]. It is therefore important that DRE systems interact with the infrastructure through syntactically and semantically stable interfaces, such as those defined in the RT-CORBA and FT-CORBA specifications. CORBA also helps improve the portability and interoperability of applications built using such interfaces.

Problem: The RT-CORBA and the FT-CORBA specifications provide the mechanisms by which real-time and fault-tolerant behavior can be achieved for DRE systems. CORBA does not, however, guarantee that two implementations that conform to the RT-CORBA and FT-CORBA specification will provide equivalent semantic behavior.

For example, assume that a system implemented on ORB$_A$ compliant with the RT-CORBA and FT-CORBA specifications successfully meets the requirement that sensor data from the navigation subsystem be propagated to all interested recipients at 35ms intervals, even when certain faults occur in the system. It is highly unlikely, however, that if ORB$_A$ were replaced by ORB$_B$, the 35 ms propagation interval will be maintained, even if ORB$_B$ implements the RT-CORBA and FT-CORBA specifications. To establish that the modified system continues to meet its requirement would require thorough testing, possible reengineering, and expensive recertification. It is this absence of a QoS standard that makes the existing RT-CORBA and FT-CORBA specifications inadequate for certain types of DRE systems.

Moreover, the implementation freedom that arises from an interface specification means that performance characteristics of various compliant products may vary greatly. Such performance variations can spell disaster when ORB$_A$ is replaced by a different, yet compliant, ORB$_B$. For instance, one product may choose to use an IPC mechanism, such as shared memory, to communicate between objects collocated on the same computer, whereas another may pass the message through a TCP loopback pipe. These two mechanisms can produce radically different levels of performance. Replacing or updating an ORB may therefore require costly redesign, reimplementation, and revalidation of DRE systems.

We discuss the future work that is needed to resolve this challenge in Section 5.

3.5 Challenge 5: Lack of Standard End-to-end QoS Configurability

Context: DRE Systems are frequently distributed and heterogeneous. The heterogeneity spans different types of computers used in the system and different interconnection technologies used to connect these computers. Some computers may be fast and/or provide good QoS guarantees, whereas others may be slow and/or provide poor QoS guarantees. The difficulties associated with location transparency, distribution, providing backups to tolerate faults and controlling QoS calls for a standard infrastructure, such as CORBA, that can hide many of the heterogeneity from the DRE systems developer.

Problem: Though the RT-CORBA and the FT-CORBA specifications define some standard QoS specification mechanisms, they do not allow application developers to express certain requirements, such as the one outlined in Section 3.2, directly. The RT-CORBA 1.0 specification, for example, provides no standard control over end-to-end latency, focusing primarily on processing priorities. Priorities are useful for DRE systems since they provide the proper processing order of method calls within the distributed application. Priorities alone, however, do not address en-route message passing latency issues. To affect network latencies, more control is needed at the transport and lower layers. The RT-CORBA specification, in keeping with the OMG's implementation-independent philosophy, leaves the lower level control issues to ORB providers.

For a variety of reasons, an RT-CORBA provider might choose a transport mechanism (*e.g.*, UDP, TCP, or multicast), with a "best effort" level of message transport priority for all messages, despite their processing priorities. This is an area for which the DRE system architects need precise control. Unfortunately, mechanisms for exercising such options are proprietary today due to lack of specificity in the CORBA specification. For instance, setting DiffServ Differentiated Services Code Point (DSCP) bits on an IP packet via software may not be possible in a portable manner across CORBA implementations, but may be quite important to the correct functioning of a DRE system.

We discuss the future work that is needed to resolve this challenge in Section 5.

4 Empirical Evaluation of New Techniques

Section 3 outlined the challenges of using FT-CORBA and RT-CORBA together to build DRE systems. Though the challenges outlined in Sections 3.3, 3.4 and 3.5 must be addressed to built robust, reliable and long-running DRE systems, the challenges outlined in Sections 3.1 and 3.2 must be addressed first, since they are more fundamental and form the basis by which the others can be addressed. This section presents a strategy that can potentially address the issues raised in Sections 3.1 and 3.2. This strategy can guarantee real-time and dependability characteristics without having the overhead and non-determinism associated with the fault tolerance strategies outlined in the FT-CORBA specification.

4.1 The Semi-Active Replication Style

The SEMI-ACTIVE replication style is based on the European Delta-4 (XPA) architecture [24] (where this term was coined), which we have adapted and applied to CORBA-based DRE systems. This replication style is designed to have some of the benefits of both the active replication and passive replication styles, including predictable fail over times and deterministic behavior during program execution. Figure 3 illustrates how the replicas are arranged to tolerate faults in the systems. The key

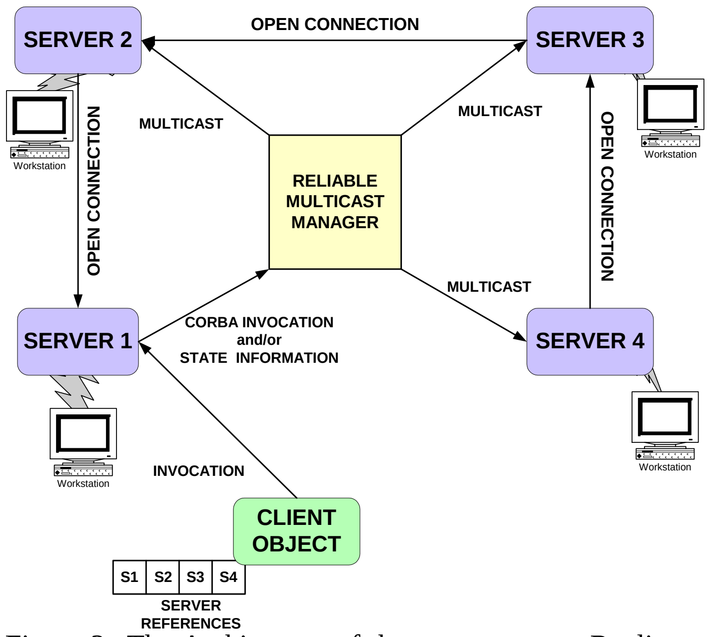

Figure 3: The Architecture of the SEMI-ACTIVE Replication Style

features of this architecture are outlined below:

- The replicas are arranged as a linked list of nodes with each replica connected through a transport-level connection to the one ahead in the queue.
- The linked list of replicas is created at startup time.
- The replica at the head of the list is designated the primary.
- When the primary fails, the next secondary replica in the list is promoted to become the primary.
- Failures are detected by the next replica in the list when transport-level connections close.
- The promotion of the secondary to the primary is done by the secondary when it detects a failure.

- All invocations to the primary are reliably multicast by the primary to the secondaries, such that secondaries consume messages in the same order the primary consumes, maintaining replica consistency.
- Replica consistency can also be maintained by reliably multicasting state synchronization messages to all replicas.
- The multicast protocol that is used for request invocation or state transfer needs to enforce message ordering. A simple reliable model that ensures data delivery alone is not sufficient.
- Applications can choose to use either one or both the synchronization strategies outlined above to maintain replica consistency.
- An ordered list of references is passed to the client, which must honor the order of the list.

The SEMI-ACTIVE replication style resolves the following challenges with the existing mechanisms described in Section 3.1 and 3.2:

- Faster and predictable failure detections ensures deterministic recovery times when compared to COLD_PASSIVE and WARM_PASSIVE replication.
- No need for a totally ordered reliable multicasting for ACTIVE replication style and other protocols having heavy overhead to enforce identical behavior or message processing order across replicas.
- Reduced heartbeat and poll messages on the network since they are not used for detecting failures.
- No restrictions imposed on the applications for using the underlying middleware infrastructure.

Despite the advantages mentioned above that make the SEMI-ACTIVE replication style a potential candidate for use in DRE systems, it does have the following disadvantages:

- If a non-primary replica in the list fails, the replica just following the failed replica could declare itself as a primary and wait for messages to be processed, which can potentially partition the list. To prevent partitioning requires a remote token manager to disseminate tokens to replicas and promote them as primaries in a deterministic manner.
- When a primary fails, the FT-CORBA model does not place any restriction on the client's choice of a backup from the list of replica references to make the invocation. The SEMI-ACTIVE replication style could restrict the client to use the list of references as an ordered list, which is not compliant with the FT-CORBA spec.
- Applications choosing to multicast state information instead of multicasting the requests must use interceptor mechanisms to handover the state information to the subsystem doing the multicasts. Installing the interceptor is an additional responsibility for developers.[4]

Fortunately, the disadvantages outlined above do not affect the SEMI-ACTIVE replication style's determinism, which makes it a good candidate for use in DRE systems. The remainder of this section empirically evaluates some of the properties of this replication style in the context of a client making continuous invocations to a server. Our experiments assumed a single-failure model with no nested failures. The faults occuring in our experiments are assumed to be fail-silent, *i.e*, after failure they have no interaction with the environment.

4.2 Empirical Evaluation of the Semi-Active Replication Style

This section describes the results of empirical benchmarking studies we conducted to measure how well the SEMI-ACTIVE replication style can provide real-time and fault-tolerance support to DRE systems. A key goal in conducting these benchmarks is to show the determinism in

1. The *detection time* to detect failures
2. The *response time* required for clients to connect to a new primary if an existing primary fails and
3. The *synchronization time* required to synchronize the state during every invocation.

To evaluate, we built several tests that demonstrate specific use cases for these benchmarks. The tests were based on ACE [27] and TAO [10], versions 5.2.2 and 1.2.2, respectively. The tests were run on a single endsystem – a 930 MHz Pentium III processor with 512 MB

[4]There is a class of DRE applications whose object state changes are triggered by occurrence of events, such as time triggers or alarm conditions, rather than from CORBA requests.

RAM running 2.4.9 of the Linux kernel in the FIFO real-time scheduling class.

4.2.1 Measuring Failure Detection Time on the Server-side

Rationale. We define the *failure detection time* on the server as the time taken to detect a failure. For the SEMI-ACTIVE replication style, this includes the time taken by the secondary to detect its connection to the primary is closed after the failure of the primary. This value is important since it represents the time taken by the middleware infrastructure to detect and react to faults.

Methodology. A server that uses Acceptor and Connector [28] framework components in ACE was used to model the replicas. The primary waits for input requests from the client on a specified port and sends a response to the client to mark the end of an invocation. In addition to waiting for connections from the client, the secondary also connects to the primary, as shown in Figure 3. The client establishes connections to all the replicas since this reduces the jitter due to connection establishment during failover.

A script invokes the primary and all the replicas. We then allow a client to connect to the primary replica and invoke remote operations. At this point, we invoke the server object's `shutdown` operation, which crashes the primary. This crash initiates a detection process in the secondary, which promotes itself to the primary and waits to receive invocations.

We measure the failure detection time as the time between the failure of the primary replica and the time when the secondary actually detects a failure. To measure the failure detection time, we recorded two time stamps:

- The first time stamp was recorded when the primary was killed.
- The second timestamp was recorded when the secondary detected its connection to the primary was closed.

We conducted several iterations of this experiment by killing the primary replica at randomly selected times.

Failure detection time. Figure 4 shows the variation of detection times over several iterations, the overall average detection time, and the upper and lower bounds on the detection time.

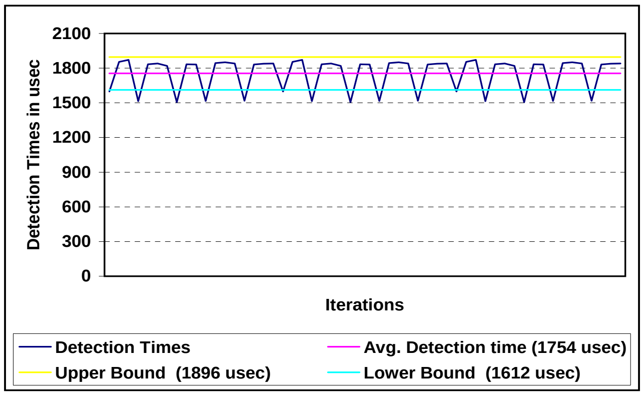

Figure 4: Average and Bounds on Failure Detection Time within the Object Group

Analysis. Figure 4 indicates the following:

- Failure detection is in the millisecond range
- The bounds are approximately ±4-5% of the average.

The smaller detection times and its boundedness within a 5% range are properties needed for DRE systems, as opposed to ± 100% with the traditional heartbeat and polling styles described in [20].

4.2.2 Measuring Fault Detection and Recovery Times on the Client-side

Rationale. A client invoking a remote operation will experience some delay if its server fails during the operation. This delay has three parts:

1. The time taken by the infrastructure to detect the fault
2. The time taken by the infrastructure to promote a backup to become the primary and
3. The time taken by the client to detect a failed primary and make invocations on the secondary.

Below, we describe the experiment conducted to measure the combination of these times, which is the actual delay experienced by a client. This time actually indicates the bounds of the latency that the client will experience when faults occur.

Methodology. The experimental setup is similar to the one described in Section 4.2.1. To measure the effect of failures—and to compute the total recovery time—we allow the client to shutdown the primary by invoking the server object's `shutdown` operation.

We measure the failure detection time as the time interval between when the client invoked the `shutdown` operation to the time when the client can make the next request to the secondary. The time interval includes the error handling and the time needed for the client to retrieve an established connection and make the request to the secondary.

Failure detection time. Figure 5 shows the variation of detection times over several iterations, the overall average detection time, and the upper and lower bounds on the detection time.

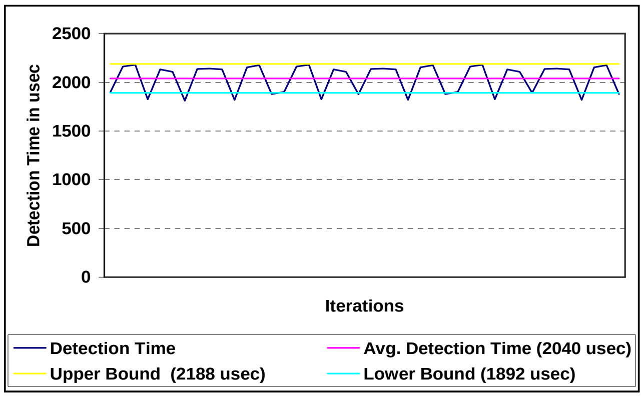

Figure 5: Average and Bounds on Failure Detection Time by the Client

Analysis. Figure 5 shows behavior similar to the one described in section 4.2.1. The difference in the average detection times of the server and client indicate the infrastructure costs associated in detecting and recovering from the failure on the client.

A common feature observed between Figures 4 and the 5 is the periodic nature of the detection times. This periodicity stems from the fact that we repeated the same experiment a number of times to collect the data. The graphs are drawn from the ordered data.

4.2.3 Latencies from State Transfer

Rationale. A client invoking a remote operation will experience some delay if the server multicasts the requests or multicasts the state updates reliably to all the replicas, in addition to executing the invocation on the primary. Below, we describe the experiment conducted to measure the combined time, which is the actual delay experienced by a client for every invocation. This experiment measures the latency experienced by the client when making invocations on an object group possessing state synchronization capabilities.

Methodology. Rather than modeling a communication subsystem that makes invocations to all the secondaries, we used TAO's Real-time Event Channel [12] to propagate state information to all the replicas with every invocation. We chose TAO's Real-time Event Channel for the following reasons:

1. The Event Channel offers a "push-pull" communication model, where all the registered event suppliers can publish events of interest to registered consumers.
2. TAO's Event Channel has been used in dozens of production DRE systems.

The primary in the SEMI-ACTIVE replication style acts as a supplier of events to the channel and all the replicas subscribe to the channel as consumers to receive events. To add reliability to the delivery of events to the channel through the `push` operation, we set the *reliable one-way* policy SYNC_WITH_SERVER policy at the ORB level. This policy ensures the thread invoking the `push` operation only returns after it receives a confirmation from the remote ORB as a reply.

We measured the time the client takes to make every invocation on the remote object. We varied the number of replicas receiving state information and captured the minimum, maximum, and average times. We also calculated the upper and lower bounds associated with this.

Latencies from State Transfer. Figure 6 shows the variation of minimum, average, and maximum latency associated with communicating with primaries with varying number of replicas in the configuration outlined above. Figure 7 shows the average, upper, and lower bounds on the latency in the same experiments.

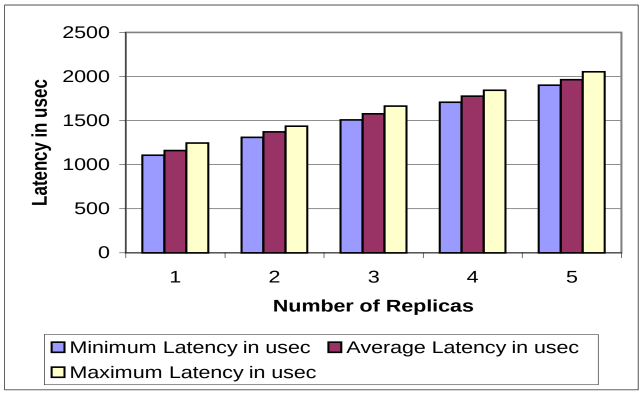

Figure 6: Latency on the Client with Increase in Number of Replicas

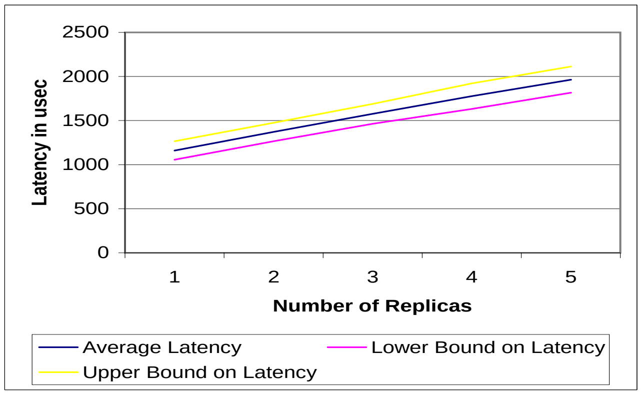

Figure 7: Bounds on Latency with Increase in Number of Replicas

Analysis. Figure 6 and 7 indicate the following:

1. The latencies increasing with the number of replicas as observed by the client and
2. The bounds on latencies being maintained with an increase in the number of replicas.

The increase in latency occurs since we use reliable oneways as opposed to a regular CORBA oneway call. Adding reliability to message transmission to every replica incurs additional overhead, as indicated by the results. A key engineering challenge is therefore striking the right balance between the degree of replication and the affordable latency reduction that DRE application developers can afford.

4.3 Summary of Results and Recommendations

Based on the results presented above, we now describe some of the key challenges that the SEMI-ACTIVE replication strategy resolves when designing dependable DRE systems. Though the empirical evaluation is made in the context of dependable DRE systems, this strategy is applicable to a larger class of distributed applications requiring dependability support from the middleware.

Challenge 1. A non-trivial amount of time is spent by the FT middleware to detect and recover from faults, which is antithetical to the latency and performance requirements of DRE systems.

Resolution. The SEMI-ACTIVE replication style that we propose places less overhead on the infrastructure to detect and recover from faults. Similar to the distributed computing paradigm which distributes computation between different nodes, the SEMI-ACTIVE replication style distributes the overhead of fault detection and recovery between replicas. The detection and recovery times from the SEMI-ACTIVE replication style are bounded allowing the possibility to accomplish critical real-time tasks even in the presence of faults.

Challenge 2. Synchronizing message processing order across all replicas deterministically using the ACTIVE replication style calls for the usage of protocols with high overhead.

Resolution. The SEMI-ACTIVE replication imposes the primary's order of message execution to all the replica in the group. The primary decides on the message order based on the local real-time QoS parameters and imposes that on all the replicas. The primary could either choose to multicast the request chosen for processing to all the replicas or choose to multicast the state information at the end of request processing. This flexibility is particularly useful for a class of DRE systems where the state of the system changes with external events in addition to CORBA requests.

Challenge 3. The ACTIVE replication style in order to maintain replica consistency could constrain the way in which application objects interacts with their environment, like preventing execution threads from doing tasks that could change object states not associated with CORBA request.

Resolution. The SEMI-ACTIVE replication resolves this challenge by allowing the primary to send state updates to all its secondaries in addition to CORBA requests from the clients. This gives applications the necessary flexibility to schedule and dispatch tasks that could affect the state of the object.

Finally, we present an observation and a recommendation based on the empirical evaluation of the SEMI-ACTIVE replication style.

Observation. Client latencies tend to increase as a function of an increasing degree of replication.

Recommendation. Middleware researchers and implementors should carefully study application use cases, failure rates of DRE applications, and expected performance from the system to determine the replication degree automatically. Being able to configure the degree of replication adaptively would help simplify the development and deployment of DRE systems. Likewise, DRE system developers should carefully evaluate the tradeoffs associated with increased replication degrees on the performance of their systems.

5 Future Directions

While the work presented above shows significant progress in solving the problems presented in Sections 3.1 and 3.2 above, the problems of Sections 3.3, 3.4 and 3.5 stand as open challenges to the middleware R&D community.

Byzantine and partial failures. The detection, diagnosis, and response to failures other than crash failures is an exceedingly hard problem to address in an application-independent manner. For example, consider an object that is returning correct responses too slowly. The fault may be in the component, in the connection with the component, or in some otherwise unrelated component that is sharing some resource with the slow component. These issues have received intense and protracted study [29]. Perhaps the most promising direction for the middleware fault-tolerant community would be to provide interfaces through which application-specific detection, diagnosis, and response mechanisms can act.

Comparable QoS benchmarks. The ability to compare qualities of service across different infrastructure implementations requires an agreement on the set of qualities that apply to each service, and rigorous definitions of those qualities. For example, if "invocation latency" is agreed to be a relevant quality of a client-server service, then it must be determined exactly how this value will be measured. Since any set of measurements of such qualities will then present a distribution of values, the useful statistics that it must be determined exactly how this value will be measured. Since any set of measurements of such qualities will present a distribution of values, the useful statistics that are to be derived from such a distribution must be agreed on, *e.g.*, worst-case, or average and standard deviation. It would also be helpful to have standard benchmarking suites for these qualities.

Standard interfaces for QoS control. It is clearly desirable that DRE applications be able to control the end-to-end qualities of service that it receives by standard, implementation-independent, mechanisms. But as the DSCP example given in Section 3.5 shows, implementation dependence at least in the implementation of QoS control mechanisms is probably inescapable. It may, however, be possible to specify a standard, implementation-independent, interface to such QoS control mechanisms. Such an interface would probably strongly resemble the QoS specifications discussed in the preceding paragraph.

Although solving all the problems presented above is clearly hard, the consequences of solving them would be profound: it would be possible to construct DRE systems that reliably meet their functional and quality requirements, and do so when operating on any sufficiently powerful infrastructure.

6 Concluding Remarks

Distributed real-time and embedded (DRE) systems are playing an increasingly important role in many application domains, including telecommunication networks (*e.g.*, high-speed central office switching), telemedicine (*e.g.*, remote surgery), manufacturing process automation (*e.g.*, hot rolling mills), and aerospace (*e.g.*, avionics mission computing). Although there are many types of DRE systems, they have one thing in common: *the right answer delivered too late becomes the wrong answer*. Providing the right answer at the right time is therefore imperative for mission-critical DRE systems.

Our effort for adding dependability to DRE systems focuses on developing and deploying strategies, such as the one explained in Section 4, that can provide timeliness and performance guarantees to the application even during occurence of crash faults or fail-silent faults. Our goal in this effort is to provide *the right answer at the right time* by lowering the infrastructure overhead needed to detect and recover from faults.

References

[1] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[2] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering*, John Marciniak and George Telecki, Eds. Wiley & Sons, New York, 2001.

[3] Michi Henning and Steve Vinoski, *Advanced CORBA Programming With C++*, Addison-Wesley, Reading, Massachusetts, 1999.

[4] Ralph Lachenmaier, "Open Systems Architecture Puts Six Bombs on Target," `http://www.cs.wustl.edu/~schmidt/TAO-boeing.html`, Dec. 1998.

[5] Douglas C. Schmidt, "R&D Advances in Middleware for Distributed, Real-time, and Embedded Systems," *Communications of the ACM special issue on Middleware*, vol. 45, no. 6, June 2002.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 edition, Dec. 2001.

[7] Object Management Group, *Fault Tolerant CORBA Specification*, OMG Document orbos/99-12-08 edition, December 1999.

[8] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 edition, March 1999.

[9] Joseph K. Cross and Douglas C. Schmidt, "Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware," in *Patterns and Skeletons for Distributed and Parallel Computing*, Fethi Rabhi and Sergei Gorlatch, Eds. Springer Verlag, 2002.

[10] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[11] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, no. 2, March 2001.

[12] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, Atlanta, GA, October 1997, ACM, pp. 184–199.

[13] Fred Kuhns, Douglas C. Schmidt, and David L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, Vancouver, British Columbia, Canada, June 1999, IEEE, pp. 154–163.

[14] Carlos O'Ryan, Fred Kuhns, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference.* ACM/IFIP, Apr. 2000.

[15] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[16] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, Michael Kircher, and Jeff Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference.* ACM/IFIP, Apr. 2000.

[17] Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, Stanford, CA, August 1996, ACM, pp. 306–317.

[18] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*, San Diego, CA, May 1999, USENIX.

[19] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the* 2^{nd} *International Symposium on Distributed Objects and Applications (DOA 2000)*, Antwerp, Belgium, Sept. 2000, OMG.

[20] Balachandran Natarajan, Aniruddha Gokhale, Douglas C. Schmidt, and Shalini Yajnik, "Applying Patterns to Improve the Performance of Fault-Tolerant CORBA," in *Proceedings of the* 7^{th} *International Conference on High Performance Computing (HiPC 2000)*, Bangalore, India, Dec. 2000, ACM/IEEE.

[21] Douglas C. Schmidt and Fred Kuhns, "An Overview of the Real-time CORBA Specification," *IEEE Computer Magazine, Special Issue on Object-oriented Real-time Computing*, vol. 33, no. 6, June 2000.

[22] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 edition, Oct. 2000.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.

[24] P. Barrett, A. Hilborne, P. Bond, D. Seaton, P. Verssimo, L. Rodrigues, and N. Speirs, "The Delta-4 Extra Performance Architecture (XPA)," in *Proceedings of the 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, 1990.

[25] Kane Kim and Subbaraman C, "PSRR: A Scheme for Time-Bounded Fault Tolerance in Distributed Object-Based Systems," in *Proc. IEEE High-Assurance Systems Engineering (HASE) Workshop*, Ontario, Canada, Oct. 1996, IEEE.

[26] Kane Kim and Subbaraman C, "Fault-Tolerant Real-Time Objects," *Communications of the ACM*, Jan. 1997.

[27] Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*, Addison-Wesley, Boston, 2002.

[28] Douglas C. Schmidt and Stephen D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, Massachusetts, 2002.

[29] Miguel Castro and Barbara Liskov, "Practical Byzantine Fault Tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, Feb. 1999.