Trade-Offs Between Real-Time and Fault Tolerance for Middleware Applications *

Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213-3890
priya@cs.cmu.edu

Abstract

The OMG's Real-Time CORBA (RT-CORBA) and Fault-Tolerant CORBA (FT-CORBA) specifications make it possible for today's CORBA implementations to exhibit either real-time or fault tolerance in isolation. While real-time requires *a priori* knowledge of the system's temporal operation, fault tolerance necessarily deals with faults that occur unexpectedly, and with possibly unpredictable fault recovery times. When both real-time and fault-tolerance are required to be satisfied within the same system, it is rather likely that trade-offs are made during the composition. The contribution of this paper is the identification of the conflicts between real-time and fault tolerance.

1 Introduction

Middleware platforms, such as CORBA and Java, are increasingly being adopted because they simplify application programming by rendering transparent the low-level details of networking, distribution, physical location, hardware, operating systems, and byte order. Since CORBA and Java have come to incorporate support for many "-ilities" (e.g., reliability, real-time, security), these middleware platforms have become even more attractive to applications that require a higher quality of service.

. For CORBA middleware, there exist the Fault-Tolerant CORBA [20] and the Real-Time CORBA [21] specifications that aim to provide fault tolerance and real-time, respectively, to CORBA applications. The push for Commercial-Off-The-Shelf (COTS) products, along with the recent support for "-ilities" within middleware, have furthered the adoption of middleware within mission-critical applications.

Despite its many attractive features, middleware still does not quite support applications that have *multiple simultaneous* quality-of-service (QoS) requirements, in terms of their reliability and real-time. It is simply not possible today for a CORBA application to have *both* real-time and fault-tolerant support through the straightforward adoption of implementations of the Real-Time and Fault-Tolerant CORBA standards, primarily because the two specifications are incompatible with each other.

To some extent, this is because the real-time and fault tolerance standards for CORBA were developed independently of each other, and cannot be readily reconciled. In reality, though, this is a manifestation of a much harder research problem – the fact that real-time and reliability are system-level properties (i.e., properties that require a more holistic consideration of the distributed system, and not just of components or objects in isolation) that are not easy to combine because they often impose conflicting requirements on the system.

Real-time operation requires the application to be predictable, to have bounded request processing times, and to meet specified task deadlines. Typically, for a CORBA application that is required to be real-time, the behavior of the application, in terms of the actual time and frequency of client invocations, the relative priorities of the various invocations, the worst-case execution times of the invocations at the server, and the availability and allocation of resources for the application's execution, must be known ahead of run-time. Armed with this information, the real-time CORBA infrastructure then computes a schedule ahead of run-time, and the application executes according to this predetermined schedule. Because every condition has been anticipated, and appropriately planned for, the system behaves predictably. This predictability is often the single most important characteristic of real-time systems.

On the other hand, fault-tolerant operation requires that the application continue to function, even in the presence of unanticipated events such as faults, and potentially time-consuming events such as recovery from faults. For a CORBA application, fault tolerance is typically provided through the replication of the application objects,

*The author wishes to acknowledge partial support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

and the subsequent distribution of the replicas across different processors in the system. The idea is that, even if a replica (or a processor hosting a replica) crashes, one of the other replicas of the object can continue to provide service. Because it is not sufficient for a truly fault-tolerant system merely to detect the fault, most fault-tolerant systems include some form of recovery from the fault. For a fault-tolerant CORBA system, recovery is likely to occur through the launching of a new replica, and its subsequent reinstatement to take the place of one that crashed. Of course, this implies the ability to restore the state of the new replica to be consistent with those of currently executing replicas of the same object. The consistency of the states of the replicas, under fault-free, faulty and recovery conditions, is often the single most important characteristic of fault-tolerant systems.

Thus, there exists a fundamental difference in the philosophy underlying the two system properties of real-time and fault tolerance. While real-time requires *a priori* knowledge of the system's temporal operation, fault tolerance is built on the principle that faults can, and indeed do, occur unexpectedly, and that faults must be handled through some recovery mechanism whose processing time is usually unknown in advance. While inconsistency in state across replicated entities is detrimental to fault-tolerant behavior, missed task deadlines are detrimental to real-time behavior. When both real-time and fault tolerance are required to be satisfied within the same system, it is rather likely that trade-offs are made during their composition. For instance, the consistency semantics of the data might need to be traded against timeliness, or vice-versa.

Combining reliability and real-time is further complicated by the fact that each system property has its own range of "values". For instance, reliability can vary from weakly-consistent replication (where the availability of a server is more important, even if the data in some replicas is somewhat stale or inconsistent) to strongly-consistent replication (where the availability of the server and the integrity/consistency of the data across all replicas are equally important). The composition of real-time behavior and reliability must take into account the varying degrees of both real-time and reliability support, and the effective quality-of-service (QoS) that results.

This work looks at the difficult issues in providing support for both fault tolerance and real-time in a distributed asynchronous system. CORBA is chosen as the vehicle for initial investigation because CORBA currently incorporates separate real-time and fault tolerance standards within its specifications. Section 2 provides the necessary background, including the specifications of the Fault-Tolerant CORBA and the Real-Time CORBA standards, respectively, as they exist today. Section 3 outlines the conflicts that we have identified between real-time and fault-tolerant operation. Section 4 looks at existing approaches to real-time fault-tolerant systems. Section 5 concludes with the insights that we have gained from our preliminary experiments, along with possible strategies for resolving the conflicts that we have identified.

The work in this paper represents an essential foundational step in the research necessary to develop real-time fault-tolerant middleware. The major contribution of this paper is the identification of the conflicts between real-time and fault tolerance.

2 Background

The Common Object Request Broker Architecture (CORBA) [19] middleware supports applications that consist of objects distributed across a system, with client objects invoking server objects that return responses to the client objects after performing the requested operations. CORBA's Object Request Broker (ORB) acts as an intermediary in the communication between a client object and a server object, transcending differences in their programming language (language transparency) and their physical locations (location transparency). The Portable Object Adapter (POA), a server-side entity that deals with the actual implementations of a CORBA server object, allows application programmers to build implementations that are portable across different vendors' ORBs. CORBA's General Internet Inter-ORB Protocol (GIOP) and its TCP/IP-based mapping, the Internet Inter-ORB Protocol (IIOP), allow client and server objects to communicate regardless of differences in their operating systems, byte orders, hardware architectures, etc.

The following section describes the Fault Tolerant CORBA and the Real-Time CORBA standards, both of which are optional sets of extensions to CORBA that enhance ORBs with support for fault tolerance and real-time, respectively. In this section, we present the standards as they exist; neither the Fault-Tolerant CORBA standard [20] nor the Real-Time CORBA standard [21] addresses, or intended to address, how real-time and fault tolerance impact each other.

2.1 Fault-Tolerant CORBA (FT-CORBA)

Current fault-tolerant CORBA systems can be classified into the integration, service or interception approaches. The integration approach to fault-tolerant CORBA incorporates the reliability mechanisms into the ORB infrastructure, resulting in modified, non-standard, CORBA implementations. Examples of the integration approach include Electra [10], AQuA [3], Maestro [28] and Orbix+Isis [7]. The service approach to fault-tolerant CORBA provides reliability through a collection

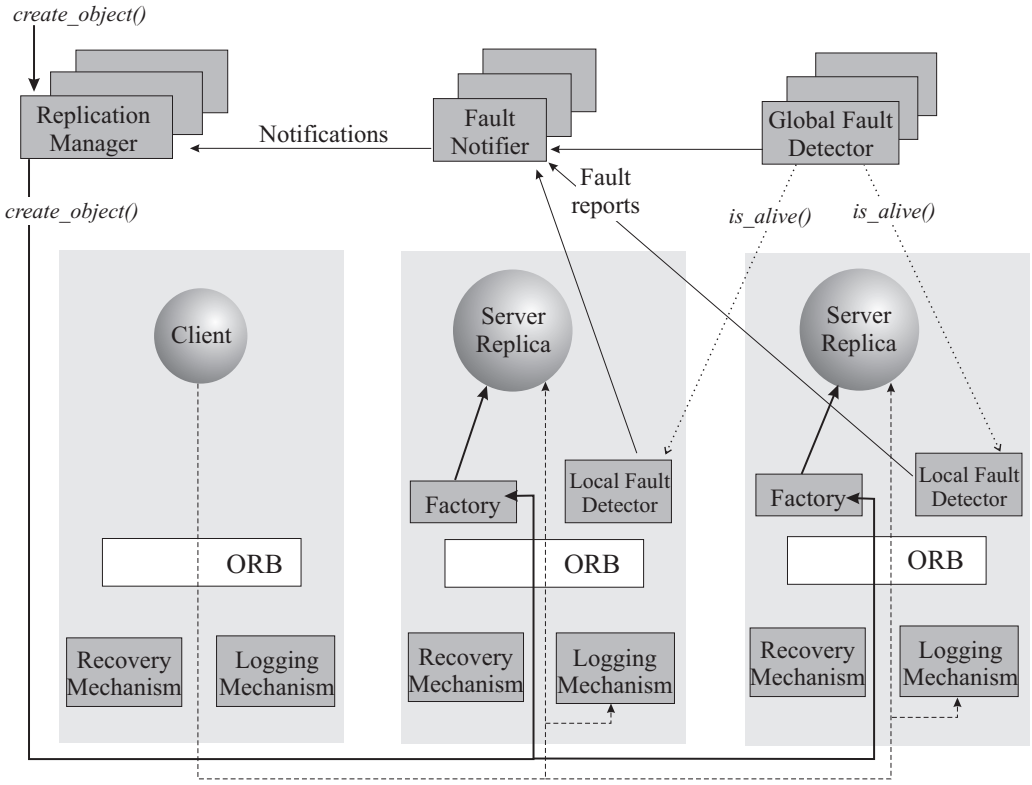

Figure 1: Architecture of the Fault-Tolerant CORBA (FT-CORBA) standard, showing support for a client-server application with two-way replication of the server.

of CORBA objects which are explicitly used by the application. Examples of the service approach include OGS [5], FRIENDS [4], DOORS [17], IRL [11], NewTOP [14] and FTS [6]. The interception approach to fault-tolerant CORBA provides reliability as a transparent add-on to existing CORBA applications by means of an interceptor that can add new services to existing ORBs. The Eternal [15] and the Immune [16] systems provide transparent reliability and survivability, respectively, to unmodified CORBA applications running over unmodified ORBs using the interception approach.

The Object Management Group formally adopted a standard specification for fault-tolerant CORBA (FT-CORBA) in March 2000. The FT-CORBA specification provides reliability through the replication of CORBA objects, and the subsequent distribution of the replicas of every object across the processors in the system. Figure 1 shows the various components of the FT-CORBA infrastructure, and their interaction with the supported CORBA application. The Replication Manager handles the creation, the deletion and the replication of both the application objects and the infrastructure objects. The Replication Manager replicates objects, and distributes the replicas across the system, and allows the user to configure an object's fault tolerance properties, including:

- **Factories** – processors on which replicas are to be created,
- **Minimum Number of Replicas** – the number of replicas that must exist for the object to be sufficiently protected against faults, also known as the degree of replication,
- **Checkpoint Interval** – the frequency at which the state of the object is to be retrieved and logged for the purposes of recovery,
- **Replication Style** – stateless, actively replicated, cold passively replicated or warm passively replicated.
- **Fault Monitoring Interval** – interval between successive "pings" of the object for liveness.

The FT-CORBA infrastructure provides support for fault detection and notification. The Fault Detector is capable of detecting host, process and object faults. Each application object inherits a `Monitorable` interface to allow the Fault Detector to determine the object's status. The Fault Detector communicates the occurrence of faults

to the Fault Notifier. The Fault Detectors can be structured hierarchically, as shown, with the global replicated Fault Detector triggering the operation of local fault detectors on each processor. Any faults detected by the local fault detectors are reported to the global replicated Fault Notifier.

On receiving reports of faults from the Fault Detector, the Fault Notifier filters them to eliminate any inappropriate or duplicate reports, and then distributes fault-event notifications to interested parties. The Replication Manager, being a subscriber of the Fault Notifier, receives reports of faults that occur in the system, and can, therefore, initiate appropriate recovery actions.

The Logging and Recovery Mechanisms are located underneath the ORB, in the form of non-CORBA entities, on each processor that hosts replicas. They are intended to capture checkpoints of the application, and to store them for the correct restoration of a new replica. Each application object inherits a `Checkpointable` interface to allow its state to be retrieved and assigned, for the purposes of recovery.

2.2 Real-Time CORBA (RT-CORBA)

The Real-Time CORBA (RT-CORBA) specification [21], with compliant implementations like TAO [25], ORBexpress, e*ORB and VisiBroker, aims to facilitate the end-to-end predictability of activities in the system, and to allow CORBA developers to manage resources and to schedule tasks. The current standard supports only fixed-priority scheduling. However, there exists a specialized CORBA specification for dynamic scheduling [22], as a part of Real-Time CORBA, version 2.0.

As shown in Figure 2, the specification includes a number of components, each of which must be designed or implemented by the RT-CORBA vendor to be predictable. The components include the real-time ORB (RT-ORB), the real-time POA (RT-POA), the mapping of the ORB-level priorities to the operating system's native priorities, and the server-side thread pool. In addition to the core CORBA infrastructural enhancements, the specification also includes a Real-Time CORBA Scheduling Service for the offline scheduling of the application's tasks, typically in accordance with the proven Rate Monotonic Analysis algorithm [9]. Using design-time information, such as the associations between activities, objects, priorities and resources, the Scheduling Service selects the appropriate CORBA priorities, priority mappings and POA policies to achieve a uniform real-time scheduling policy at run-time.

A server-side threadpool is used to avoid the run-time overhead of thread creation. The threadpool contains a number of pre-spawned threads, of which is selected when a task is required to be dispatched by the application. The threadpool abstraction provides interfaces for preallocating threads, partitioning threads, bounding thread usage, and buffering additional requests. A threadpool can be created with lanes, with each lane containing threads at a specific RT-CORBA priority.

RT-CORBA deals with, and supports the management of, three kinds of resources: process, storage and communication resources. Process resources are handled by providing for control over threadpools, for the assignment of threads to objects, and for the assignment of priorities to threads. Storage resources are appended to thread pools for creating additional threads, in response to more concurrent run-time requests than originally anticipated. Communication resources are managed by providing control over which transport connections are shared, and at what priorities they operate.

RT-CORBA uses a thread as a schedulable entity; threads form a part of what is known as an activity. Activities are scheduled through the scheduling of their constituent threads. The application can attach priorities to threads for scheduling purposes. RT-CORBA supports a platform-independent priority scheme designed to transcend the heterogeneity of the operating system-specific priority schemes. A `NativePriority` type represents the priority native to a specific operating system, and a `PriorityMapping` interface, shown in Figure 2 allows the real-time ORB to translate the application-specified RT-CORBA priority onto the operating system's native priority. The characteristics of a thread can be manipulated via the threadpool creation and the `RTCORBA::Current` interfaces. The `RTCORBA::Current` interface provides access to the RT-CORBA priority of a thread. There also exists a user-defined priority transform whereby the priority of an invocation can be modified even as the invocation is being processed at the server. A server can process a client's invocation at a specific priority based on two different models. In the client-propagated priority model, the client specifies the priority for the invocation, and the server honors this priority. In the server-declared priority model, the server specifies the priority at which it will execute the invocation. A client can communicate with a server over multiple different priority-banded connections, $i.e.$, with each connection handling invocations at a different priority. To improve the predictability of the system, clients are allowed to set timeouts to bound the amount of time that they wait for a server's response.

3 Conflicts Between Real-Time and Fault Tolerance

Typically, for a CORBA application that is required to be real-time, the behavior of the application, in terms of

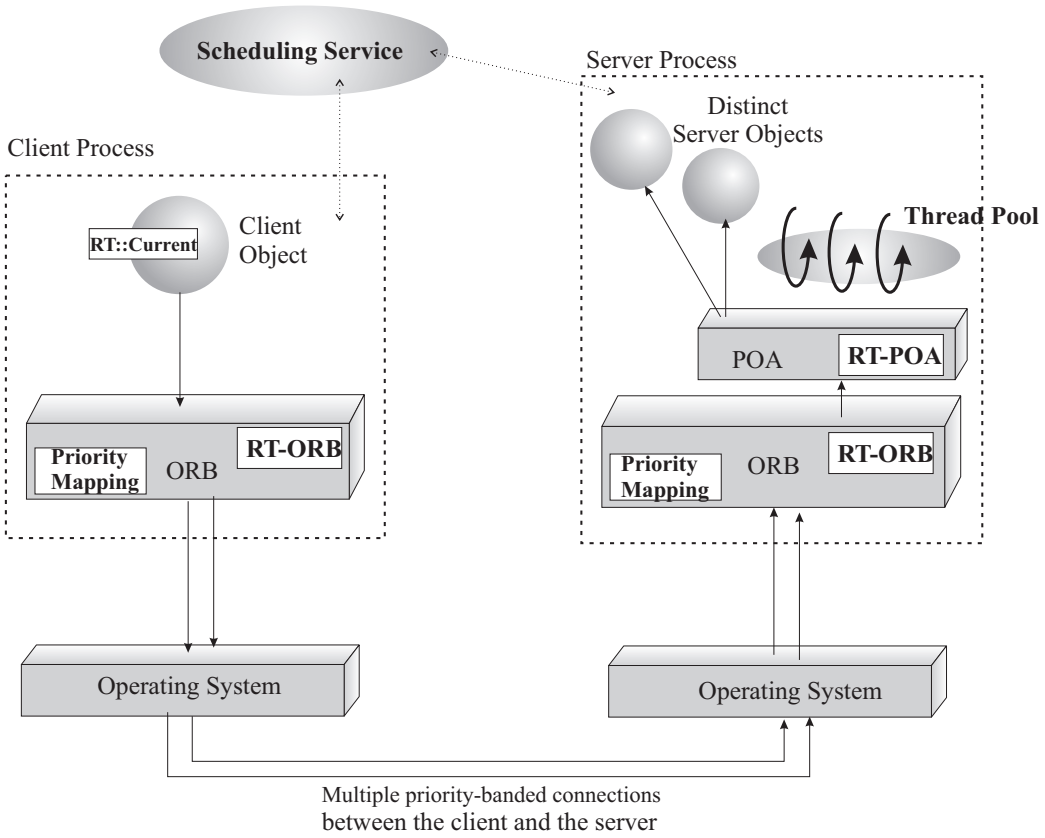

Figure 2: Architecture of the Real-Time CORBA (RT-CORBA) standard, showing support for a client-server application, including the real-time priority mapping, the RT-ORB, the RT-POA, the server-side thread pool, the client-side RT::Current and the Scheduling Service.

the actual time and frequency of client invocations, the relative priorities of the various invocations, the worst-case execution times of the invocations at the server, and the availability and allocation of resources for the application's execution, must be known ahead of run-time. Armed with this information, the real-time CORBA infrastructure can then compute an offline schedule ahead of run-time, and the application can then execute according to this predetermined schedule. Because every condition has been anticipated, and appropriately planned for, the system behaves predictably.

Fault tolerance for a CORBA application is typically provided through the replication of the application objects, and the subsequent distribution of the replicas across different processors in the system. The idea is that, even if a replica (or a processor hosting a replica) crashes, one of the other replicas of the object can continue to provide service. Because it is not sufficient for a truly fault-tolerant system merely to detect the fault, most fault-tolerant systems include some form of recovery from the fault.

End-to-end temporal predictability of the application's behavior is the single most important property of a real-time CORBA system. *Strong replica consistency*, under fault-free, faulty and recovery conditions, is often the single most important characteristic of a fault-tolerant CORBA system. The rest of this section outlines the real-time $vs.$ fault tolerance challenges that the proposed research will address.

3.1 Non-Determinism

The real-time and fault tolerance communities disagree even on the definition of terms such as determinism. From a fault tolerance viewpoint, an object is said to be deterministic if any two of its replicas (on the same processor or on different processors), when starting from the same initial state and executing the same set of operations in the same order, reach the same final state. If an object did not exhibit such reproducible behavior, one could no longer maintain the consistency of the states of its replicas. For a real-time system, an object is said to be deterministic if its behavior is bounded, from a timeliness

standpoint. End-to-end predictability for a fixed-priority CORBA system typically implies that thread priorities of the client and server are respected in resource contention scenarios, that the duration of thread priority inversions are bounded, and that the latencies of operation invocations are bounded.

Thus, for an application to be deterministic in terms of both real-time and fault tolerance, the application's behavior must be reproducible and identical across multiple replicas distributed across different processors; in addition, the application's tasks must be predictable and bounded from a temporal standpoint.

For CORBA applications, fault-tolerant determinism can be achieved by forbidding the application's use of any mechanism that is likely to produce different results on different processors. This includes a long list of items, such as local timers, local I/O, hidden channels of communication (such as non-IIOP communication), multithreading, etc. Real-time determinism can be satisfied by ensuring that the application's tasks are bounded in terms of processing time. Trivial CORBA applications can clearly satisfy the notions of determinism from both real-time and fault tolerance perspectives. For more realistic applications, it is often not possible to satisfy both determinism requirements.

One classic example is the use of time. Real-world time is a fundamental building-block in real-time systems, with absolute and precise time required everywhere to enable timers, timeouts, deadlines, as well as to bound and to time application behavior. RT-CORBA applications may set a timeout on a specific invocation in order to bound the time that the client is blocked waiting for a reply. The timeout mechanism is used to improve the predictability of the system. However, it contributes to fault-tolerant non-determinism because different replicas of the same object, on different processors, might obtain different values of time from their local processor clocks. Typically, this is resolved through some kind of global clock synchronization. Of course, the global time service must also be replicated (to avoid it being a single point of failure) and must not be a bottleneck (to avoid it being inefficient). This merely leads to a new set of problems.

3.2 Ordering of Operations

Real-time and fault tolerance both require the application's operations to be ordered, but for different reasons. From an real-time viewpoint, the most important criterion is to order the application's task in order to meet deadlines. When the same application is replicated for fault tolerance, the most important criterion is to keep the replicas consistent in state, even as they receive invocations, process invocations, and return responses. This requires delivering the same set of operations, in the same order, to all of the replicas assuming, of course, that the application is deterministic.

Thus, fault tolerance requires operations to be ordered to preserve replica consistency, while real-time requires operations to be ordered to meet deadlines. The problem arises when the two orders of operations are in conflict. For example, consider a processor P1 hosting replicas of objects A, B and C while processor P2 hosts replicas of objects A, B and D. The schedules of operations on the two processors might depend on their respective local resource usage and resource limits. It is perfectly possible that P1's replica of A and P2's replica of A see different orders of operations based on the individual schedules on their respective processors. Also, if A's replica on P1 dies, leaving only replicas of B and D behind on processor P1, the order of operations at B's replicas on the two processors might start to differ. Real-time operation is sensitive to resource usage and resource availability: meeting operation schedules, given the distribution of replicas onto different processors and the occurrence of faults, can lead to replica inconsistency.

3.3 Bounding Fault Handling and Recovery

Detecting a fault is not necessarily an asynchronous event, particularly if periodic heartbeats, or "pings", are used for fault detection. On the other hand, a fault itself is an asynchronous event because it is aperiodic. The time to handle a fault might be non-trivial and unpredicatable, and depends on many factors – the source of the fault, the point in time (relative to the rest of the system's processing) that the fault occurs, the ramifications of the fault (on the rest of the system's processing), activities in the system that are collocated with the faulty object/process/processor, $etc.$ Thus, when a fault occurs, the entire schedule might be "upset" at having to deal with the fault. The time to detect the crash fault of an object might vary considerably, depending on the ongoing activities of the other objects within the failed object's containing process, on the amount of time it takes for the underlying protocol and the ORB to detect connection closure, on the load and memory of the processor hosting the failed object, $etc.$

The recovery of a new replica is yet another event that might "upset" the pre-planned schedule of events in a real-time system. For a fault-tolerant CORBA system, recovery is likely to occur through the launching of a new replica, and its subsequent reinstatement to take the place of one that crashed. Of course, this implies the ability to restore the state of the new replica to be consistent with those of currently executing replicas of the same object. The time to recover a new replica depends on various factors: (i) state-retrieval duration, i.e., the time to retrieve the state from an executing replica (which might

depend on the size of the object's state), (ii) state-transfer duration, i.e., the time to transfer this retrieved state to the new replica across the network, (iii) state-assignment duration, i.e., the time to assign the transferred state to the new replica (which might sometimes involve instantiating multiple internal objects at the new replica), and (iv) message-recovery duration, i.e., the time for the new replica to "catch up" on relevant events that might have occurred in the system even as the replica was undergoing recovery. The replica is considered to be fully recovered only after phases (i)-(iv) are completed.

The last contributing factor, the message-recovery duration, requires the fault-tolerant infrastructure to record all incoming messages that are destined for the new replica (while it is recovering), and feeding this ordered queue of logged messages to the new replica after the state transfer to the new replica has occurred. Based on the instant at which the fault occurred, and on the instant at which recovery is initiated, the recovery time can vary considerably. For instance, the message-recovery duration (iv) might be zero (if no new invocations arrive for the new replica during the state-retrieval, the state-transfer or the state-assignment phases) or might never become zero (if new invocations arrive faster than the replica is able to catch up with the previously enqueued ones, even after the state-assignment phase has completed. Potentially unbounded events, such as fault handling, logging and recovery, are anathema to a real-time system.

3.4 CORBA-Specific Issues

In the current state-of-the-art, replicas of the same object cannot be hosted over different real-time ORBs, or even over different real-time operating systems. Other CORBA-specific issues that are concerns for real-time or fault tolerance, or both are:
CORBA also supports an asynchronous/oneway invocation mode, where the client sends an invocation to the the server, and does not expect any response. Oneways are inherently unreliable, with even the ORB not knowing for certain when the oneway completes. Asynchronous message handling is used in real-time CORBA to make a relatively quick up-call to a server object, and then to allow the server object to complete actual processing asynchronously, along with other concurrent client requests.

In fault-tolerant CORBA applications, the completion of one operation is used to signal that the next operation may be safely delivered. Not knowing when an operation completes, and not knowing if the operation shares state with other concurrent requests on the same object, can lead to replica inconsistency. RT-CORBA uses threads as the entity for scheduling while FT-CORBA uses objects as the entity for replication. Since there can exist multiple threading models (thread-per-object, thread-per-connection, $etc.$) within an ORB, care must be taken to reconcile the object $vs.$ thread models. The RT-CORBA priority transform, which can change the priority of a request on the fly, is not idempotent. Therefore, executions of the transform at different replicas of the same object might produce different results.

Every replicated CORBA object is associated with application-level state, ORB/POA-level state and infrastructure-level state. As a part of this project, we have already begun to identify the additional pieces of information that a Real-Time ORB might store (and that would need to be recovered), over and above standard non-real-time ORBs. For instance, an RT-CORBA client can have multiple connections to a server, with each connection at a different priority. For instance, the mapping of the priority to the physical connection within the Real-Time ORB constitutes a part of the client-side ORB-level state.

4 Related Work

Middleware systems that provide purely real-time or fault tolerance have already been covered in Section 2.2 and Section 2.1, respectively. This section looks exclusively at current approaches to provide some combination of real-time and fault tolerance. Stankovic's work [27] recognizes the tension between real-time and fault tolerance, and looks at providing fault tolerance for real-time system through a planning-mode scheduler and an imprecise computation model. This model prevents timing faults by reducing the accuracy of the results in a graceful manner, $i.e.$, the accuracy of the results increase with additional computation or execution time.

The work in [12] describes a time-redundancy approach to solve the problem of scheduling real-time tasks, along with transient recovery requests, in a uniprocessor system. The slack-stealing scheduling approach is exploited to determine various levels of responsiveness, with each responsiveness level based on the slack available at a specified priority in the system, and on the criticality of the request. A recovery request is accepted if it can meet its precomputed deadline, while being serviced at an assigned responsiveness level. Otherwise, the recovery request is rejected.

ARMADA [1] is a set of communication and middleware services that provide support for fault-tolerance and end-to-end Guarantees for embedded real-time distributed applications. The two distinct aspects of this project include the development of a predictable communication service for QoS-sensitive message delivery. The second thrust of this project includes a suite of fault-tolerant group communication protocols with timeliness guarantees, and a timed atomic multicast. The system supports

real-time applications that can tolerate minor inconsistencies in replicated state. To this end, ARMADA employs passive replication where the backups' states are allowed to lag behind the primary's state, but only within a bounded time window. The Maruti [24] system aims to support the development and deployment of hard real-time applications in a reactive environment. Maruti employs a combination of replication for fault tolerance and resource allocation for real-time guarantees. In this system, the object model can be enhanced to specify timing requirements, resource requirements and error handling.

The MARS project [8] is aimed at the analysis and deployment of synchronous hard real-time systems. MARS is equipped with redundancy, self-checking procedures and a redundant network bus. It employs a static offline real-time scheduler, along with tools to analyze the worst-case execution time of the application. For predictable communication with timeliness guarantees, the time-triggered protocol (TTP) is used for communication within the system. The Delta-4/XPA architecture [2] extended the original Delta-4 system [23] work to reliable group communication support, with bounded latencies and loose synchrony, for real-time applications. A comparison of the MARS and Delta-4/XPA systems can be found in [13]. More recently, fault tolerant features have been added to the real-time CORBA implementation, TAO [18]. The work adopts the semi-active replication style pioneered by Delta-4/XPA in order to provide some guarantees of fault-tolerant determinism. The implementation currently supports single-threaded applications.

The Real-time Object-oriented Adaptive Fault Tolerance Support (ROAFTS) architecture [26] is sdesigned to support the adaptive fault-tolerant execution of both process-stuctured and object-oriented distributed real-time applications. ROAFTS considers those fault tolerance schemes for which recovery time bounds can be easily established, and provides quantitative guarantees on the real-time fault tolerance of these schemes. A prototype has been implemented over the CORBA ORB, Orbix, on Solaris.

5 Conclusion

The work in this paper represents an essential foundational step in the research necessary to develop real-time fault-tolerant middleware. The major contribution of this paper is the identification of the conflicts between real-time and fault tolerance. Although these contributions are described in the context of CORBA, they apply to any application that requires real-time and fault tolerance support simultaneously, in a distributed asynchronous environment.

From our preliminary empirical evaluation, we conclude that new mechanisms for fast fault detection and fast recovery are required for real-time fault-tolerant CORBA applications. Current CORBA implementations produce fault detection, fail-over and recovery times in the order of seconds, and show great variability, even under identical, reproducible conditions. Because the behavior of even a simple real-time CORBA application is not sufficiently bounded or predictable in the presence of faults, additional mechanisms need to be incorporated into the system to compensate for, and to smooth out, the unpredictability.

Our current research is focussed on the kinds of infrastructural support and mechanisms required to resolve the real-time $vs.$ fault tolerance conflicts, to support middleware applications that must have *both* real-time and fault tolerance.

References

[1] T. F. Abdelzaher, S. Dawson, W. chang Feng, F. Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. G. Shin, Z. Wang, H. Zou, M. Bjorkland, and P. Marron. ARMADA middleware and communication services. *Real-Time Systems*, 16(2-3):127–153, 1999.

[2] P. Barrett, P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs, and P. Verissimo. The delta-4 extra performance architecture (xpa). In *Proceedings of the Fault-Tolerant Computing Symposium*, pages 481–488, Newcastle, UK, June 1990.

[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] J. C. Fabre and T. Perennou. A metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach. *IEEE Transactions on Computers*, 47(1):78–95, 1998.

[5] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

[6] R. Friedman and E. Hadad. FTS: A high performance CORBA fault tolerance service. In *Proceedings of IEEE Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002.

[7] Isis Distributed Systems Inc. and Iona Technologies Limited. *Orbix+Isis Programmer's Guide*, 1995.

[8] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, February 1989.

[9] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):40–61, 1973.

[10] S. Maffeis. Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*, pages 135–146, Monterey, CA, 1995.

[11] C. Marchetti, M. Mecella, A. Virgillito, and R. Baldoni. An interoperable replication logic for CORBA systems. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Antwerp, Belgium, September 2000.

[12] P. Mejia-Alvarez and D. Mosse. A responsiveness approach for scheduling fault recovery in real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.

[13] S. Melro and P. Verssimo. Real-time and dependability comparison of delta-4/xpa and mars systems. Technical report, INESC Technical Report RT/09-92, University of Lisbon, Portugal, 1992.

[14] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and implementation of a CORBA fault-tolerant object group service. In *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Finland, June 1999.

[15] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Department of Electrical and Computer Engineering, University of California, Santa Barbara, December 1999.

[16] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 507–516, Austin, TX, May 1999.

[17] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, Antwerp, Belgium, September 2000.

[18] B. Natarajan, N. Wang, C. Gill, A. Gokhale, D. C. Schmidt, J. K. Cross, C. Andrews, and S. J. Fernandes. Adding real-time support to fault-tolerant CORBA. In *Proceedings of the Workshop on Dependable Middleware-Based Systems*, pages G1–G15, Washington, D.C., June 2002.

[19] Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. OMG Technical Committee Document formal/2001-12-01, December 2001.

[20] Object Management Group. Fault tolerant CORBA. OMG Technical Committee Document formal/2001-09-29, September 2001.

[21] Object Management Group. Real-time CORBA. OMG Technical Committee Document formal/2001-09-28, September 2001.

[22] Object Management Group. Real-time CORBA 2.0: Dynamic scheduling specification. OMG Technical Committee Document ptc/2001-08-34, September 2001.

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

[24] M. Saksena, J. Silva, and A. Agrawala. Design and implementation of maruti-ii, 1994.

[25] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998.

[26] E. Shokri, P. Crane, K. H. Kim, and C. Subbaraman. Architecture of ROAFTS/Solaris: A Solaris-based middleware for real-time object-oriented adaptive fault tolerance support. In *Proceedings of the Computer Software and Applications Conference*, pages 90–98, Vienna, Austria, Aug. 1998.

[27] J. A. Stankovic and F. Wang. The integration of scheduling and fault tolerance in real-time systems. Technical Report UM-CS-1992-049, Department of Computer Science, University of Massachusetts, Amherst, , 1992.

[28] 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.