CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:1527–1545
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.882

MEAD: support for Real-Time Fault-Tolerant CORBA

P. Narasimhan[*,†], T. A. Dumitraş, A. M. Paulos, S. M. Pertet,
C. F. Reverte, J. G. Slember and D. Srivastava

Electrical and Computer Engineering Department, Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh, PA 15213-3890, U.S.A.

SUMMARY

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. The MEAD (Middleware for Embedded Adaptive Dependability) system attempts to identify and to reconcile the conflicts between real-time and fault tolerance, in a resource-aware manner, for distributed CORBA applications. MEAD supports transparent yet tunable fault tolerance in real-time, proactive dependability, resource-aware system adaptation to crash, communication and timing faults with bounded fault detection and fault recovery. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: real-time; fault tolerance; trade-offs; non-determinism; CORBA; predictability; recovery

1. INTRODUCTION

Middleware platforms, such as CORBA (Common Object Resource Broker Architecture) 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 (QoS). For instance, there exist the Fault-Tolerant CORBA (FT-CORBA) [1] and the Real-Time CORBA (RT-CORBA) [2] specifications that aim to provide fault tolerance and real-time, respectively, to CORBA applications.

Despite its many attractive features, middleware still does not quite support applications that have *multiple simultaneous* QoS requirements, in terms of their reliability and real-time. It is simply not

[*] Correspondence to: Priya Narasimhan, Electrical and Computer Engineering Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3890, U.S.A.
[†] E-mail: priya@cs.cmu.edu

Received 30 March 2003
Accepted 20 September 2003

possible today for a CORBA application to have *both* real-time and fault-tolerant support through the straightforward adoption of implementations of the RT- and FT-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 [3] on the distributed 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 RT-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, 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 FT-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, particularly for middleware applications. Our research on the MEAD (Middleware for Embedded Adaptive Dependability) system attempts to identify and to reconcile the conflicts between real-time and fault tolerance in a resource-aware manner. MEAD was born out of the lessons learned from implementing FT-CORBA [4], from recognizing its limitations, and from the emerging need to support: (i) informed decision-making to assign fault-tolerance properties; (ii) trading off real-time and fault tolerance to suit the application's needs; and (iii) the transparent tuning of fault tolerance, on the fly, in response to dynamic system/resource conditions.

The MEAD infrastructure aims to enhance distributed RT-CORBA applications with new capabilities including: (i) transparent, yet tunable, fault tolerance in real-time; (ii) proactive dependability; (iii) resource-aware system adaptation to crash, communication and timing faults; with (iv) scalable and fast fault detection and fault recovery.

We chose CORBA as the vehicle for our initial investigations because CORBA currently incorporates separate real-time and fault-tolerance standards within its specifications, and provides us with the opportunity for reconciling the features of the two standards.

Section 2 provides the necessary background, including the specifications of the FT-CORBA and the RT-CORBA standards, respectively, as they exist today. Section 3 outlines the specific conflicts between real-time and fault tolerance for distributed applications. Section 4 describes the MEAD infrastructure that provides real-time fault-tolerant support for CORBA applications, along with our empirical evaluation of MEAD's specific features. Section 5 looks at existing related work on real-time fault-tolerant systems, while Section 6 concludes with the insights that we have gained from our research.

2. BACKGROUND

The CORBA [5] 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 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.

In this section, we present the FT-CORBA standard [1] and the RT-CORBA standard [2] as they currently exist—neither standard addresses, or intended to address, its impact on the other.

2.1. FT-CORBA

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. The Replication Manager allows users to create replicated objects in the same way that they would create unreplicated objects. Through the Replication Manager, users can also exercise direct control over the creation, deletion and location of individual replicas of an application object. Although each replica of an object has an individual object reference, the Replication Manager fabricates a group reference for the replicated object that clients use to contact the replicated object. For each application object, the user can configure the following fault-tolerance properties through the Replication Manager's interface:

- *factories*—nodes 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 node, process and object faults. Each application object inherits a `Monitorable` interface to allow the Fault Detector to determine the object's status. The Fault Detectors can be structured hierarchically, with the global replicated Fault Detector triggering the operation of local fault detectors on each node. Fault reports from the local Fault Detectors are sent 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. RT-CORBA

The RT-CORBA specification [2] 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 [6], as a part of RT-CORBA version 2.0.

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 RT-CORBA Scheduling Service for the offline scheduling of the application's tasks, typically in accordance with the proven Rate Monotonic Analysis algorithm [7]. 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.

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 server-side threadpool is used to avoid the run-time overhead of thread creation. The threadpool contains a number of pre-spawned threads, one 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.

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

End-to-end temporal predictability of the application's behavior is the single most important property of a RT-CORBA system. Strong replica consistency, under fault-free, faulty and recovery conditions, is often the single most important characteristic of a FT-CORBA system. The rest of this section outlines the real-time versus fault-tolerance issues for CORBA applications.

Non-determinism

The philosophical difference between real-time and fault-tolerant behavior manifests itself in many ways. 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. It is this reproducible behavior of the object that lends itself well to reliability. 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. Note that this definition of predictability applies to a single copy of the object executing on a single processor, and not to multiple, simultaneously executing copies of the object, across multiple (and possibly heterogeneous) processors.

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.

Ordering of operations

RT-CORBA and FT-CORBA both require the application's operations to be ordered, but for different reasons. From a real-time viewpoint, the most important criterion is to order the application's tasks in order to meet deadlines. To achieve this, the application's tasks are usually analyzed statically, and a schedule of operations and processing is computed ahead of time; the application then executes according to this ordered schedule at run-time. Thus, for every incoming operation, the schedule must be consulted to see if and when the operation ought to be delivered. 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. If the two orders of operations are identical, clearly, there is no conflict between the goals of the FT-CORBA and the RT-CORBA specifications. 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.

Bounding fault handling and recovery

Faults are unanticipated events that cannot really be predicted ahead of time. In a real-time system, the schedule of event occurrences is usually pre-planned, and then executed in order to meet the predicted deadlines. 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 systems 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 FT-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.

Based on the instant at which the fault occurred, and on the instant at which recovery is initiated, the recovery time can vary considerably. Potentially unbounded events, such as fault handling, logging and recovery, are anathema to a real-time system.

4. THE MEAD SYSTEM

The MEAD infrastructure aims to enhance distributed RT-CORBA applications with new capabilities including: (i) transparent, yet tunable, fault tolerance in real-time; (ii) proactive dependability; (iii) resource-aware system adaptation to crash, communication and timing faults; with (iv) scalable and fast fault detection and fault recovery. The following sections describe the various features of MEAD, including interception, versatile dependability, fault-tolerance advising, proactive fault tolerance and correction of non-determinism.

4.1. Transparency through interception

CORBA incorporates support for interception through the Portable Interceptors mechanism [8]. However, these are restricted to monitoring only CORBA's IIOP messages, and do not capture all other kinds of network communication that the application might employ, and that really ought to be accounted for. Furthermore, the application needs to be modified, and to be recompiled, in order to use the Portable Interceptor mechanisms.

Library interposition is our preferred method to intercept a process' network system calls by using the dynamic linker's run-time support [9] to load the MEAD Interceptor (a shared object library) into the process' address-space, ahead of all of the other dynamically linked libraries (DLLs). MEAD's Interceptor contains overridden definitions of common network-related library routines; each time the CORBA process or the ORB invokes a network-related library routine, the loader's symbol-resolution finds (and transparently forces the intercepted process to use) the first symbol definition in the Interceptor, rather than in the default DLL provided by the operating system. In turn, each overridden library routine in the Interceptor can find, and invoke, the corresponding routine in the default DLL, using dynamic linking calls such as `dlsym` and `dlopen`. The Interceptor overrides specific network-related functions (`read`, `write`, `sendmsg`, `recvmsg`, etc.) to perform network-traffic monitoring, along with the re-routing of the CORBA application's IIOP messages over the Spread group communication system [10] instead.

This form of interception allows us to insert the MEAD infrastructure in a manner that is transparent to the application and to the CORBA middleware. Library interpositioning also allows us to be language-neutral because the resulting Interceptor works off the standard library routine definitions, without requiring modifications to the operating system, without requiring recompilation of the application, and without requiring root/supervisory access. The Interceptor also provides easy hooks

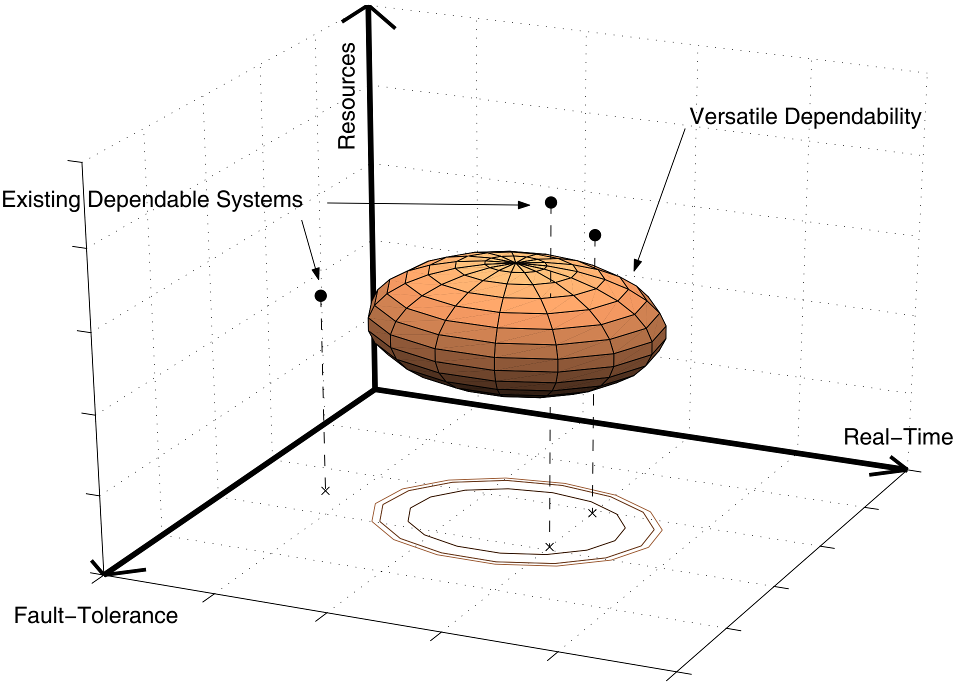

Figure 1. The design-space of dependable systems.

for immediate, inexpensive fault detection because it is uniquely positioned to detect the closure of sockets between clients and servers.

4.2. Resource-aware versatile dependability

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. We visualize the development of dependable systems through a three-dimensional *dependability design-space*, as shown in Figure 1, with the following axes: (i) the *fault-tolerance* 'levels' that the system can provide; (ii) the *real-time* guarantees that the system can provide; and (iii) the amount of *resources* that the system needs for each pairwise {fault-tolerance, real-time} choice.

We quantify the properties represented on these three axes by using various metrics, e.g. crash fault rates and network fault rates for the fault-tolerance axis, latency, jitter and number of missed deadlines for the real-time axis, and CPU, memory and bandwidth usage for the resource axis. Through this dependability design-space, we provide informed choices for the design parameters for real-time, fault-tolerant distributed applications. Our novel *versatile dependability* framework [11] offers a better coverage of the dependability design-space, by focusing on an operating region (rather than an operating point) within this space, and by providing a set of 'knobs' for tuning the trade-offs and properties of the system. In contrast to the existing point-solutions, versatile dependability encompasses larger regions of the design space (see Figure 1) because its tunability allows designers to choose a range of appropriate configurations for their specific needs.

Note that MEAD does not impose/require a 'one-style-fits-all' strategy; instead, it allows the maximum possible freedom in selecting a different replication style for each CORBA process (and allowing processes with different replication styles to communicate with each other), should that be necessary, and dynamically transitioning from one replication style to the other at run-time, in response to system/resource conditions.

At the implementation level, there are design choices that versatile dependability can control freely (e.g. how often to transfer the state) and design choices that are dependent on the application (e.g. how much state to transfer). The mechanisms that MEAD implements to support replication styles (active, warm passive, cold passive and semi-active replication) and tunability, within and across, the different replication styles, include the following.

- State extraction/restoration mechanism (checkpointing): needed in all of the replication styles to synchronize the state among replicas. For this purpose, the application being replicated must provide two special functions: `get_state()` and `set_state()`.
- State transfer protocol: required to maintain replica consistency. In all of the replication styles, the replicas must synchronize their states in order to maintain consistency. Active replication needs to execute a state transfer only during the initialization of a new replica, while passive replication performs state synchronization at each periodic checkpoint. The transfer protocol is, in practice, a reliable ordered multicast to the whole group of replicas.
- Quiescence detection: required to ascertain the 'safe' delivery of the next invocation to a server replica. 'Safety' here means that any shared state within the process is not corrupted by the simultaneous execution of multiple invocations/threads. Quiescence is particularly important in a multi-threaded application to prevent the state from being modified while the `get_state()` function is executing (otherwise the checkpointed state can be inconsistent).
- Request queuing: performed inside the Interceptor, and involves matching the sequence numbers associated with requests and replies, and also storing incoming invocations/responses while the supported replica undergoes recovery or checkpointing.
- Primary reelection: needed in the case of passive replication when MEAD must elect a new primary following a crash of the old primary replica.
- Duplicate detection and suppression: required for active replication during normal operation and for passive replication during recovery. The sequence numbers maintained/accessed by the request-queueing mechanism above can be exploited for this purpose.
- Ordering of requests/responses: required for maintaining strong replica consistency when dealing with (typically) non-independent/idempotent operations.
- Reliable fault detection: extremely important, particularly for determining if the primary replica has failed in passive replication, and if recovery ought to be initiated for either replication style.

Through the appropriate selection of the precise implementation choice for each of these mechanisms, along with the higher-level choice of replication style, MEAD allows the user to select the

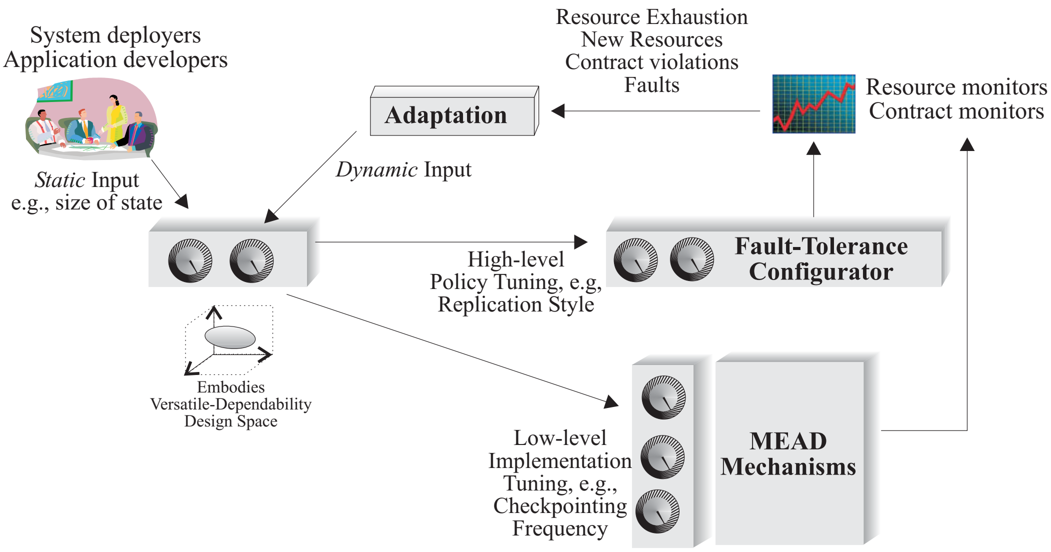

Figure 2. Versatile dependability in action at run-time.

'right amount/kind of fault tolerance' for the application, for each mode of the application (if modes exist within the application, as is often the case with real-time embedded systems), and for specific operating conditions.

As shown in Figure 2, the MEAD system monitors various system metrics and generates warnings when the operating conditions are about to change. If the contracts for the desired behavior can no longer be honored, MEAD adapts the fault tolerance to the new working conditions (including modes within the application, if they happen to exist). MEAD's adaptation algorithm performs this automatically by tuning the settings of its control knobs, in a manner that re-adjusts the trade-off between fault-tolerance and real-time guarantees to re-establish the contracts, if possible. If the re-enforcement of a previous contract is not feasible, versatile dependability can offer alternative (possibly degraded) behavioral contracts that the application might still wish to have. In some extreme cases, manual intervention might be required.

4.3. Fault-Tolerance Advisor

The FT-CORBA standard merely defines the fault-tolerance properties listed in Section 2.1, but does not discuss, or provide advice on, how the end-user should choose the right settings for these properties. Thus, in most FT-CORBA systems, values are assigned to these properties with little regard for the system configuration, the object's state size, the object's resource usage, the occurrence of faults, etc.

The problem is that the choice of values for an object's fault-tolerance properties really ought to be a decision based on the object's resource usage, the system's resource availability, and the object's reliability and recovery-time requirements. In the absence of assistance in deciding the appropriate values of these properties (for each object, and holistically for the entire system), the resulting arbitrary, and often inappropriate, choice of these properties could cause the system to miss the mark in terms of its target reliability and performance. In addition to a development-time tool to assist in these critical choices, there needs to be a run-time feedback framework that allows the development-time tool to re-learn and to re-adjust its decisions, dynamically, based on run-time events such as the removal or addition of resources, introduction of new applications, upgrades to the nodes or the network, fault patterns, etc.

The novel aspect of the MEAD Fault-Tolerance Advisor is that, given a CORBA application, the Advisor profiles the application for a specified period of time to ascertain the application's resource usage (in terms of bandwidth, CPU cycles, memory, etc.) and its rate/pattern of invocation. Based on this information, the Fault-Tolerance Advisor is then in a position to make suitable recommendations to the deployer on the best possible replication style to adopt for the specific application. For example, the Advisor might recommend the use of active replication, rather than passive replication, for objects which have a large amount of state, but fairly little computation.

The Advisor also pays attention to other fault-tolerance properties beyond the replication style. For instance, in passive replication, the checkpointing frequency is crucial in deciding the performance of the replicated object; higher checkpointing frequency involves trading off the benefit of faster recovery versus the disadvantage of the increased bandwidth/CPU used in frequent state retrieval and transfer. Based on the application's resource usage and its specified recovery-time bounds, the Advisor decides on the most appropriate value for the checkpointing frequency. Yet another tunable parameter is the fault-detection frequency. If the fault-detection frequency is higher, a fault can be detected faster, and fault recovery can be initiated more quickly at the expense of the resulting higher resource usage (in terms of CPU and bandwidth). The Advisor takes into account the system's fault rate, system's resource limits, the application's resource usage and the desired bound on recovery time, and produces the appropriate value of the fault-detection frequency for each object.

Of course, at run-time, multiple different applications might perturb each other's performance, leading to erroneous development-time advice. Recognizing this, the MEAD Fault-Tolerance Advisor incorporates a run-time feedback component that updates the development-time component with run-time profiling information in order to provide corrections to the original 'advice'.

4.4. Resource management

We implement the Fault-Tolerance Advisor over a fully decentralized infrastructure. As shown in Figure 3, on each node in the distributed system, there exists a MEAD Manager component and a Fault-Tolerance Advisor component. In addition to launching CORBA application programs, the MEAD Manager is responsible for fault detection and resource monitoring of its local processes, and for effecting the fault-tolerance properties specified by either a static fault-tolerance configuration or the Fault-Tolerance Advisor.

MEAD managers are symmetrically replicated across the nodes of the distributed system. They act as synchronized-state peers with no central controller and, therefore, no single point of failure. The synchronization is made possible through the use of the underlying Spread group communication system. By exploiting the reliable delivery and the ordering guarantees provided by the Spread system, we are assured of every MEAD Manager in the system receiving the same set of messages in the

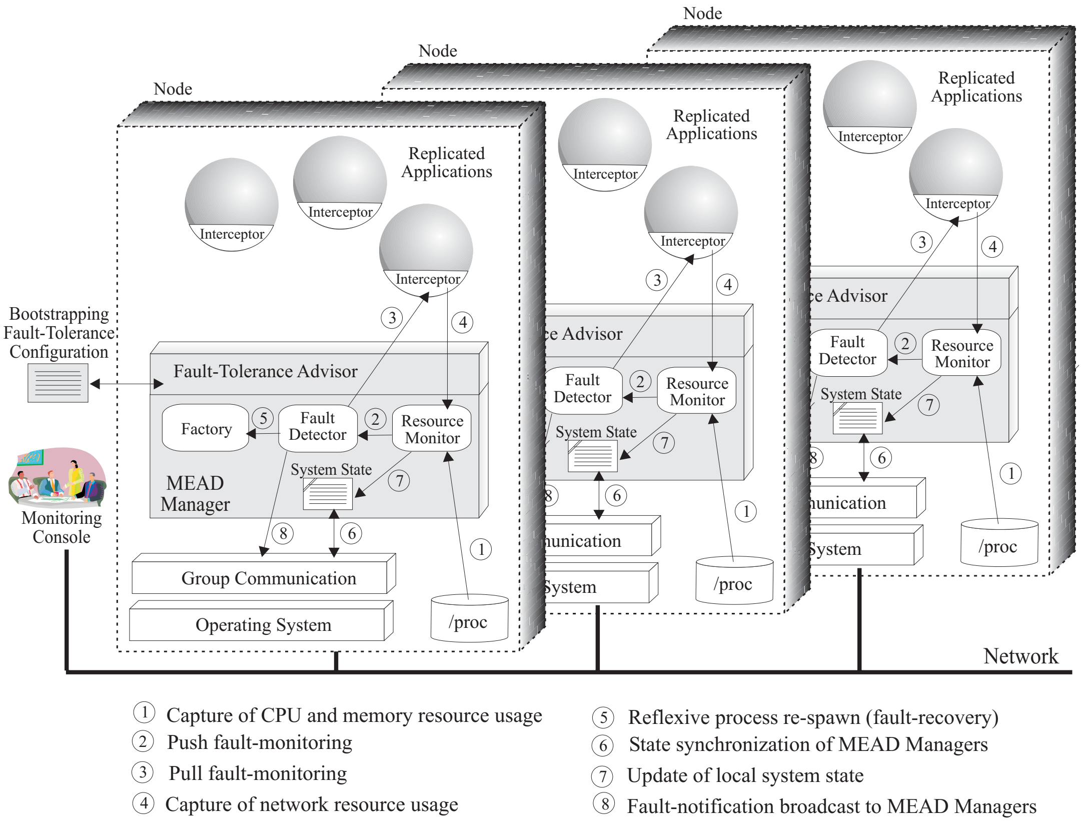
Node
Node
Node
Replicated Applications
Interceptor
Bootstrapping Fault-Tolerance Configuration
Fault-Tolerance Advisor
Factory
Fault Detector
Resource Monitor
System State
MEAD Manager
Monitoring Console
Group Communication
Operating System
/proc
Network
1 2 3 4 5 6 7 8
1 Capture of CPU and memory resource usage
2 Push fault-monitoring
3 Pull fault-monitoring
4 Capture of network resource usage
5 Reflexive process re-spawn (fault-recovery)
6 State synchronization of MEAD Managers
7 Update of local system state
8 Fault-notification broadcast to MEAD Managers

Figure 3. MEAD's decentralized resource management and the Fault-Tolerance Advisor.

same order, thereby facilitating synchronization and consistency both for MEAD and its supported applications. Of course, this implies that there must exist a Spread daemon and a MEAD Manager on each node that we wish to consider as a resource for the purposes of replication.

On each node, the MEAD Manager's resource monitoring component collects system load data. For each local application process, the resource monitor collects statistics such as the fault rate, CPU usage, memory usage, network usage and invocation rates. The local MEAD Manager shares this resource-usage data with the MEAD Managers on the other nodes through periodic broadcasts over the Spread system. By construction, the network overhead incurred in broadcasting the resource-usage data scales linearly with the number of nodes and number of processes per node, and can be as low as tens of bytes per application process per broadcast. As the system load and fault rates change, the MEAD system varies this broadcast rate dynamically.

For each local process, the Linux kernel [12] maintains resource consumption statistics, such as CPU usage and memory consumption. The MEAD Manager's resource monitor collects this kernel data periodically by accessing the `/proc` filesystem [13]. Because this data gathering is a polling type of activity, it does not capture data updates whenever they happen; however, this is acceptable due to the granular nature of process execution in a multitasking operating system. Data updates are inexpensive and have a fixed overhead per process as the load increases; thus, it is possible for MEAD to use data-collection rates that are low in overhead but that are high enough to track these resource parameters with a precision that is sufficient to enable fault-tolerance advising. The overhead of collecting and processing the resource statistics on each node scales linearly with number of processes running on that node. Network traffic statistics for individual processes are not automatically maintained by the Linux kernel, but the MEAD Interceptor makes it possible for us to account for network usage.

4.5. Proactive fault tolerance

MEAD's proactive fault tolerance [14] involves designing and implementing mechanisms that can predict, with some confidence, when a failure might occur, and then compensating for the failure even before it occurs. There are two aspects to proactive fault tolerance: firstly, the ability to predict failures; and, secondly, the mechanisms to compensate for the failures, hopefully before the failures can actually occur. The premise here is that there is a pattern of errors, or symptoms, that precede a failure, and that the observation/recognition of this pattern of errors might enable us to anticipate the onset of the failure. Thus, the ability to predict failures is dictated by our knowledge of the specific pattern, and the inter-arrival times, of any errors that herald the failure. Of course, we recognize that some failures might occur so abruptly that we cannot possibly hope to predict them; for example, if someone accidentally unplugs the power supply of a node, it might not be possible to find a pattern of errors preceding the power outage simply because there was no discernible 'lead-up' to the fault.

However, a number of interesting computer-related faults are often preceded by a visible pattern of abnormal behavior that favors the use of some form of prediction. Typically, these failures result from gradual degradation faults, such as resource exhaustion [15], or from transient and intermittent hardware faults, such as disk crashes [16] or telecommunication equipment failures [17]. Because it is not always possible to predict failures for every kind of fault, proactive dependability complements (and for certain kinds of faults, out-performs), but does not replace, the traditional reactive fault-tolerance schemes.

The current FT-CORBA standard does not support a proactive recovery strategy. All of the existing fault-tolerance mechanisms (e.g. the launching of new replicas, client-side fail-over, and the reelection of a new primary replica for passive replication) are all triggered in reaction to a detected rather than an anticipated fault. Therefore, merely applying the FT-CORBA specification to a real-time distributed CORBA application does not suffice to provide bounded temporal behavior, in the presence of faults. The reactive recovery strategy of the FT-CORBA standard results in a 'spike' in the application's latency and response times, whenever a fault occurs in the system. Using our proactive strategy, these 'spikes' are greatly mitigated to produce bounded response times in the application, even in the face of faults.

MEAD's Proactive Fault-Tolerance Manager is embedded within the server-side and client-side Interceptors. Because our initial focus is on resource-exhaustion faults, the Proactive Fault-Tolerance Manager monitors the resource usage at the server, and triggers proactive recovery mechanisms when

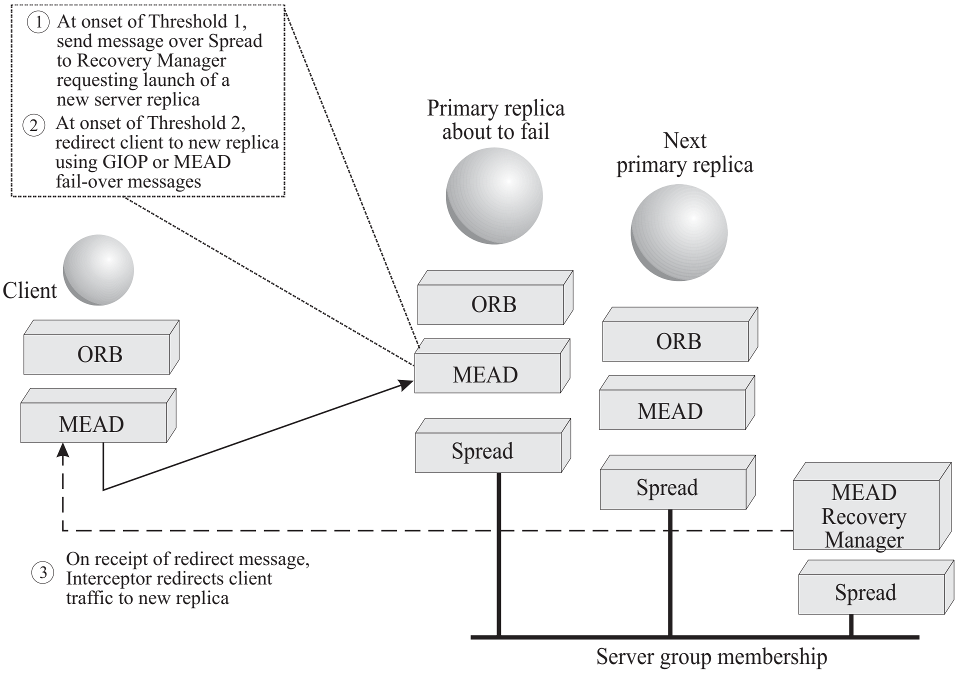
1 At onset of Threshold 1, send message over Spread to Recovery Manager requesting launch of a new server replica
2 At onset of Threshold 2, redirect client to new replica using GIOP or MEAD fail-over messages
Primary replica about to fail
Next primary replica
Client
ORB
MEAD
ORB
MEAD
Spread
ORB
MEAD
Spread
MEAD Recovery Manager
Spread
3 On receipt of redirect message, Interceptor redirects client traffic to new replica
Server group membership

Figure 4. MEAD's two-step threshold scheme for proactive failover.

it senses that resource usage has exceeded a predefined threshold. Here, 'resource' refers loosely to any resource of interest (e.g. memory, file descriptors, threads) to us that could lead to a process-crash fault if it were exhausted.

As shown in Figure 4, we implement proactive recovery using a *two-step threshold-based scheme*. When a replica's resource usage exceeds our first threshold, e.g. when the replica has used 80% of its allocated resources, the Proactive Fault-Tolerance Manager at that replica requests the Recovery Manager to launch a new replica. If the replica's resource usage exceeds our second threshold, e.g. when 90% of the allocated resources have been consumed, the Proactive Fault-Tolerance Manager at that replica can initiate the migration of all its current clients to the next non-faulty server replica in the group. Within our proactive dependability framework, the MEAD Recovery Manager is responsible for restoring the application's resilience in the event of failures. Thus, the Recovery Manager also receives messages from the MEAD Proactive Fault-Tolerance Manager whenever the Fault-Tolerance Manager anticipates that a server replica is about to fail. These proactive fault-notification messages can also trigger the Recovery Manager to launch a new replica to replace the one that is expected to fail.

Our preliminary results show that the use of MEAD's proactive fail-over messages can yield a promising 65% reduction in average fail-over times over the time that it would take to detect the failure in a reactive scheme and to resolve the next replica reference.

4.6. Correcting non-determinism

Real-time systems have several sources of non-determinism, as stated in Section 3. The FT-CORBA specification is explicit in stating its lack of support for non-deterministic applications. Thus, the only way to handle non-determinism in replicated systems today has been to eliminate it altogether from the application. Apart from this constraint, the heterogeneity of today's platforms forces FT-CORBA to mandate that all of the replicas of a CORBA object must be hosted on the same processor, operating system (version) and hardware. Otherwise, unforeseen non-deterministic effects can arise from any differences in the underlying platform that can cause replicas to diverge in state and/or behavior.

It is our belief that the knowledge of the application's structure and functionality can provide for a better way to identify, and to handle, non-determinism. MEAD's identification of non-determinism is done using program analysis techniques [18] applied to the application source code. Our sanitization of non-determinism also exploits program analysis, but has two separate components: *compile-time* and *run-time*. Our approach is deliberately not transparent to the programmer so as to allow them the opportunity to keep non-determinism from the application, and also not transparent to the user so as to allow them to decide how much non-determinism to eliminate, based on the associated overheads.

For each specific instance of non-determinism, MEAD's Cross-File Non-determinism Analyzer (CFNA), the offline program analysis tool that correlates client and server source code to detect non-determinism, also creates data structures to hold the non-deterministic information. For example, the CFNA creates a `struct` to hold the return value of `gettimeofday()`. This `struct` is then prepended to each message that outputs/propagates the non-determinism. Additionally, the recipient of the message, be it a client or a server, must also store this non-deterministic information locally. Therefore, the `struct` is also created at the receiving side. Since we know the source and destination of the non-determinism, we can determine where to insert our sanitization code, typically, just after the system call that is non-deterministic. If `gettimeofday()` is called on each invocation of a replicated time-server, then it is important that the times be corrected if there exists clock-skew across the nodes hosting the replicas; our sanitization snippet contains the code that performs this correction. These snippets are inserted automatically by the CFNA into the source code wherever appropriate. Once all of the source files have been modified, the program can then be compiled to produce a sanitized version of the original non-deterministic application.

The run-time aspect of our approach is responsible for piggybacking the non-deterministic information onto messages exchanged between the client and the server in the application. In Figure 5, we show a CORBA client communicating with a three-way actively replicated CORBA server. The MEAD infrastructure forms the fault-tolerant communication layer that enables the reliable ordered delivery of the application's messages. In Figure 5(a), the client sends a request to the replicated server and receives three distinct replies. Each active server replica executes the request, and prepends the non-deterministic results along with a unique server identifier `SID` to the GIOP response returned to the client. Each server replica stores its non-deterministic information locally, until the next invocation, as shown in Figure 5(b). In the figure, the client picks the first received response from the replicated server (in this case, from the replica $S2$) and stores the prepended information. We show the client's subsequent request to the replicated server in Figure 5(c). To every invocation that it transmits, the client prepends the non-deterministic information extracted from the previous reply that it received. In Figure 5(d), each server replica receives the request and compares its identifier with that transmitted with the prepended header. If there is a match (as in replica $S2$), then, the invocation proceeds normally.

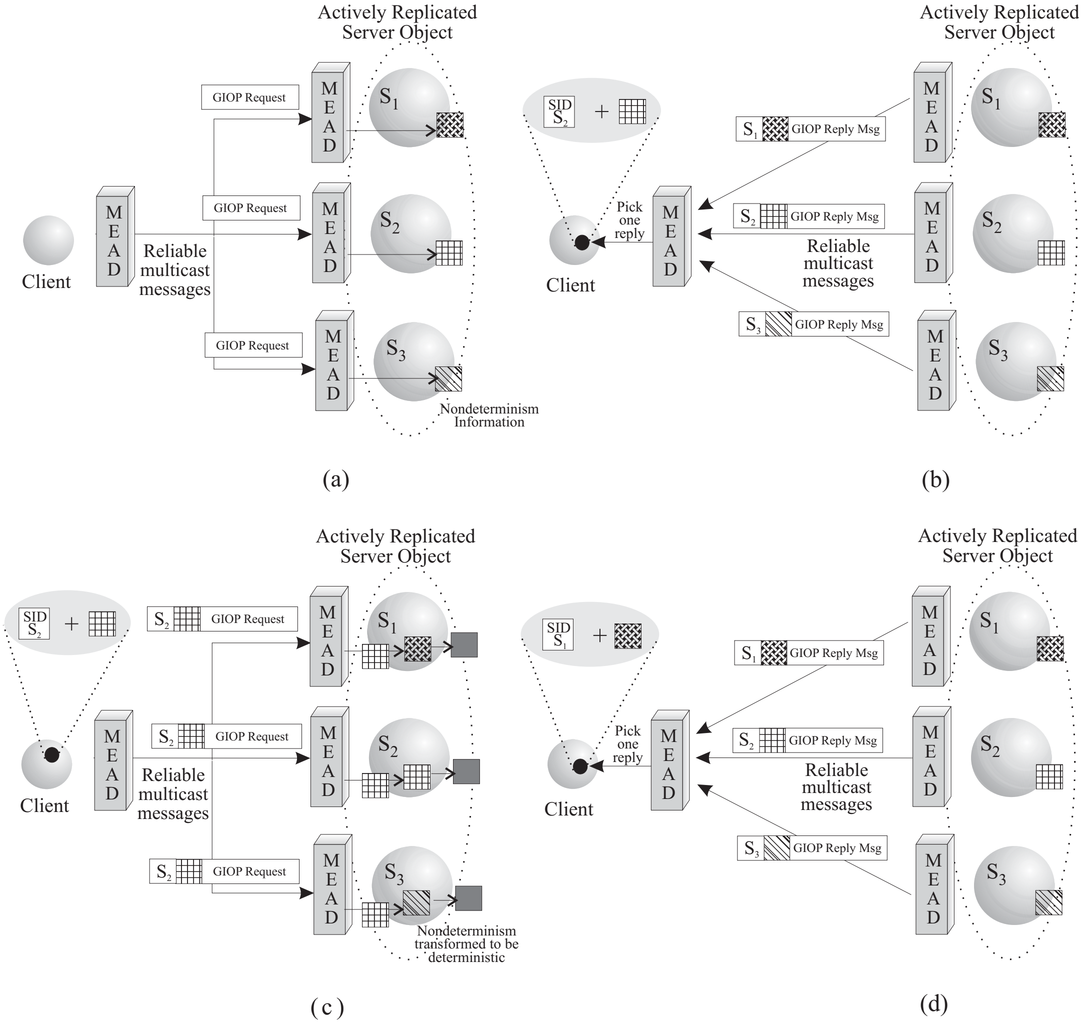
Actively Replicated Server Object
Client
MEAD
Reliable multicast messages
GIOP Request
S1
S2
S3
Nondeterminism Information
(a)
SID S2
Pick one reply
S1 GIOP Reply Msg
S2 GIOP Reply Msg
S3 GIOP Reply Msg
(b)
Nondeterminism transformed to be deterministic
(c)
SID S1
(d)

Figure 5. (a) Client invokes non-deterministic replicated server; (b) server replicas respond to client with non-deterministic values; (c) client invokes non-deterministic server again, this time prepending non-deterministic values to the request message; (d) server replicas respond to client with new non-deterministic values prepended to the reply message.

However, where there is a mismatch ($S1$ and $S2$), the server replicas execute the sanitization code after processing the invocation. For example, in the case of `gettimeofday()`, the sanitization would involve executing a local CFNA-generated snippet to compute the offset between the prepended non-deterministic information and the locally computed one. Each server replica then responds with its result for the invocation with the non-deterministic information that it generated locally. Each replica's local 'cache' of non-deterministic information is also updated. This 'back-and-forth' propagation and correction of non-determinism occurs on every invocation from the client to the replicated server.

The overheads for sanitizing non-determinism at run-time (given that the program analysis phase has already completed at compile-time) are reasonable, for the kinds of non-determinism (e.g. the use of time) that we have incorporated into CORBA applications. Compared with the baseline MEAD-supported replication of a deterministic CORBA application, the overhead of the MEAD-supported replication of the non-deterministic CORBA application is 1.6% under active replication, and 8.3% under warm passive replication. This increase is predominantly due to increased message passing as checkpoints are processed periodically in warm passive replication.

5. RELATED WORK

Stankovic and Wang's work [19] recognized the tension between real-time and fault tolerance, and looked at providing fault tolerance for real-time systems 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.

In [20] a time-redundancy approach to solve the problem of scheduling real-time tasks, along with transient recovery requests, in a uniprocessor system is described. 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 pre-computed deadline, while being serviced at an assigned responsiveness level. Otherwise, the recovery request is rejected.

ARMADA [21] 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 [22] 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 [23] 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 [24] extended the original Delta-4 system [25] 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 [26]. More recently, fault-tolerant features have been added to the RT-CORBA implementation TAO [27]. 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 [28] is designed to support the adaptive fault-tolerant execution of both process-structured 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.

6. CONCLUSION

The first contribution of this paper was the identification of the conflicts between real-time and fault tolerance for distributed CORBA applications that need both of these properties. Because the behavior of even a simple real-time CORBA application is not sufficiently bounded or predictable in the presence of faults, we recognize the need for additional mechanisms to resolve the real-time versus fault tolerance trade-offs.

The second contribution of this paper is the description of a systematic, tunable approach to building dependable distributed applications with stringent temporal and reliability requirements, under a variety of operating conditions and modes. The MEAD system that we are developing aims to provide support for real-time fault-tolerant CORBA applications through a combination of novel mechanisms: resource-aware versatile dependability, fault-tolerance advice, proactive fault-tolerance and correction of application-level non-determinism. Although our contributions are described in the context of CORBA, they apply to any middleware application that requires both real-time and fault-tolerance support, in a distributed asynchronous environment.

ACKNOWLEDGEMENTS

We gratefully acknowledge support from the NSF CAREER Award CCR-0238381, the DARPA PCES contract F33615-03-C-4110, the General Motors Collaborative Laboratory at CMU, the Air Force Research Laboratory grant number FA8750-04-01-0238 (‘Increasing Intrusion Tolerance Via Scalable Redundancy’), the Army Research Office grant DAAD19-01-1-0646, and grant number DAAD19-02-1-0389 (‘Perpetually Available and Secure Information Systems’) to the Center for Computer and Communications Security at CMU.

REFERENCES

1. Object Management Group. Fault-Tolerant CORBA. *OMG Technical Committee Document formal/2001-09-29*, September 2001.
2. Object Management Group. Real-Time CORBA. *OMG Technical Committee Document formal/2001-09-28*, September 2001.

3. Narasimhan P. Trade-offs between real-time and fault-tolerance for middleware applications. *Proceedings of the Workshop on Foundations of Middleware Technologies*, Irvine, CA, November 2000.
4. Felber P, Narasimhan P. Experiences, approaches and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers* 2004; **54**(5):497–511.
5. Object Management Group. The Common Object Request Broker: Architecture and specification, 2.6 edition. *OMG Technical Committee Document formal/2001-12-01*, December 2001.
6. Object Management Group. Real-time CORBA 2.0: Dynamic scheduling specification. *OMG Technical Committee Document ptc/2001-08-34*, September 2001.
7. Liu CL, Layland JW. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery* 1973; **20**(1):40–61.
8. Object Management Group. Portable Interceptors. *OMG Technical Committee Document formal/2001-12-25*, December 2001.
9. Levine JR. *Linkers and Loaders*. Morgan Kaufmann: San Francisco, CA, 2000.
10. Amir Y, Danilov C, Stanton J. A low latency, loss tolerant architecture and protocol for wide area group communication. *Proceedings of the International Conference on Dependable Systems and Networks*, New York, June 2000; 327–336.
11. Dumitraş T, Narasimhan P. An architecture for versatile dependability. *DSN Workshop on Architecting Dependable Systems*, Florence, Italy, June 2004.
12. Bovet DP, Cesati M. *Understanding the Linux Kernel*. O'Reilly: Sebastopol, CA, 2003.
13. Faulkner R, Gomes R. The process file system and process model in UNIX System V. *Proceedings of the Winter USENIX Conference*, January 1991.
14. Pertet S, Narasimhan P. Proactive fault-tolerance for distributed CORBA applications. *Proceedings IEEE Conference on Dependable Systems and Networks (DSN)*, Florence, Italy, June 2004; 357–366.
15. Huang Y, Kintala C, Kolettis N, Fulton ND. Software rejuvenation: analysis, module and applications. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995; 381–390.
16. Lin T-TY, Siewiorek DP. Error log analysis: Statistical modeling and heuristic trend analysis. *IEEE Transactions on Reliability* 1990; **39**(4):419–432.
17. Vilalta R, Sheng M. Predicting rare events in temporal domain. *Proceedings of the IEEE International Conference on Data Mining*, 2002; 474–481.
18. Slember J, Narasimhan P. Exploiting program analysis to identify and sanitize nondeterminism in fault-tolerant, replicated systems. *Proceedings IEEE Symposium on Reliable Distributed Systems (SRDS)*, Florianopolis, Brazil, October 2004; 251–263.
19. Stankovic JA, Wang F. 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.
20. Mejia-Alvarez P, Mosse D. A responsiveness approach for scheduling fault recovery in real-time systems. *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, Canada, June 1999.
21. Abdelzaher TF *et al.* ARMADA middleware and communication services. *Real-Time Systems* 1999; **16**(2–3):127–153.
22. Saksena M, Silva J, Agrawala A. Design and implementation of Maruti-II. 1994.
23. Kopetz H, Damm A, Koza C, Mulazzani M, Schwabl W, Senft C, Zainlinger R. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro* 1989; (February):25–40.
24. Barrett P, Bond P, Hilborne A, Rodrigues L, Seaton D, Speirs N, Verissimo P. The Delta-4 extra performance architecture (XPA). *Proceedings of the Fault-Tolerant Computing Symposium*, Newcastle, U.K., June 1990; 481–488.
25. Powell D. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer: Berlin, 1991.
26. Melro S, Verissimo P. Real-time and dependability comparison of Delta-4/XPA and Mars systems. *INESC Technical Report RT/09-92*, University of Lisbon, Portugal, 1992.
27. Natarajan B, Wang N, Gill C, Gokhale A, Schmidt DC, Cross JK, Andrews C, Fernandes SJ. Adding real-time support to fault-tolerant CORBA. *Proceedings of the Workshop on Dependable Middleware-Based Systems*, Washington, DC, June 2002; G1–G15.
28. Shokri E, Crane P, Kim KH, Subbaraman C. Architecture of ROAFTS/Solaris: A Solaris-based middleware for real-time object-oriented adaptive fault tolerance support. *Proceedings of the Computer Software and Applications Conference*, Vienna, Austria, August 1998; 90–98.