Middleware for Embedded Adaptive Dependability

P. Narasimhan, C. Reverte, S. Ratanotayanon and G. Hartman
Carnegie Mellon University, Pittsburgh, PA 15213-3890.
{priya@cs, cfr@andrew, sratanot@andrew, gghartma@cs}.cmu.edu

Abstract

The Middleware for Embedded Adaptive Dependability (MEAD) infrastructure enhances large-scale distributed real-time embedded middleware applications with novel capabilities, including (i) transparent, yet tunable, fault tolerance in real time, (ii) proactive dependability, (iii) resource-aware system adaptation to crash, communication, partitioning and timing faults with (iv) scalable and fast fault-detection and fault-recovery.

1. Introduction

Middleware, such as CORBA and Java, have come to incorporate support for many "-ilities" (e.g., reliability, real-time, security). Unfortunately, both the CORBA and the Java middleware standards can support only either real-time or fault tolerance in isolation. This is due to a deeper fundamental problem - the fact that *real-time and fault tolerance often impose conflicting requirements on a distributed system*. While real-time requires *a priori* knowledge of the system's temporal operation, fault tolerance must necessarily deal with faults that occur unexpectedly, and with possibly unpredictable fault recovery times. Our preliminary measurements [5] show that faults can disrupt a Real-Time CORBA application, and do lead to unbounded and variable fault-detection and 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 [2] are made during the composition.

Today's middleware applications (including most kinds of mission-critical applications) that require *multiple simultaneous* "-ilities", such as reliability and real-time, end up adopting some combination of the standardized solution for one "-ility" along with an ad-hoc proprietary mechanism for the other "-ility". The end result is a system that is often brittle, and one that is not easy to maintain or to upgrade, because it does not fully capture the trade-offs and the interactions between real-time and fault tolerance.

This research attempts to identify and to reconcile the conflicts between real-time and fault tolerance in a resource-aware manner. The novel ideas underlying this research are captured in the scalable, transparent, tunable, real-time, fault-tolerant MEAD[1] infrastructure. Although MEAD is described in the context of CORBA, its transparency allows it to support other middleware platforms, including Real-Time Java. The real-time fault-tolerant resource-aware MEAD infrastructure combines the strengths of:

- Informed development-time tools to assist the application programmer in making the critical choices and trade-offs between real-time and fault tolerance,
- Multi-level resource-aware feedback loops for the system to (learn to) adapt to new run-time situations,
- Profiling trends and events to predict faults even before they occur, and
- Proactive measures to compensate for, and recover from, faults before they impact the system.

MEAD extends its capabilities and protection to its own components, making it a truly self-healing system.

2. The MEAD Infrastructure

MEAD achieves its goals through a number of collaborating distributed reliable components, as shown in Figure 1, including the following.

Replication Manager: This component replicates the objects and the processes of the middleware application, and distributes the replicas across the processors in the system. The application deployer can select fault tolerance properties for each application object using a graphical user interface to the Replication Manager. This enables the Replication Manager to decide how many replicas to create for each object, where to place the replicas, and which replication style to enforce for the replicated object. The MEAD Replication Manager also assumes responsibility for maintaining a certain level of dependability by ensuring a pre-specified minimum degree of replication for each object.

[1] Mead, the legendary ambrosia of the Vikings, was believed to endow its imbibers with immortality (i.e., *dependability*), reproductive capabilities (i.e., *replication*), the wisdom for weaving poetry (i.e., *cross-cutting aspects of real-time and fault tolerance*) and a happy and long married life (i.e., *partition-tolerance*).

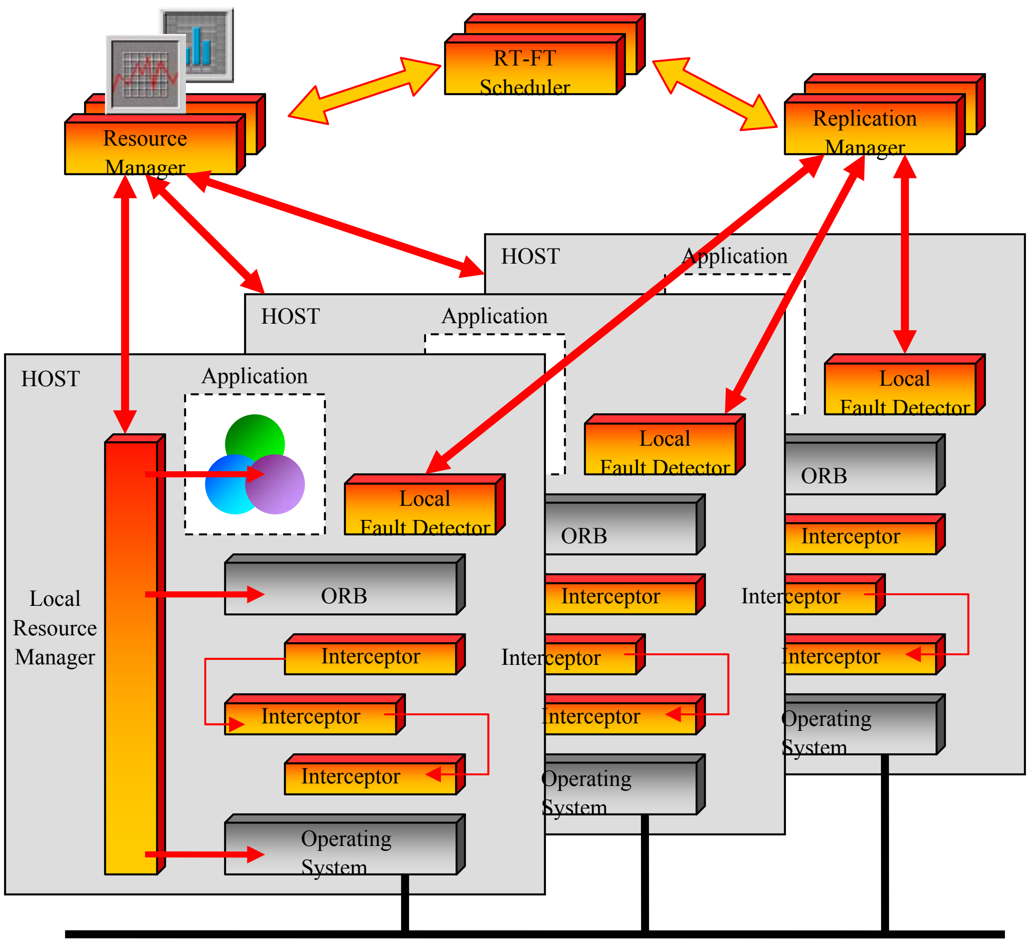

Figure 1: MEAD's scalable real-time fault-tolerant run-time infrastructure.

Hierarchical Fault Detection-Reporting-Analysis: On each processor, there exists a Local Fault Detector that detects the crash of objects and processes on that processor. The MEAD Local Fault Detectors feed their fault notifications into the Replication Manager, thereby allowing it to restore the degree of replication if a replica has crashed. The Local Fault Detectors and the Replication Manager additionally do the work of fault analysis. The fault analysis serves to conserve bandwidth (in the case of multiple fault reports that can be collated into a single fault report) and to provide a more accurate diagnosis of the source of the fault. For instance, if a processor hosting 100 objects crashes, a single processor-crash fault report might save bandwidth and provide more utility than 100 individual object-crash fault reports. The MEAD fault detection-reporting-analysis framework is structured hierarchically for reasons of scalability.

Hierarchical Resource Management Framework: On each processor in the distributed system, there runs a Local Resource Manager that monitors the resource usage of the replicas on that processor. Note that the MEAD Local Resource Manager, as shown in Figure 1, has instrumentation and profiling hooks at the operating system, the ORB and application levels in order to monitor the respective resource usage of each of these local components. The Local Resource Managers communicate this information to the MEAD Global Resource Manager. This system-wide Global Resource Manager is aware of the processor and communication resources, and their interactions, across the entire distributed system. The Global Resource Manager has a system-wide perspective of the current resource availability and resource usage across all of the processors. Thus, the Global Resource Manager is uniquely positioned to make decisions about migrating objects/processes from heavily loaded processors onto relatively lightly loaded processors, in order to meet the application's timing requirements, or to isolate resource-intensive tasks onto relatively idle processors. It can also make decisions about selectively shedding load (based on task criticality) under overload conditions.

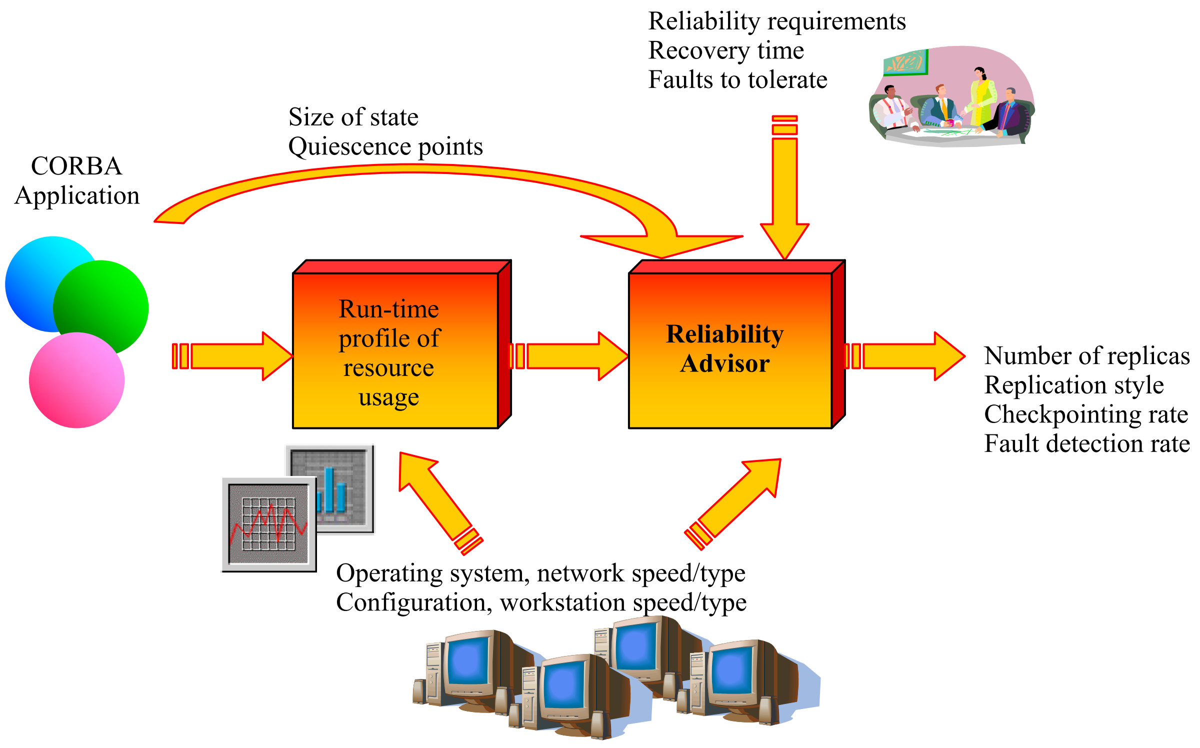

Figure 2: MEAD's development-time Reliability Advisor.

RT-FT Scheduler: On each processor, there runs a Local RT-FT Scheduler (physically collocated with the Local Resource Manager in Figure 1) that monitors the scheduling and the performance of tasks on that processor. An offline scheduler (not shown) computes a real-time schedule of the application's tasks ahead of run-time; this schedule provides the sequence of operations in the absence of faults. The MEAD Global RT-FT Scheduler then starts to execute the application according to this pre-determined schedule. To withstand runtime conditions, the Global Scheduler inspects the schedule, computes the available slack time, and reallocates work to selective replicas of an object so that the object continues to meet its original deadlines, and is yet able to accommodate dynamic situations (e.g., faults, loss of resources, object migration, recovery) that arise. Thus, the pre-computed schedule does not necessarily change, at run-time, in the presence of faults; instead, the RT-FT Scheduler works with the proactive dependability framework to determine the least disruptive point in the schedule for initiating fault-recovery. Working with the FT-Hazard Analyzer, the RT-FT Scheduler knows the worst-case object recovery times and "safe" checkpointing and recovery instants in the lifetime of the object. The RT-FT Scheduler is also responsible for triggering the dynamic scheduling of two key operations, object/process migration and recovery.

Proactive Dependability Framework: Proactive dependability involves predicting, with some confidence, when a fault might occur, and compensating for the fault even before it occurs. For instance, the knowledge that there is an 80% chance of a specific processor crashing within the next 5 minutes could allow the MEAD system a chance to survive the crash with far less penalty than waiting for the crash to occur; in this case, MEAD might elect to migrate all of the processes/objects on the moribund processor within the remaining 5 minutes. The important aspect of this proactive recovery action is that it is less disruptive on the real-time deadlines and the normal operation of the system because the recovery can be staggered out within the currently fault-free functional version of the system. Furthermore, anticipatory recovery from a fault places fewer demands on the system than reactive recovery.

Proactive dependability hinges on the ability to predict when a fault can occur. This is possible because a fault is typically preceded by a visible pattern of abnormal behavior just prior to the fault. For instance, looking through the Windows NT error log, one can see that a disk fault is usually preceded by a pattern of what might appear to be relatively minor errors and warnings. Recognizing this pattern, along with the source of, and the temporal separation between, consecutive errors/warnings in this pattern allows us to compute the "slack-time" available before the fault. Using statistical and heuristic techniques [1], the occurrence of any individual error/warning, along with a "window" spanning its adjacent errors/warnings, we can predict, with some confidence level, the time at which a fault might follow.

This requires a Local Error Detector to record all of the errors/warnings into a Log, a Fault Analyzer to sift through the log entries and to forward predictions to the Global Proactive FT-Manager. The Proactive FT-Manager collects all of these predictions, assesses the severity and the likelihood of the predicted faults, and communicates impending "doom" (i.e., impending loss of resources or of replicas or processors) to the Resource Manager and the Replication Manager. These components, in their turn, work quickly with the Global RT-FT Scheduler to close the feedback loop by triggering object/process migration, load balancing, load shedding, or any one of several other fault-recovery decisions. One example of fault-recovery involves proactively restarting, or injecting new life into, applications that might have memory leaks in them, through a process called software rejuvenation. The Local Resource Managers, through their profiling information, communicate an object's memory-leak trend to the Local Error Detectors, which then communicates it to the rest of the proactive dependability framework. Another example involves "poison requests", i.e., requests that always cause the system to crash or to miss deadlines. MEAD profiles the communication and execution patterns of all of the requests in order to identify such "poison requests", and to cause them to be discarded as a preventive measure before they can damage the system.

Chain of Interceptors: Interception [4] allows us to insert the MEAD infrastructure transparently underneath any unmodified middleware application, and under any unmodified CORBA ORB. Interception also allows us to be language-neutral because the interceptor relies only on the existence of the (IIOP) protocol interface of any middleware platform. Underneath the ORB, MEAD inserts a chain of interceptors, with each additional interceptor transparently enhancing the ORB with some new capability. The value of interceptors is two-fold. For one, interceptors can be installed at run-time, into the process address space, of an unmodified CORBA application.

By being readily installable and uninstallable at run-time, interceptors allow MEAD's mechanisms to "kick in" only when a fault occurs, or when a resource is lost or depleted. This prevents MEAD from adversely impacting the application's performance in the absence of faults or in the presence of sufficient resources.

Secondly, interceptors can be chained together to produce even more interesting cross-cutting functionality. We envision the real-time fault-tolerant MEAD infrastructure using a number of interceptors, each providing some additional assurances of dependability, e.g., there would exist interceptors for reliable multicast, TCP/IP, real-time annotations of messages, task profiling, checkpointing, resource monitoring. Installing or uninstalling such an interceptor would then increase, or reduce, respectively, the infrastructure's offered dependability. Interceptors are analogous to the concept of aspect-oriented programming (AOP), where the cross-cutting concerns of a system are represented by different aspects, which are then woven together to capture the system interactions. We intend to embody the concepts of AOP into our interception technology to capture and to resolve the real-time vs. fault tolerance trade-offs, and to provide for different runtime-customizable levels of real-time and fault tolerance.

Reliability Advisor: This development-time tool allows the application deployer to select the right reliability configuration settings for his/her application, as shown in Figure 2. These settings include the fault tolerance properties for each of the objects of the application that requires protection from faults. The properties for each object include (i) degree of replication, i.e., the number of replicas, (ii) replication style, i.e., one of the active, active with voting, warm passive, cold passive and semi-active replication styles, (iii) checkpointing frequency, i.e., how often state should be retrieved and stored persistently, (iv) fault detection frequency, i.e., how often the object should be "pinged" for liveness and (v) the physical distribution, or the precise location, of these replicas. Unlike current dependability practices, we do not decide on these properties in an ad-hoc unsystematic manner. Instead, MEAD makes these property assignments through the careful consideration of the application's resource usage, the application's structure, the reliability requirements, the faults to tolerate, etc. The novel aspect of the MEAD reliability advisor is that, given any application, the advisor will profile the application for a specified period of time to ascertain the application's resource usage (in terms of bandwidth, processor cycles, memory, etc.) and its rate/pattern of invocation. Based on these factors, the advisor makes recommendations [7] to the deployer on the best possible reliability strategy to adopt for the specific application. Of course, at run-time, multiple different applications might perturb each other's performance, leading to erroneous development-time advice. Keeping this in mind, the MEAD reliability advisor has 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". This feedback component is nothing but the Local Resource Managers operating in concert with the Global Resource Manager.

3. Key Concepts and Strategies

Replication – Not Just for Reliability, But for Real-Time: Replication has primarily used to obtain fault tolerance, i.e., having multiple copies, or replicas, of an object/process distributed across a system can allow some replicas to continue to operate even if faults terminate other replicas. MEAD goes beyond this in *exploiting replication, a common fault tolerance technique, to derive better real-time behavior*! With replication, there always exist redundantly operating replicas which receive and process the same invocations, and deterministically produce the same responses. The invocations and responses are synchronized across the different replicas of the same object in logical time; of course, the invocations/responses might be received at individual replicas at different physical times. Thus, a faster replica might produce a response more quickly, but nevertheless in the same order as a slower replica of the same object. If we send an invocation to two replicas of the same object, where one replica is faster and the other is slower, the faster replica will return the response faster, and can move onto the next scheduled invocation more quickly. In this case, the slower replica's response is a duplicate and can probably be safely discarded. In any case, it might be late, or miss the deadline for producing the response. The faster replica's result can be fed back to the slower replica's FT-infrastructural support, thereby suppressing the duplicate response. By staggering the times at which invocations are released to replicas of different speeds, we can always ensure that we have at least one replica that beats the others in terms of meeting the specified deadline for producing the response. This technique is effective in that we exploit the duplicate responses of the replicas, as well as their redundant processing, in order to meet deadlines and to tolerate faults. This technique is effective when triggered with due consideration of the network topology.

Partition-Tolerance: Large-scale systems are particularly vulnerable to insidious network partitioning faults, where the system partitions into disconnected components. With additional mechanisms, MEAD could also sustain continuous, albeit degraded, operation in a partitioned system, and could facilitate remerging and recovery when the partition heals. The problems of network partitioning are aggravated by replication. When a partition occurs, some of the replicas of an object might reside in one component, while the other replicas of the same object are "trapped" in the other, disconnected, component. In the partitioned, or disconnected, mode of operation, the two sets of replicas of the same object might make different decisions which cause their respective states to diverge. When the partition heals, these diverging states and actions might be difficult, if not impossible, to reconcile. Application programmers are ill-equipped to deal with this reconciliation, and there needs to be some automated support for network remerging in a dependable system. The MEAD infrastructure contains key building-blocks [3] that can be exploited to support partition-tolerant systems and to facilitate replica consistency during remerging.

Support for Nested Operations: Nested, or chained, operations are the bane of every fault tolerance developer. A nested operation involves a multi-tiered chain of objects, where one tier invokes the following tier down the chain. Thus, a nested operation could span objects that act simultaneously as both client and server (server to the preceding tier in the chain and client to the following tier in the chain), and has the potential to lead to cascaded faults. Nested operations require imposing different real-time deadlines [6] from normal operations, and different infrastructural timeouts need to be set on them. For instance, suppose that we have a three-tiered nested application, with object A (tier 1) invoking object B (tier 2) which then invokes object C (tier 3). If object A invokes object B in a non-nested fashion, then, B would process the invocation and return a response immediately. If object A invokes object B in a nested fashion, then, object B processes the invocation, invokes object C, in its turn; object C returns a response to object B's invocation, after which B responds to A. The problem here is that the timeout for A's invocation to B varies in the non-nested and nested cases. However, object A might not know ahead of time, without any knowledge of object B's internal operation, if its invocation of object B might lead to a nested invocation of object C. Secondly, if a fault occurs in some stage/tier of a nested operation, we do not necessarily want to restart the entire nested operation, which could span multiple stages/tiers.

MEAD's underlying infrastructure knows whether a specific invocation triggers a further, nested, invocation. This information is useful when determining the precise value of the timeout to set on the deadline for receiving a response across the tiers of a nested operation. MEAD stores information about partially complete operations and caches their state-updates and response in order to disseminate these changes, and to recover more quickly, in the case of nested operations with faults.

Incremental Checkpointing: Checkpointing, or saving the state, of applications with a large amount of state, is a non-trivial exercise. It involves the retrieval of the application's state, the transfer of this state across the network, and the logging of this state onto some persistent stable storage. When the state of an application is large, checkpointing consumes both CPU cycles and bandwidth, and can choke up the entire system. MEAD employs a differential, or incremental, checkpointing scheme, analogous to the way in which versioning systems, such as CVS, operate. CVS stores incremental "diffs" between different versions of the same file, rather than store the entire file for each new version; the advantage is that less information is stored, although some price has to be paid for reconstructing each version. Incremental checkpointing operates on the same principle; instead of checkpointing the entire state, MEAD checkpoints "diffs", or state-increments, between two successive snapshots of the state of an object/process. The state-increments are usually smaller than the entire state itself, and can be transferred more quickly, leading to faster recovery times and more RT-deterministic behavior, under faulty and recovery conditions. The mechanisms for incremental checkpointing involve understanding the application code sufficiently to extract state-increments. We are currently investigating the development of tools to assist the application programmer in identifying "safe" incremental-checkpointing points in the code, as well as the size and the structure of each state-increment.

Self-Healing Mechanisms: Components of the MEAD infrastructure are also replicated in the interests of fault tolerance. Their resource usage is equally monitored, along with that of the application. Typical fault tolerance issues include the "Who watches the watchers?" problem, e.g., how the Replication Manager replicates itself, and recovers a failed replica of itself, how the Proactive FT-Manager deals with a fault-report predicting a fault within one of its replicas, how the Resource Manager reacts to one of its replicas being migrated, etc. MEAD handles this by having the replicas of its own components employ a "buddy-system" approach, i.e., the replicas of each of MEAD's own components watch over each other, recover each other, and maintain their own degree of replication. At the same time, replicas that are "buddies" should not adversely impact each other's performance or reliability under either fault-free or recovery conditions, and must be able to pair up with an operational "buddy" if their existing "buddy" fails. Bootstrapping (i.e., starting up from scratch) such a self-monitoring and self-healing system is tricky because it requires bringing the system up to a certain initial level of reliability and functionality before allowing the application to execute.

Fault Prediction: The algorithms for fault prediction and dependability forecasting are tricky as they depend on knowing the pattern of abnormal behavior that precedes different kinds of faults. MEAD will employ, and extend, algorithms for data mining in order to look through error logs that it maintains (in addition to the system error-logs themselves) to forecast the occurrence of faults. For each prediction, MEAD needs to associate a confidence level in order to allow the adaptation framework to determine whether or not to take proactive action. Low confidence levels assert that proactive action might be an overkill because the chances of the fault occurring are low; high confidence levels might require urgent processing and high priority. Statistical and heuristic techniques are valuable in making predictions and in ascertaining our confidence in those predictions.

4. Conclusion

The MEAD infrastructure aims to provide resource-aware real-time support to middleware applications, with protection against crash, communication, partitioning and timing faults. MEAD novel capabilities include mechanisms for proactive dependability, a reliability advisor for making critical performance *vs.* reliability trade-offs at development-time, and interceptors for transparent, yet tunable fault tolerance.

References

[1] T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis", *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 419-432, October 1990.

[2] P. Narasimhan, "Trade-Offs in Providing Both Real-Time and Fault Tolerance to Middleware Applications", *Workshop on Foundations of Middleware Technologies*, Irvine, CA, November 2002.

[3] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Replica Consistency of CORBA Objects in Partitionable Distributed Systems," *Distributed Systems Engineering Journal*, vol. 4, no. 3, pp. 139-150, September 1997.

[4] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, pp. 62-68, July 1999.

[5] P. Narasimhan and S. Ratanotayanon, "Evaluating the (Un)Predictability of Real-Time CORBA Under Fault-Free and Recovery Conditions," submitted for review.

[6] S. Ratanotayanon and P. Narasimhan, "Quantifying the Impact of Faults and Recovery on Nested Middleware Applications", submitted for review.

[7] C. F. Reverte and P. Narasimhan, "Performance and Dependability Trade-Offs for Varying Checkpointing Frequencies", submitted for review.