A Middleware for Dependable Distributed Real-Time Systems

Tom Bracewell
Raytheon IDS
bracewell@raytheon.com

Priya Narasimhan
Carnegie Mellon University
priya@cs.cmu.edu

Abstract

New middleware is proposed to support the development of dependable distributed real-time systems for avionics, sensor and shipboard computing. Many of these systems require distributed computing in order to perform increasingly complex missions. They also require real-time performance, dependable software, and may face constraints that limit hardware redundancy.

Real-time performance and fault tolerance are not easily combined. A reusable approach to achieving fault tolerance in distributed real-time systems is proposed in MEAD, a Middleware for Embedded Adaptive Dependability.

Introduction

The developers of avionics, sensor and shipboard computing systems face a growing need to build software fault tolerance into distributed real-time computing environments. These systems have become more software-intense as their missions have grown in complexity. As hardware has grown more reliable, software has become the key to dependability. Software faults are the main cause of downtime in large systems [1,2,3]. Software faults cannot be entirely eliminated, and must be dealt with at run time.

Distributed computing has become necessary while the need to execute reliably in real-time has remained. Real-time performance is not easily ensured in distributed systems; and in a real-time system, a missed deadline is a type of fault.

Legacy systems often used top-level hardware redundancy and application-level software fault detection and recovery in single-thread environments to make real-time systems dependable. This approach does not carry forward into today's distributed object-oriented systems. Top-level hardware redundancy is also undesirable in systems that face severe size, power, weight and cost constraints.

A scalable, reusable, application-transparent approach to software fault tolerance in distributed real-time systems is needed. This presents many challenges.

Challenges

Dependable, Scalable, Real-time Systems. Real-time systems face increasing demands for multi-mission computing capabilities, dependability, and at the same time, lightweight, low power, compact implementations. Typical examples include shipboard, mobile and tethered sensor systems [4] and military combat systems [5]. Scalable real-time environments have become essential to the implementation of these systems. The next generation of combat systems is intended to be network-centric, and is likely to be based on adaptive and reflective middleware [6], in particular, CORBA.

A clear need has arisen for middleware that can address both the real-time and the fault tolerance requirements in these military systems. The lack of fault-tolerant real-time CORBA solutions forces developers to implement ad-hoc solutions to fault tolerance in the application layer. These brittle solutions are neither reusable nor predictably dependable. With failover still implemented at the top level, they add unnecessary hardware size, power, and cost to large real-time systems whose availability requirements could be met with much less hardware if lower-level software failover were managed against bounded real-time constraints. An ability to manage real-time fault tolerance at the processor/application level will improve recovery times, provide a means to manage hardware size, power and cost, and provide an infrastructure that supports graceful degradation, thus extending the mission life of many real time systems.

Real-time vs. fault tolerance. Real-time and fault tolerance constraints can impose conflicting requirements on a distributed system. Real-time operation requires an application to be predictable, to have bounded request processing times, and to meet specified task deadlines. This predictability is often the most important characteristic of real-time systems. In contrast, fault-tolerant operation requires that an application continue to function, even in the presence of unanticipated, potentially time-consuming events such as faults and fault recovery. Faults are often viewed as asynchronous unpredictable events that can upset a real-time system's scheduled operation. Sustained operation with consistency of application data in the face of faults is often the single most important characteristic of fault-tolerant systems. Thus, there is a fundamental conflict between the philosophies underlying the two system properties of real-time and fault tolerance. While real-time performance requires *a priori* knowledge of the system's temporal operation, fault tolerance is built on the principle that faults can and do occur unexpectedly, and that faults must be handled through some recovery mechanism whose processing time is uncertain. When both real-time and fault tolerant operation are required in the same system today, trade-offs are made at design time, not at run-time.

To combine real-time operation and fault tolerance today, middleware-based applications may try to combine ad-hoc and standardized solutions for each. The result is a brittle system that is hard to maintain and upgrade, that does not really capture the trade-offs and interactions between real-time and fault tolerance. These trade-offs and interactions should be managed by the middleware at run time - not by the application layer - and they should be based on policies defined by application developers at design time.

In the event of a fault, real-time decisions must be made as to whether the system or application should seek to perform its mission during fault recovery by operating out of spec with regard to meeting real-time deadlines or with regard to maintaining replica consistency.

Any solution to combining real-time operation and fault tolerance lies in identifying their precise conflicts and in exploring the possible trade-offs when a conflict occurs. Trade-off analysis is complicated by the fact that it involves (a) exhaustively listing the scenarios that might result from the composition of real-time and reliability; (b) assessing the risks and the degraded QoS that might arise for either real-time or reliability, or both, in certain scenarios; and (c) providing solutions for scenarios where there are direct conflicts between the requirements for timeliness and replica consistency.

Predictable recovery times. Today's ORBs have fault-detection and fault-recovery times on the order of seconds [7]. These times vary considerably and unpredictably even under reproducible conditions for very simple applications. This poses a problem for applications that must meet millisecond-level deadlines in the face of faults and recovery.

Assignment of fault tolerance properties. Fault tolerance properties include degree of replication, the physical distribution of replicas, the replication style (active, passive, etc.), the checkpointing frequency and the fault-detection frequency. In systems that use FT CORBA, these properties have been assigned with little regard for system configuration, the object's state size, the object's resource usage, etc. The choice of values for an object's properties really ought to be a resource-aware decision based additionally on the software component's reliability and recovery-time requirements. Without a pre-deployment tool to assist in deciding the appropriate values of these properties for each object, and holistically for the entire system, the ad-hoc choice of properties may cause a system to miss the mark in terms of reliability.

A run-time feedback framework would allow the development-time tool to improve its decision-making based on run-time system modifications, such as the removal or addition of resources, introduction of new applications, workstation and network upgrades.

Configurable real-time and fault tolerance levels. Combining reliability and real-time is further complicated by the fact that each system property has its own range of values. For instance, the need for replica consistency can vary from weak (where abstract server availability is more important, even if the data in some replicas is somewhat stale or inconsistent) to strong (where server availability and data integrity/consistency across all replicas are equally important). The problem in the composition of real-time behavior and reliability is that we have to take into account the varying degrees of both real-time and reliability support, and the effective quality-of-service (QoS) that will result. In cases where trade-offs exist between real-time and fault tolerance requirements, the resulting (potentially degraded) QoS needs to be well-defined so that the application developer can specify what he/she requires from the system, and have some assurance of the guarantees that the application can expect to obtain from the system under fault-free, faulty and recovery conditions.

Dependability metrics and benchmarks. There currently exists no universally accepted benchmarks or metrics for evaluating reliable systems. Software reliability has been difficult to assign, predict, model and validate. We need to develop benchmarks that allow us to characterize and quantify the success of a system or its infrastructure in providing dependability. The challenge lies in defining metrics and benchmarks in ways that can be measured and compared across test-beds, middleware, applications and dependability requirements. The practice of injecting software faults [4] into unmodified applications via CORBA will provide a mechanism for measuring the middleware's ability to improve system availability in the presence of faults.

Instrumentation and profiling. Faults often end up depleting resources; e.g., a processor crash may take a processor out of action for a while. Faults can also end up consuming resources; e.g., detecting, isolating and recovering from a fault might require excessive bandwidth and CPU time. In an environment subject to faults such as processor/process/object crashes, lost messages, missed deadlines and network partitioning, it becomes critical to manage system resources in order not to compromise the performance of the functioning part of the system. The problem lies in deciding which of the resources in the system to monitor, and how often to monitor them. Profiling of resources is by no means a zero-penalty activity; monitoring an entity is often intrusive and can impact performance. We need mechanisms for the selective surveillance of the system, to collect the right statistics at the right points in time. The profiling of the system's resource usage should ideally be a dynamic run-time decision.

Resource-aware adaptation. Part of the resource management problem lies in the modified dependability of the system following a fault. Because a fault might temporarily take a processor out of action, we need to account for the loss of the processor (and any replicas that it might have hosted) in figuring out the system's dependability. It might be possible for a system to continue to operate, although at a reduced level of dependability, while recovery is ongoing. The challenge here is to determine the appropriate level of dependability while the system is undergoing recovery in a gracefully degraded mode. The loss or addition of resources needs to be communicated to the system rapidly enough for it to realize its new level of dependability, and to adjust its end-to-end guarantees and its recovery strategy appropriately.

Proactive dependability. Faults are often viewed as unexpected events that can upset real-time operations. An ability to predict with confidence when faults will occur would be very valuable in a real-time system. Some faults caused by resource exhaustion, error accumulation and other patterns, can be highly predictable. Fault prediction is based on the premise that certain patterns of abnormal events typically manifest themselves in a system prior to the actual occurrence of the fault. The problem lies in identifying the pattern of abnormal events prior to a fault, and in the correct diagnosis of the kind of fault that is imminent. This is not a trivial exercise because it involves knowledge of where, how and when the abnormal harbingers of a fault are likely to be detected and to be recorded. Some predictors, such as memory leaks, may exhibit definite trends; other predictors are often more subtle. A deeper problem is ascertaining our confidence in the prediction, given a certain pattern of abnormal events. This information needs to be fed back into the system, before the fault occurs, in order for proactive action to be taken. Proactive dependability actions might include relocating a replica or rebooting a logical partition, while maintaining real-time operation, before a fault occurs.

Transparency with tunability. CORBA interceptors have been shown to provide a viable approach to providing transparent fault tolerance on many unmodified vendor ORBs hosting various unmodified applications. Applications that have real-time behaviors are more difficult to render fault-tolerant (transparent to the application) because application-level semantics are involved in scheduling decisions. These resource-sensitive and timing-sensitive features must be captured at the interceptor level in order to provide simultaneous fault tolerance and real-time support. APIs need to be defined to allow middleware to observe and influence these features at run-time. The goal is to provide "out-of-the-box" real-time and fault tolerance to systems and applications, without requiring substantial effort by application developers.

A Middleware for Embedded Adaptive Dependability (MEAD)[1]

A middleware for embedded adaptive dependability (MEAD) is proposed to address these challenges. MEAD will achieve its goals through a set of collaborating distributed reliable software components that create a real time fault tolerant CORBA. Key components in MEAD include the following.

Replication Manager: This component replicates application components and distributes the replicas across system processors. During development, the application developer can select fault tolerance properties for each application component through a Replication Manager GUI. This enables the Replication Manager to decide how many replicas to create, where to place the replicas, and which replication style to enforce for each replicated object. The Replication Manager assumes responsibility for supporting a certain level of dependability by ensuring a pre-specified minimum degree of replication at all times.

Hierarchical Fault Detection-Reporting-Analysis: A Local Fault Detector on each processor detects the crash of objects and processes on that processor. The Local Fault Detectors feed fault notifications into the Replication Manager, allowing it to restore the required degree of replication if a replica has crashed. The Local Fault Detectors and the Replication Manager also do fault analysis. Fault analysis can conserve bandwidth (in the case of multiple fault reports that can be combined into a

[1] Mead, the legendary ambrosia of the Vikings, was believed to endow its imbibers with immortality and reproductive capabilities (*dependability* and *replicas*).

single fault report) and 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 object-crash fault reports. The fault detection-reporting-analysis framework is structured hierarchically in order to support scalability.

Hierarchical Resource Management Framework: A Local Resource Manager on each processor monitors the resource usage of the replicas on that processor. The Local Resource Manager has instrumentation and profiling hooks at the processor, 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 a Global Resource Manager. The system-wide Global Resource Manager is aware of processor and communication resources, and their interactions across the entire system. The Global Resource Manager has a system-wide perspective of resource availability and usage across all processors. Thus, it is uniquely positioned to make load-balancing decisions to migrate objects and processes from heavily loaded processors onto relatively lightly loaded processors, in order to meet timing requirements, or to isolate resource-intensive tasks onto relatively idle processors. It can also make decisions about selectively shedding replica load, based on task criticality, under overload conditions.

RT-FT Scheduler: A Local RT-FT Scheduler on each processor monitors the scheduling and the performance of tasks on that processor. An offline scheduler computes a real-time schedule of an application's tasks ahead of run-time; this schedule provides the sequence of operations in the absence of faults. The Global RT-FT Scheduler then starts to execute the application according to this pre-determined schedule. However, to withstand dynamic system conditions, the Global Scheduler is capable of inspecting the schedule, computing the available slack time, and reallocating work to selective replicas so that an application continues to meet its original deadlines in the presence of dynamic situations (e.g., faults, loss of resources, object/process migration, fault-recovery) that may arise. Thus, the pre-computed schedule does not necessarily change 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 a FT-Hazard Analyzer, the RT-FT Scheduler predicts worst-case recovery times and the safest checkpoint and recovery points during runtime. The RT-FT Scheduler is responsible for dynamically scheduling replica migration and fault recovery.

Proactive Dependability Framework: Proactive dependability involves predicting, with some confidence, when a fault might occur, and compensating for the fault before it occurs. This allows MEAD to execute a more controlled response to a fault scenario at less risk to real time operation. Knowing, for instance, that there is an 80% chance of a specific processor crashing within the next 5 minutes will help MEAD 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 on the moribund processor within the remaining 5 minutes before the crash. This proactive recovery action is far 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. Anticipatory and preventive recovery from a fault will place fewer demands on the system than reactive recovery.

Proactive dependability hinges on the ability to predict when a fault can happen. This is possible because a fault is typically preceded by a visible pattern of abnormal behavior or resource exhaustion. Recognizing these patterns and their timelines will allows us to compute the "slack-time" available before a fault. Using statistical and heuristic techniques, we can predict, with some confidence level, the time at which certain faults might occur [8].

This requires a Local Error Detector 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 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, as explained above.

Chain of Interceptors: Interception allows us to insert the MEAD infrastructure transparently underneath any unmodified ORB. Interception 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, each transparently enhancing the ORB with some new capability. One of the authors, Priya Narasimhan, has used interceptors to enhance CORBA applications [9] with profiling, totally ordered reliable multicast, replication mechanisms, survivability and controlled thread scheduling.

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 can allow MEAD mechanisms to "kick in" only when needed. This prevents MEAD from unnecessarily impacting application performance.

Secondly, interceptors can be chained together in parallel or in series to implement crosscutting functionality. We envision the real-time fault-tolerant MEAD infrastructure having a number of interceptors, each providing additional assurances of dependability; e.g., there would be interceptors for reliable multicast, TCP/IP, real-time annotations of messages, task profiling, checkpointing and resource monitoring.

Interceptors are analogous to the concept of aspect-oriented programming (AOP) [10], where the cross-cutting concerns of a system are represented by different aspects which are then woven together to capture the system interactions. MEAD will apply the concepts of AOP in its interception technology (i) to capture and to resolve the real-time vs. fault tolerance trade-offs, and (ii) to provide for different runtime-customizable levels of real-time and fault tolerance.

Reliability Advisor: This development-time tool allows the application developer to select the best reliability configuration settings for his/her application. 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 (via heartbeat) 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, structure, reliability requirements, 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 to the engineer as to 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 improve development-time advice. This feedback component just the Local Resource Managers operating in concert with the Global Resource Manager.

Innovations Underlying MEAD

The components of the MEAD infrastructure embody many innovative concepts, some of which are described below.

Replication - Not Just for Reliability, But for Real-Time: Object/process replication has primarily been viewed as a way 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 of the object/process. MEAD goes beyond this in exploiting replication, a common fault tolerance technique, to derive better real-time behavior! With replication, there always exist redundant replicas that 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 infrastructure 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 exploits the duplicate responses/outputs of active replicas, as well as their redundant processing, in order to meet deadlines and tolerate timing 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. 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 much 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 get each application object to implement an Updateable interface to extract a state-increment. It is not always trivial for an application programmer to decide sensible state-increments. We will need to develop 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 replicated in the interests of fault tolerance. Their resource usage is monitored along with that of the applications. 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.

Periodic Software Rejuvenation: Software rejuvenation, or the periodic restart of software, is often useful in prolonging the run-time life of software that might contain memory leaks and other defects. MEAD's profiling of the resource usage of the application, along with its creation and distribution of replicas of the application objects, allows it to inject new life into long-running replicas by restarting them at appropriate moments in the system's life. Care must be taken to stagger rejuvenation across replicas so that the replicated object continues to provide service through one or the other of its operational replicas. By starting replicas at different times, and by rejuvenating them at the same frequency, we can minimize the disruption and simultaneously tolerate yet another kind of fault, i.e., memory leaks.

Fault Prediction: The algorithms for fault prediction and dependability forecasting 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 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 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.

High Performance Enablers: MEAD will work with CORBA high performance enablers as they become available. High performance enablers reduce middleware latency and jitter by eliminating unnecessary data marshalling/demarshalling and by providing zero-copy transfers. As minimized and low-footprint real-time ORBs become available, the MEAD architecture will readily adapt to them.

Conclusions

The MEAD infrastructure aims to provide a reusable, resource-aware real-time support to applications to protect against crash, communication, partitioning and timing faults. MEAD provides novel mechanisms for proactive dependability, application-transparent fault tolerance and for making trade-offs to maximize mission availability during runtime.

References

[1] Jim Gray. Why do computers stop and what can be done about it? Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems, pages 3--11. IEEE Computer Society Press, catalog number 86CH2260--8, 1986.

[2] J.N. Gray. A Census of Tandem System Availability between 1985 and 1990. IEEE Trans. Reliability, vol.39, no.4, pp.409-418, 1990.

[3] Mark Sullivan, Ram Chillarege. Software Defects and their Impact on System Availability A Study of Field Failures in Operating Systems. 21st International Symposium on Fault-Tolerant Computing (FTCS-21), 1991.

[4] D. M. Blough, T. D. Bracewell, J. Cooper, and R. Oravits. Realizing software fault tolerance in radar systems through fault-tolerant middleware and fault injection. Proceedings of the Workshop on Dependable Middleware-Based Systems, pages G117--G121, Washington, D.C., June 2002.

[5] L. DiPalma and R. Kelly. Applying CORBA in a contemporary embedded military combat system. In OMG Workshop on Real-Time and Embedded Distributed Object Computing, June 2001.

[6] D. C. Schmidt, R. E. Schantz, M. W. Masters, J. K. Cross, D. C. Sharp, and L. P. DiPalma. Adaptive and reflective middleware for network-centric combat systems. Crosstalk: The Journal of Defense Software Engineering, pages 10--16, November 2001.

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

[8] K. Vaidyanathan and Kishor S. Trivedi, "Workload-Based Estimation of Resource Exhaustion in Software Systems," Dept. of Electrical & Computer Engineering, Duke University, 1999.

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

[10] Special issue on Aspect-Oriented Programming. Communications of the ACM, 44(11), October 2001.