Decentralized Resource Management and Fault-Tolerance for Distributed CORBA Applications[*]

Carlos F. Reverte and Priya Narasimhan
Electrical & Computer Engineering Department
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213-3891
cfr@andrew.cmu.edu, priya@cs.cmu.edu

Abstract

Assigning an application's fault-tolerance properties (e.g., replication style, checkpointing frequency) statically, and in an arbitrary manner, can lead to the application not achieving its target resilience and performance. The resource management infrastructure that we have developed transparently determines a CORBA application's resource usage and its rate/pattern of invocation across a distributed system. Using this information, our infrastructure makes more informed decisions about the application's fault-tolerance properties, and dynamically adapts these decisions, as faults occur, and as resources are added and removed from the system. We have designed our prototype to be decentralized so that it is scalable and does not itself constitute a single point of failure.

1 Introduction

Middleware, such as CORBA and Java, have come to incorporate support for many "-ilities" (e.g., reliability, survivability, real-time). For CORBA middleware, there exist the Fault-Tolerant CORBA [8] and the Real-Time CORBA [10] specifications that aim to provide fault tolerance and real-time, respectively, to CORBA applications. Unfortunately, the CORBA middleware standard can only support 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. Our preliminary measurements [11] demonstrate 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 [7] are made during the composition.

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. Our previous experience with Eternal [6] has led to our development of the MEAD system in order to provide for more intelligent, resource-aware tuning of the level of real-time and/or fault tolerance needed by the application. The MEAD infrastructure aims to enhance distributed real-time 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.

MEAD consists of several run-time and development-time components, including the *Fault-Tolerance Advisor*, which allows the application deployer to select the right reliability configuration settings for his/her application. In most current systems, finding the appropriate settings for these fault-tolerance properties is mostly ad-hoc, and usually involves an unsystematic series of guesses at "good" fault-tolerance configurations, with the guesses improving (and hopefully becoming more informed) over time. Instead, MEAD makes these property assignments through the careful consideration of the CORBA application's resource usage, the application's structure, the user's reliability requirements, the desired recovery-time bounds, the system's resource limits, the number and kinds of faults required to be tolerated, etc.

This paper discusses (i) the development of MEAD's resource management/monitoring infrastructure, (ii) the issues with designing this infrastructure to be decentralized, scalable, and to avoid single points of failure, along with (iii) measurements to quantify the overhead and the performance of this infrastructure.

[*]This work has been partially supported by the NSF CAREER grant CCR-0238381 and DARPA PCES contract F33615-03-C-4110.

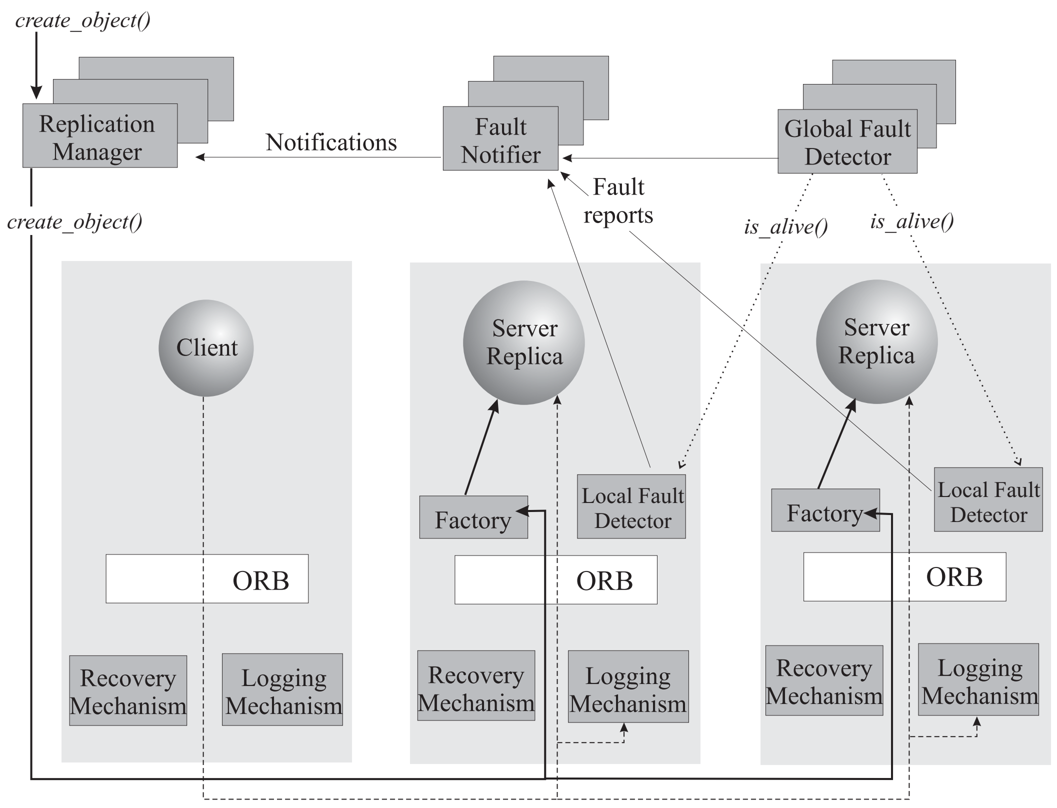

Figure 1. Architectural overview of the Fault-Tolerant CORBA standard.

2 The Fault-Tolerant CORBA Standard

The recent Fault-Tolerant CORBA [8] standard describes minimal fault-tolerant mechanisms to be included in any CORBA implementation, as well as interfaces for supporting more advanced fault-tolerance. Fig. 1 shows the architecture of the FT-CORBA specification. The Replication Manager replicates objects, and distributes the replicas across the system. Although each server replica has an individual reference, the Replication Manager fabricates an object-group reference that clients can use to contact the replicated server. The Replication Manager's functionality is achieved through the Property Manager, the Generic Factory and the Object Group Manager.

The Property Manager allows the user to configure each object's *fault-tolerance properties*, such as the such as the replication style (stateless, active, cold passive, or warm passive replication), the list of factories (locations where replicas can be created), the initial number of replicas, the minimum number of replicas to be maintained, the checkpointing frequency and the fault-detection frequency.

The Generic Factory allows users to create replicated objects in the same way that they would create unreplicated objects. The Object Group Manager allows users to control directly the creation, deletion and location of individual replicas of an application object, and is useful for expert users who wish to exercise direct control over the replication of application objects.

The Fault Detector is capable of detecting node, process and object faults. Each CORBA object inherits a `Monitorable` interface to allow the Fault Detector to determine the object's status. The Fault Detector communicates the occurrence of faults to the Fault Notifier. The Fault Detectors can be structured hierarchically, with the global replicated Fault Detector triggering the operation of local fault detectors on each node. On receiving reports of faults from the Fault Detector, the Fault Notifier filters them to eliminate duplicate reports. The Fault Notifier then distributes fault-reports to all interested parties. The Replication Manager, being a subscriber of the Fault Notifier, can initiate appropriate recovery actions on receiving fault-reports. To support state transfer and recovery, every replicated object must inherit the `Checkpointable` interface that defines operations for the retrieval and the assignment of the state of the object.

3 Fault-Tolerance Advisor

The Fault-Tolerant CORBA standard merely defines the fault-tolerance properties listed in Section 2, but does not discuss, or provide advice on, how the end-user should choose the right settings for these properties. Thus, in most fault-tolerant 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 the position of making 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 $vs.$ 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 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 Decentralized Resource Management

We had a number of objectives in mind when designing the architecture of MEAD. Constraining the number of objects, processes or nodes would limit the scope and applicability of our solution. Furthermore, we want to avoid single points of failure, not just for the application, but for the MEAD infrastructure itself. Through its decentralized resource-management architecture, MEAD supports both scalability and true fault-tolerance.

Resource management in a distributed fault-tolerant system necessarily involves handling the possibly degraded resilience of the system following a fault. Because a fault can temporarily take a node out of action, MEAD needs to account for the loss of the node (and any replicas of objects or processes that it might have hosted) in figuring out the system's fault-tolerance. It might be possible for the system to continue to operate, albeit at a reduced level of resilience, while fault-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. MEAD needs to discover the loss or addition of system resources rapidly enough to re-compute and re-enforce the system's new level of resilience, and to adjust the system's end-to-end guarantees and recovery strategy appropriately.

We note that, while we target CORBA applications for this research, *the MEAD architecture and infrastructure are independent of CORBA, and can be equally used for non-CORBA applications*. In the context of this paper, we make the follwing general set of assumptions:

- Operation in a distributed, asynchronous system,
- Deterministic, reproducible behavior of the CORBA application and the ORB,
- Independent failures across distinct nodes and processes, and
- Fault model encompassing node-crash, process-crash and message-loss faults (the other kinds of faults that MEAD tolerates are not discussed in this paper).

4.1 Architecture

We implement the Fault-Tolerance Advisor over a fully decentralized infrastructure. As shown in Figure 2, 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

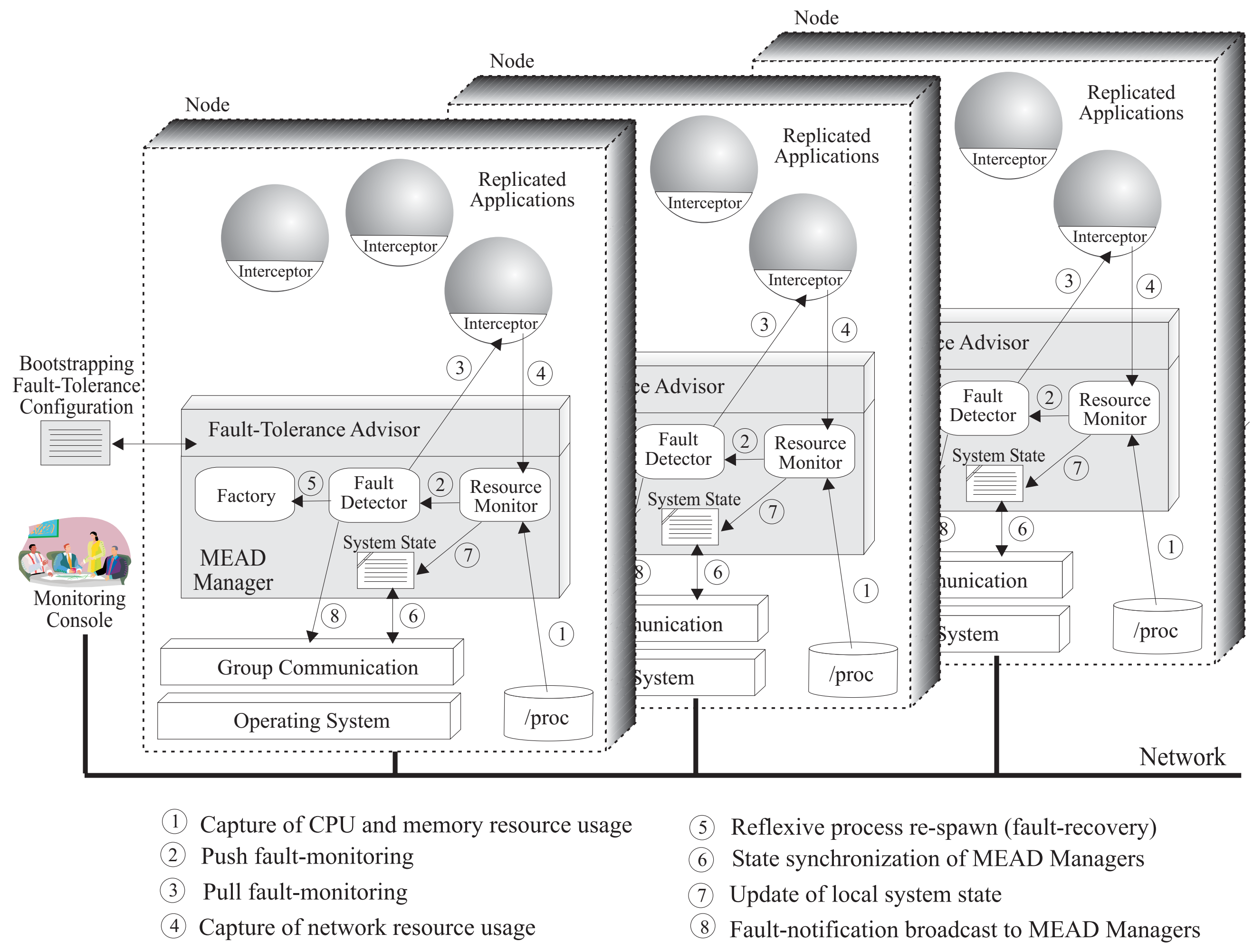
Node
Node
Node
Replicated Applications
Interceptor
Bootstrapping Fault-Tolerance Configuration
Fault-Tolerance Advisor
Factory
Fault Detector
Resource Monitor
System State
MEAD Manager
Monitoring Console
Group Communication
Operating System
/proc
Network
1 Capture of CPU and memory resource usage
2 Push 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 2. The MEAD decentralized resource management architecture and the Fault-Tolerance Advisor.

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 [1] 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 [1]. 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 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.

Apart from synchronizing their states, the MEAD Managers on the different nodes perform their duties without requiring coordination, and therefore, with minimal network communication. This provides for a low overhead fault-tolerant infrastructure with scalability as the load and the number of nodes increase.

4.2 Resource Monitoring

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

[1]The MEAD Managers in the system use a symmetric, rather than active, style of replication because they have the same state through their periodic synchronization, but do not perform the same set of actions; this avoids unduly high network overhead.

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 [2] 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 [3]. 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 multiple approaches are possible for the accounting of network usage. One simple method is to use wrapper functions for system network calls that count the network traffic (number of bytes sent and received) over those calls. This would give us a good picture of network activity with the minimum possible overhead; unfortunately, it also requires recompilation of the application, and may require additional effort on the part of the application programmer. Another option would be to use a network proxy to serve as a gateway that logs incoming and outgoing network data. Proxies can provide accurate traffic accounting transparently to the application programmer, but are expensive in terms of overhead, because they require context switches for every network-related operation, and might increase latency depending on buffering.

4.3 Use of Interceptors

The best compromise between transparency and overhead for monitoring network traffic is achieved by using interceptors to capture network calls and to count the number of incoming and outgoing bytes. CORBA incorporates support for interception through the Portable Interceptors mechanism [9]. 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 [5] 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 calculate the incoming and outgoing bytes at each instance of network communication, before invoking the default implementation of the function.

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 for immediate, inexpensive fault detection because it is uniquely positioned to detect the closure of sockets between clients and servers. Regardless of our Interceptor-based approach, monitoring network traffic is, in general, more expensive than monitoring CPU and memory usage because the overhead increases with the amount of network traffic.

As seen from Figure 2, the Interceptor periodically communicates the network statistics to the MEAD Manager's resource monitor over a private socket connection; this push of information from the Interceptor is triggered by the occurrence of specific events, and also by timeouts. During periods of heavy network traffic, the Interceptor can send more messages to the resource monitor in order to trigger fault-tolerance advising for high-bandwidth conditions; the Interceptor can also reduce the amount of communication to the resource monitor to produce lower overhead. During periods of no network traffic, the Interceptor refrains from communicating as much with the resource monitor because there are no significant events of interest. Timeout-driven communication between the Interceptor and the MEAD resource monitor is also possible; however, timers are difficult to implement in the Interceptor library because the `SIGALRM` signal produced by timers in the UNIX operating system can interfere with the intercepted process' signal handling.

4.4 Fault Detection

A *node-crash fault* is considered to have happened if the hardware or operating system fails on a specific node, or if the MEAD manager or the group communication service on that node crashes. In either case, the end-result is that the node is no longer functional, for the purposes of replicating the application.

The failure of a node's hardware or operating system manifests itself as a node-crash fault that MEAD can readily detect by exploiting the facilities of the underlying group communication system. The Spread group membership service maintains up-to-date information about the number of functional nodes that are running the Spread system. By connecting to the Spread daemon running on its local node, the MEAD Manager on that node can receive membership notifications about the addition or removal of nodes from the system.

To verify that the MEAD Managers are still functional, they can be configured either to broadcast heartbeat messages to the MEAD group or to invoke each other across the network. The MEAD resource monitor's broadcast messages serve as a heartbeat for MEAD Manager fault-detection. Fault detection with heartbeat broadcasts is easier to configure and less expensive than the inter-Manager invocation scheme. Since the resource state is synchronized across the system, the broadcast rate for any node can be determined and a node-crash fault is flagged if the hearbeat delay reaches a timeout.

A *process-crash fault* occurs when a process fails, and also if a process hangs and behaves unexpectedly by failing to produce a response within a given period of time. Local process-crash faults can be detected by periodically checking if their UNIX process identifiers are still valid; clearly, not only is this expensive, but it also depends on how often the checks are done. Instead, MEAD exploits its Interceptor as a fault-detector with much lower delay and overhead. At process launch-time, the Interceptor establishes a socket connection to its local MEAD Manager. If the intercepted process crashes, the connection between the Interceptor and the MEAD Manager is forcibly closed by the kernel. The MEAD Manager, which awaits events on all of its socket connections, will detect an error condition on one of its sockets; by associating this connection with its associated process, the MEAD Manager can transparently detect process-crash faults quickly, when they happen.

To detect hung processes transparently, we could possibly use the application's execution patterns or network traffic. However, because process execution is often unpredictable, process-hang detection is better accomplished through application-level support by requiring every CORBA object to support the `Monitorable` interface containing the `is_alive()` method. Clearly, while the invocation of this method is transparent to the application, the existence of the mechanism is no longer transparent to application programmers. The MEAD Manager's fault-detector calls the `is_alive()` function on every locally hosted CORBA process at a frequency that is configurable by the MEAD Fault-Tolerance Advisor.

Another advantage of our using a reliable group communication service is its tolerance to *message-loss faults* through retransmissions. This is transparently handled at the Spread daemon level, and is conveniently not exposed to the MEAD Managers. Thus, message-loss faults do not have to be explicitly handled by MEAD.

4.5 Failover and Recovery

MEAD tolerates node-crash and process-crash faults by replicating CORBA processes, by redirecting client requests to functioning replicas, and by utilizing extra resources to enable functional nodes to take over the responsibilities of a crashed node.

Recovery of a failed node is fairly straightforward – a node with operating system or transient hardware faults is often recoverable with a reboot. Of course, the Spread daemon and the MEAD Manager (and their states) need to be restored before the node can be considered fully functional.

Recovery of failed processes (due to either a node-crash or a process-crash fault) involves different actions. When a fault is detected, the MEAD infrastructure will take immediate steps to restore service and recover from the fault. When MEAD detects a node-crash fault, if primary replicas of passively replicated processes were running on the failed node, MEAD promotes backup replicas on the other functional nodes to be the new primaries. The location of new primaries is chosen according to a deterministic recovery heuristic. In addition, MEAD ensures that clients of these passively replicated processes perform a fail-over to the new primary replicas. Note that primary re-election and fail-over do not occur for active replication.

A node-crash fault can lead to possibly multiple process-crash faults associated with its hosted processes. Thus, due to a node-crash fault, some CORBA processes might have degraded resilience because their replicas might have crashed, too. To perform effective recovery, the MEAD Manager that detects a process-crash fault broadcasts fault-notifications to MEAD Managers on other nodes. Based on feedback from the Fault-Tolerance Advisor, the MEAD system might launch new replicas on available nodes in order to restore the resilience of the application. The loading of nodes, and the distribution of primary replicas are taken into account in this repopulation process in order to balance system load and reduce the maximum potential impact of any future failure.

4.6 Enabling Fault-Tolerance Configuration

MEAD provides an interface that end-users and system administrators can use for the static or the dynamic configuration of the application's fault-tolerance properties. The

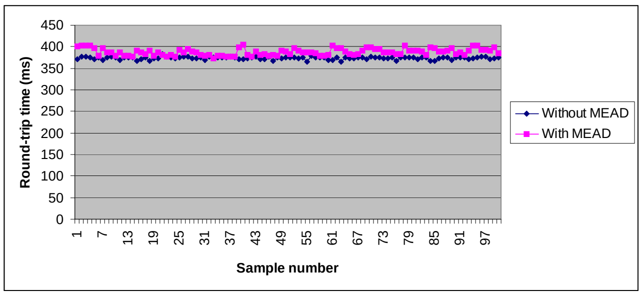

Figure 3. Impact of MEAD's resource monitoring on the performance of the application.

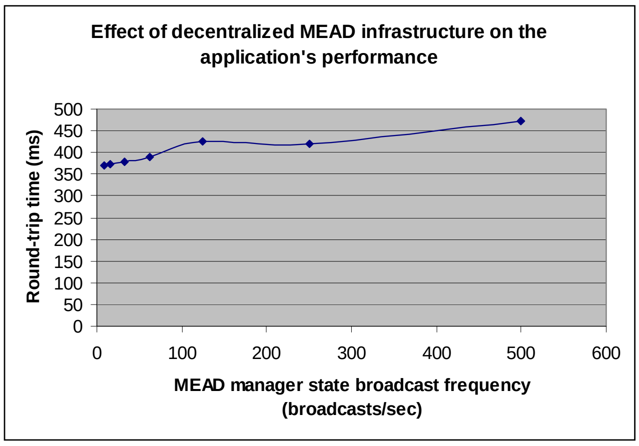

Figure 4. Variation of the round-trip time with the frequency of broadcasting the MEAD Manager state.

baseline operating mode involves the Fault-Tolerance Advisor extracting the fault-tolerance properties from a configuration file at the time of bootstrapping.

Run-time Fault Tolerance Advisors can be implemented in two ways. The Fault-Tolerance Advisor can be replicated like a CORBA service by the MEAD infrastructure. Note that there will not necessarily be an Advisor replica running on each node; the Advisor replicas will be distributed across a select group of nodes. The replicated Advisor can subscribe to MEAD's resource monitoring and fault-notification broadcasts to gather system state information. Based on that information, it can generate fault-tolerance advice and configuration updates and send them to the MEAD Managers through a special advising group. This approach would be flexible and would allow for different Advisors to be compared against each other, or even swapped in and out of a running system without disruption. Multiple Advisors, each customized for different application types, could run simultaneously and be responsible for specific, non-overlapping applications. If the replicated Advisor crashes, the MEAD infrastructure can continue to operate normally; only the ability to configure fault-tolerance automatically will be temporarily suspended until the Advisor service is restored. Such an Advisor service would be straightforward to implement, but the computation and the information involved in generating the advice would not be spread across all the nodes; also, each node does not have the capability to influence local advising decisions.

The alternative approach that we employ is to create a distributed Advisor that is integrated with each MEAD Manager. The Advisor component is automatically symmetrically replicated across all nodes, and operates on the Manager's synchronized system state to generate the advice. Rather than broadcast its configuration updates, each Advisor directly updates the state of its local MEAD manager. This is more difficult to implement, but more effective in terms of scalability and overhead.

5 Empirical Evaluation

Because the focus of this paper is on the decentralized MEAD infrastructure, rather than the Fault-Tolerance Advisor, we focus on our empirical evaluation of the resource monitoring and state synchronization, assuming a static fault-tolerance configuration.

Our test-bed consisted of seven nodes (1GHz processor, 256MB RAM, running RedHat Linux version 9) connected over a 100 Mbps Ethernet. The underlying group communication facilities were provided through the Spread system version 3.17.1. Each node in the system runs a Spread daemon and a MEAD Manager.

Our CORBA test applications are built with the TAO ORB (ACE version 5.2.8 and TAO version 1.2.8). The application mimics a distributed unmanned aerial vehicle (UAV) application, and consists of a simple client that connects to a server on another node and transfers 64KB images. To load MEAD's resource monitoring system and to impose infrastructural traffic on the network, each MEAD Manager launches and monitors 15 dummy processes on its node. Each Manager collects local resource usage data and broadcasts it to the Manager group at a fixed period with unsynchronized phase.

We vary the MEAD resource monitor's broadcast rate, and measure the round-trip time for a block of data transfers between the test client and server; we record the round-trip time for fifty 64KB images. This round-trip time is on the order of half a second. We then conducted one hundred trials spaced one second apart, in an attempt to reduce any bias from low-frequency nodes and network load variations. After each set of trials, we shut down the entire MEAD system, and then restart the system with a different monitor broadcast rate before repeating the experiment. We choose broadcast periods ranging from 125ms to 2ms, halving the period in consecutive runs (125ms, 63ms, 32ms, ..., 2ms).

Figure 3 shows the effect of adding resource monitoring to an application. The graph shows the round-trip times of the application with no monitoring and with monitoring with 62 broadcasts/sec to transmit the MEAD Manager system state. With MEAD's resource monitoring, the round-trip time degrades by only 4.5%, even for the high broadcast rate that we intentionally chose.

Figure 4 shows the effect of increasing the MEAD resource monitor's broadcast frequency on the round-trip time. The graph shows that the round-trip time increases linearly, with low impact on performance at a high frequency (50 broadcasts/sec) of broadcasts.

While MEAD can support high rates of broadcasting the monitor state, they are unlikely to be necessary because the Fault-Tolerance Advisor (which uses this system state) is occurs at a relatively low or moderate rate. Thus, MEAD's overhead in the presence of fault-tolerance advising is likely to be much lower.

6 Related Work

Current fault-tolerant CORBA systems can be classified into the integration, service or interception approaches. The integration approach to fault-tolerant CORBA incorporates the reliability mechanisms into the ORB infrastructure, resulting in modified, non-standard, CORBA implementations. The specific examples of fault-tolerant CORBA systems are discussed in greater detail in [4].

Examples of the integration approach include Electra, AQuA, Maestro and Orbix+Isis. The service approach to fault-tolerant CORBA provides reliability through a collection of CORBA objects which are explicitly used by the application. Examples of the service approach include OGS, FRIENDS, DOORS, IRL, NewTOP and FTS. The interception approach to fault-tolerant CORBA provides reliability as a transparent add-on to existing CORBA applications by means of an interceptor that can add new services to existing ORBs. The Eternal and the Immune systems provide transparent reliability and survivability, respectively, to unmodified CORBA applications running over unmodified ORBs using the interception approach. More recently, fault tolerant features have been added to the real-time CORBA implementation, TAO, by adopting the the semi-active replication style pioneered by Delta-4/XPA.

To the best of our knowledge, none of these systems or approaches completely identifies, or addresses, the real-time vs. fault tolerance conflicts for middleware systems, or provides support for the easy and dynamic configuration of fault-tolerance properties in a resource-aware manner.

7 Conclusion

Most fault-tolerant CORBA systems arbitrarily assign values to properties such as an object's replication style, its checkpointing frequency, *etc.* 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.

The MEAD system encompasses a decentralized infrastructure to obtain the application's resource usage across the distributed system. Exploiting this information, MEAD's Fault-Tolerance Advisor makes informed decisions about the application's fault-tolerance configuration, and dynamically adapts these decisions to run-time events, such as faults. We describe MEAD's resource management infrastructure, its design for scalability and no single points of failure, and then evaluate its overhead for test applications.

References

[1] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 327–336, New York, NY, June 2000.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, 2003.

[3] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *Proceedings of the Winter USENIX Conference*, Jan. 1991.

[4] P. Felber and P. Narasimhan. Experiences, approaches and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, To appear 2003.

[5] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

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

[7] P. Narasimhan. Trade-offs between real-time and fault-tolerance for middleware applications. In *Workshop on Foundations of Middleware Technologies*, Irvine, CA, November 2002.

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

[9] Object Management Group. Portable Interceptors. OMG Technical Committee Document formal/2001-12-25, December 2001.

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

[11] S. Ratanotayanon and P. Narasimhan. Estimating fault-detection and fail-over times for nested real-time CORBA applications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 2003.