Designing Fault tolerant Mission-Critical Middleware Infrastructure for Distributed Real-time and Embedded Systems[⋆]

Matthew Gillen[1], Paul Rubel[1], Jaiganesh Balasubramanian[2], Aaron Paulos[3], Joseph Loyall[1], Aniruddha Gokhale[⋆⋆2], Priya Narasimhan[3], and Richard Schantz[1]

[1] BBN Technologies, Cambridge, MA 02138, USA
[2] Dept of EECS, Vanderbilt University, Nashville, TN 37235, USA
[3] Dept. of ECE, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. Fault tolerance is a crucial design consideration for mission-critical distributed real-time and embedded (DRE) systems, such as avionics mission computing systems, and supervisory control and data acquisition systems. Increasingly more of these systems are created using emerging middleware standards, such as publish-subscribe communication services and component based architectures. Most previous R&D efforts in fault tolerance middleware has focused on client-server object systems. Application of this research to concrete domains frequently requires specialization, hand-tailoring, and customization to accommodate for the real world challenges of these systems, including nondeterminism, scale, and interaction patterns.

This paper describes our current applied R&D efforts to develop fault tolerance technology for a specific piece of mission critical DRE infrastructure, a dynamic resource manager, built using CORBA components and exhibiting characteristics representative of real-world DRE systems. This paper makes three contributions to the design and implementation of fault tolerant support in DRE system. First, we describe the fault tolerance challenges presented by these systems, including support for component infrastructure, mixed mode FT techniques (supporting active and passive fault tolerance), support for nondeterminism, issues of scale (limiting the spread of the FT infrastructure to those elements that require it), and the need for predictable and bounded recovery times. Second, we describe the design of our fault tolerant DRE architecture. Finally, we illustrate the fault recovery times in our FT DRE infrastructure in the presence of faults for a representative domain, a mission critical computing environment.

Keywords: Fault tolerance, Mission-critical Middleware

[⋆] Work supported by the DARPA ARMS Program
[⋆⋆] Contact Author: a.gokhale@vanderbilt.edu

1 Introduction

An emerging class of enterprise distributed real-time and embedded (DRE) systems, such as those found in aerospace, defense, telecommunications and healthcare domains, are increasingly being realized as compositions of loosely coupled, interacting services. Each service is designed as an assembly of one or more components implemented in potentially different languages and hosted on different hardware and software platforms. These DRE systems have stringent quality of service (QoS) requirements both in terms of performance attributes, such as low and bounded latencies, and dependability attributes, such as high availability, reliability, and fast and bounded fault detection and recovery timings.

Advances in component middleware platforms, such as Enterprise Java Beans [1] and the CORBA Component Model (CCM) [2], have made it possible to realize the composable structure of these DRE systems. Additionally, these middleware provide effective reuse of the core intellectual property (i.e., the "business logic") of an enterprise. Although these middleware platforms have several desired characteristics, they are not yet suitable for providing the QoS guarantees, particularly fault tolerance (FT), to the class of DRE systems we are interested in for the following reasons:

Limitations of FT R&D state of the art. A substantial amount of research in fault tolerant distributed systems has predominantly concentrated on providing fault tolerance solutions to intrinsically homogeneous, two-tier client-server systems with request-response semantics or cluster-based server systems with transactional semantics. These research artifacts most often assume single language, single platform systems, which when incorporated in the middleware form point solutions, limit reuse and are too restrictive for these DRE systems.

Lack of support for mix-mode FT semantics. DRE systems of interest to us require mix mode FT wherein parts of the system may require ultra high availability calling for solutions that require active replication while other parts of the systems may demand passive forms of replication to overcome issues with non-determinism. Standardized solutions to fault tolerance, such as FT-CORBA [3], provide a *one-size-fits-all* approach, which do not support the mix-mode FT semantics required by the DRE systems. Moreover, these solutions do not seamlessly apply to component-oriented systems.

Lack of support for heterogeneity. The current state of the art in component middleware do not yet support multi-language (i.e., C++ and Java), multi-platform (i.e., CORBA and non-CORBA) and multi-paradigm (i.e., mix of component, object, and procedural models) heterogeneity. Existing solutions that address this concern tend to become point solutions.

There is thus a general lack of reusable, configurable and robust middleware solutions that address the problems outlined above. The contributions of this paper are manifold. We demonstrate how fault tolerance can be designed for mission critical DRE systems, which have stringent, predictable requirements on fault detection and recovery timings. Our fault tolerance solution provides reusable capabilities to handle DRE systems, which are multi-language; with elements requiring different levels and types of fault tolerance (some need constant availability, others less so); includes databases, clients, servers, and client/servers; and a combination of CORBA objects, CORBA components, and non-CORBA applications.

The rest of the paper is organized as follows: Section 2 describes the challenges in designing the fault tolerance solutions for these DRE systems; Section 3 describes the design and implementation choices we made in our fault tolerant architecture; Section 4 describes experimental results; Section 5 describes related research; and Section 6 provides concluding remarks and future research.

2 Design Challenges for Fault-tolerant Mission-Critical Systems

In this section we describe the challenges faced designing and developing fault tolerant infrastructure for mission critical DRE systems. To better enable us to describe the challenges and our solutions (described in Section 3), we focus on a representative DRE system we are currently working on as part of the DARPA Adaptive and Reflective Middleware (ARMS) program. The DARPA ARMS program is seeking solutions to dynamic resource management in mission critical systems.

2.1 DRE System Case Study

Our DRE system provides dynamic resource management capabilities to applications running in a large mission critical environment. Since the application mix in our domain comprises mission critical entities, the resource management infrastructure is responsible for making real-time resource management decisions. Figure 1 illustrates our DRE dynamic resource manager [4,5]. It is a multi-layered architecture that translates high level operational requirements into the sets of applications that must be run and their real-time requirements, and maps the applications (grouped into end-to-end application strings) over computational and communication resources. We will refer to this multi layered architecture as a Multi-Layered Resource Management (MLRM) framework. The MLRM system adjusts the resource allocations as resource availability changes (e.g., due to failures), to accommodate for system load, or to satisfy changing operational requirements.

The MLRM is implemented using a number of base technologies including CIAO [6], which is a C++ implementation of the lightweight CORBA Component Model (CCM) [7]; OpenCCM [8], which is a Java implementation of CCM; real-time CORBA ORBs, such as TAO [9]; Java ORBs, such as JacORB [10], and non-CORBA technologies, such as MySQL databases, and the MEAD [11] replica management framework.

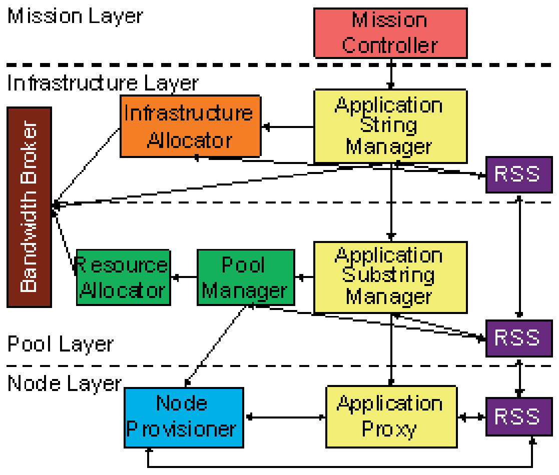

Fig. 1: MLRM Architecture

2.2 Fault Model for MLRM

In designing our FT solution we assume that all faults are fail-stop at the process level, *i.e.*, when an application process fails it stops communicating and does not obstruct the normal functioning of other unrelated applications. Network and host failures can be seen as a collection of process failures on the network or host that has failed. Some examples of failures that we will tolerate include power being disrupted to a host, an application crashing, or a data center being destroyed. Some examples of failures that we do not currently tolerate are network partitioning, where only parts of the network fail, leaving two groups of uncoordinated replicas, and malicious or Byzantine [12] failures where a process may intentionally attempt to deceive other members or misrepresent data.

2.3 FT Challenges in DRE Systems

The MLRM is a crucial piece of our mission critical system and must be made fault tolerant, however, illustrates a number of *crosscutting concerns* described below that affect the strategies that we must employ in order to make it fault tolerant:

1. Nearly continuous availability. The MLRM must be nearly continuously operational. This means that faults must be *masked* or any failover time must be very small. This motivated the use of active replication wherever possible since it offers continuous availability. However, since active replication has limitations in the case of non determinism, this strategy must interwork seamlessly with other solutions that deal with non-determinism.

2. Dealing with non-determinism. Some of the elements of MLRM, such as the bandwidth broker (BB) are fairly deterministic, while others are necessarily nondeterministic, including multi-threading, file I/O, and other nondeterministic functionality. The replication styles used must therefore address these challenges.

3. Layered FT capabilities with minimal impact. In the layered architecture of the MLRM shown in Figure 1, the top-level MLRM elements had to be made fault tolerant without requiring all elements in the system to be affected. The MLRM system has hundreds of elements at the bottom level, i.e., at the node layer, as well as potentially thousands of applications that it is deploying. These elements should be affected by the fault tolerance infrastructure of the MLRM, e.g., by being forced to use group communication systems (GCS), required by the applications that interact with the top-level MLRM elements, but unnecessary for those that do not. This necessitates co-existence of GCS and non-GCS frameworks.

4. Non-traditional communication patterns. The MLRM has multi-tier and peer-to-peer communication patterns. That is, many of the elements act as both clients and servers throughout the MLRM operation, sometimes simultaneously. The FT design must therefore be able to handle interactions that are much more complicated than traditional client-server CORBA systems. For example, one unexpected interaction that required special attention was a case where two processes had *both* client and server connections to each other (i.e. there was a cyclic dependency between the applications caused by callbacks).

5. Multi-language, multi-platform environment. The MLRM is implemented as a mix of C++ and Java elements. Moreover, the system is made up of elements using the component middleware technologies, such as the C++ CIAO and Java OpenCCM implementations, elements that use both CORBA and non-CORBA elements, such as MySQL databases and MEAD, which is a library that uses procedural semantics. The FT design must account for this heterogeneity when dealing with replica reconstitution and state synchronization in the presence of GCS and non-GCS communication patterns.

6. Statefulness of MLRM. When starting a new replica it is imperative that the new replica is prepared to behave exactly as an existing replica. Even if the application being replicated is stateless the middleware does need to transfer state from existing to new replicas. Thus, when a new replica is added into a group, there is a need for consistent state management, which works seamlessly in the face of heterogeneity.

7. Fault detection in multi layered architecture. When we inject failures (say during testing) into the system the time of fault injection is easy to note. Differentiating detection from recovery, however, is not as easy. Our FT middleware solution makes use of the Spread GCS system [13] for notification of failure. The middleware level of our system becomes aware of the failure when Spread notifies existing replicas. However, Spread must have been aware of the failure at least a small amount of time before it reported to the FT middleware so getting a precise detection time is non-trivial.

Section 3 describes our reusable FT infrastructure that addresses the challenges outlined above.

3 Architecture of the Fault Tolerant DRE Middleware

Providing fault-tolerance for our MLRM middleware required a careful design and a number of improvements to move us from a non-fault-tolerant solution to one that could survive multiple cascading failures. A number of areas that needed improvement became apparent as we developed both the individual pieces making up the MLRM functionality as well as the interactions between the MLRM pieces. This section describes the individual improvements we made and how they worked together to provide a comprehensive solution.

Our FT design for the MLRM is incorporated into the conventional layers of middleware depicted in Figure 2. The rest of the section describes how individual pieces of our complete FT-MLRM map to these different middleware layers and resolves the FT challenges manifested in systems such as the dynamic resource manager.

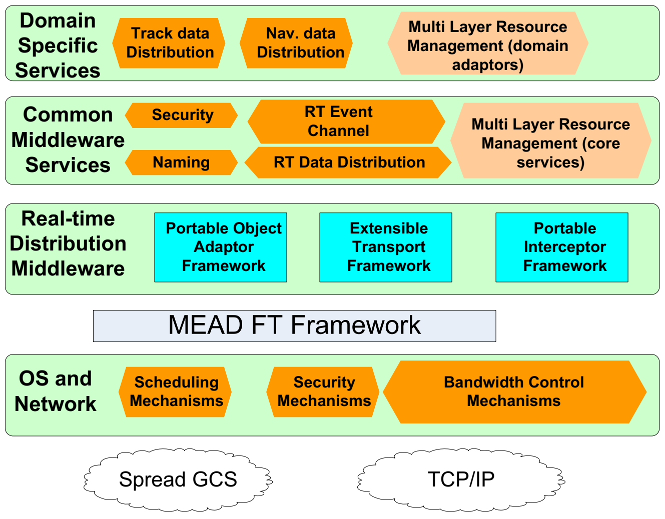
Domain Specific Services
Track data Distribution
Nav. data Distribution
Multi Layer Resource Management (domain adaptors)
Common Middleware Services
Security
Naming
RT Event Channel
RT Data Distribution
Multi Layer Resource Management (core services)
Real-time Distribution Middleware
Portable Object Adaptor Framework
Extensible Transport Framework
Portable Interceptor Framework
MEAD FT Framework
OS and Network
Scheduling Mechanisms
Security Mechanisms
Bandwidth Control Mechanisms
Spread GCS
TCP/IP

Fig. 2: MLRM Functionality Mapping on Middleware Layers

As explained in Section 2.1, the MLRM and our FT solution uses a number of individual base technologies, such as CIAO, OpenCCM, JacORB, TAO, MEAD and MySQL. A number of middleware improvements were necessary to make a foundation strong enough to stand up a fault-tolerant MLRM. At the fault-tolerant middleware layer we needed to improve the MEAD infrastructure to better deal with application state transfer (necessary for replica reconstitution), duplicate message detection (necessary when multiple replicas may all reply to an incoming message), peer-to-peer calling conventions, and support for multi-ORB and multi-language solutions. Another improvement made to MEAD, not used by replicas, but used by processes interacting with replicas was support for limiting the use of group communications via a "gateway." Above MEAD, at the component middleware layer, improvements were necessary to support state exchange for components. Moving further towards the application layer we made small changes to MySQL clustering to better support the fault-model in which the MLRM is deployed. Our enhancements result in a reusable framework with fast and bounded failure recovery times.

3.1 Co-existence of Protocols: Singleton Gateway Approach

In order to keep replicas consistent, our fault-tolerance infrastructure ensures that messages are reliably delivered to each replica in the same order, using a Group Communication System (GCS) system. Any interactions with a replica, after the initial middleware layer (i.e., CCM) bootstrapping, pass through GCS. In the simple case of a client interacting with a replicated server, all interactions are necessarily over GCS. The existing FT Middleware solution forced all communications in a fault-tolerant system to use GCS.

Our situation is more complex in the MLRM case, a three-layered hierarchical architecture, shown in Figure 1. We are (currently) replicating only the top layer. Due to the need for consistency, elements in the middle layer interact with the top layer using GCS. However, since the middle and bottom layers are not replicated, they do not require the same consistency guarantees and do not need to use GCS to communicate with one another. Furthermore, we do not want to force the extra overhead of the GCS infrastructure onto the lowest layer, which can include hundreds of components, each with real-time performance requirements and frequent communication needs. To provide acceptable performance for components that require GCS and to more efficiently use resources, we implemented functionality that limits the use of GCS to where it is strictly necessary. We call these applications that use both GCS and non-GCS communication as "gateways".

These are different from "gateways" used elsewhere in fault tolerance solutions, since we don't introduce any extra architectural elements. Instead, as illustrated in Figure 3, we simply introduce an extra role that existing elements take on. Replicated elements all communicate group communication. Most non-replicated elements communicate using non-group communication, such as TCP. Those elements that interact directly with replicated elements speak both group communication (with the replicated elements) and non-group communication with everything else. To implement this, we modified the interception approach used by our MEAD software (explained next), which intercepts all interprocess communication and directs it over group communication (Spread), to support both the default communication and group communication and to selectively route messages based on whether it is going to a replicated element.

Implementing this solution turned out to be more difficult than initially anticipated. Up to that point we had replicated only single-threaded applications. Without the constraints imposed by being a replica many gateway applications ended up being multi-threaded. We improved our middleware to be able to deal with multi-threaded applications. This allowed our gateways to work but also allowed our passive replication scheme to be used in multi-threaded applications.

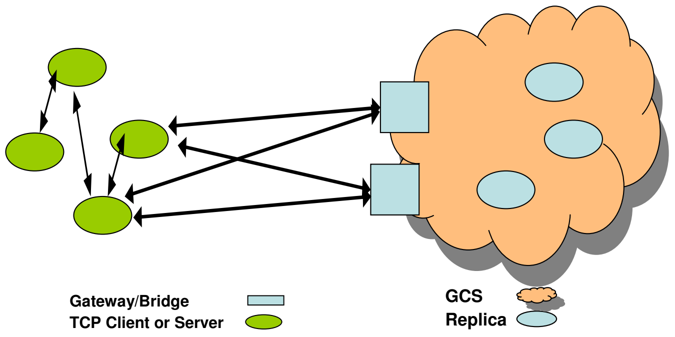

Fig. 3: Gateway Approach to Mix-mode Communication

3.2 MEAD Enhancements

The MEAD system [11] that we have developed at Carnegie Mellon University is a fault-tolerance infrastructure that supports the transparent replication and recovery of components in stateful CORBA applications. In order to ensure the system-wide consistency of replicas of every CORBA component, MEAD exploits the guarantees of the underlying Spread group communication system [13] to convey the replicated application's messages. In addition, to ensure exactly-once semantics for processing requests and replies in the presence of replication, MEAD performs the detection and suppression of duplicate requests and replies before delivering messages to the application. Below we describe our enhancements to MEAD to resolve the challenges described in Section 2.

ORB and CCM interactions. Originally, MEAD had support for CORBA 2, non-CCM style interactions. In order to interoperate with the component world a number of changes were necessary. In a CCM context the ORB may be up before the hosted objects are ready to interact with the rest of the system. The ORB is ready for bookkeeping interactions rather than application messages. Initially MEAD assumed that when the ORB was up it was ready to receive messages. In order to ensure this was the case we implemented a notification from the ORB indicating all its components were ready. This allowed MEAD to make invocations to get and set the state when they could be processed by the ORB.

Multiplatform support. Initially our FT middleware solution worked only with the TAO C++ ORB. This was not an acceptable solution as pieces of the system, which needed to be replicated, also use the JacORB Java ORB. We upgraded our middleware to support both TAO/CIAO/C++ and also JacORB/Java. This allowed interactions between replicated C++ components and replicated Java objects.

Multiprotocol support. MEAD exploits library interpositioning to intercept a process' network system calls by using the dynamic linker's run-time support. Using the `LD_PRELOAD` environment variable, we load the MEAD system as a shared object library into the address space of each GCS-hosted component and of the singleton, ahead of all of the other dynamically linked system and application libraries. The MEAD library overrides some socket and network functions (such as `socket`, `connect`, `bind`, `read`, `write`, etc.) to perform the transparent re-routing of the CORBA application's IIOP messages over the Spread GCS [13].

The MEAD way of performing the TCP-GCS bridging introduces additional considerations for the implementation of its infrastructure. The singleton that forms the bridge between unreplicated TCP clients and replicated servers must also support GCS clients, albeit in a mutually exclusive manner. It is perfectly possible for the end-to-end operations passing through the singleton to arise from either TCP or GCS clients.

Because MEAD operates at the network/socket level of the CORBA application (in user space, underneath the ORB, but not within the kernel), its operation hinges upon the sequence of network-level interactions that CORBA clients and server undergo during connection establishment, during normal communication of requests and replies, and during connection teardown. For synchronous CORBA client-server communication, the MEAD library underneath either the GCS client or server is typically driven off interrupt-driven I/O (through a `select` system-call with a timeout value set to `NULL`, indicating that call blocks until interrupted by a signal indicating incoming I/O) whenever it receives notifications of received GCS messages that are waiting for processing.

The singleton infrastructure is assigned to a separate group of its own, which allows it to communicate over the GCS protocols within the replicated domain. The singleton's addressing information (a group identifier for GCS access and an IP address and port number for TCP access) is made available to its GCS and TCP clients.

Because MEAD conveys the application's messages through GCS, any component supported by MEAD will have its connections funneled/mapped by MEAD onto a single socket connection to the local Spread GCS daemon. Effectively, when a CORBA client/server/singleton is supported over MEAD, there is only way into and out of that component through MEAD, and this "GCS channel" (accessed via a `SP_receive` blocking call on the Spread GCS API) supplies only totally ordered messages. As long as it delivers (to its hosted clients and server) only the ordered sequence of messages that it receives off this GCS channel, MEAD can ensure replica consistency because every MEAD-hosted server replica will "see" the same requests and replies.

Duplicate suppression. Another update to MEAD involved duplicate suppression. When multiple replicas reply, MEAD must suppress duplicate messages from reaching the intended recipient. Initially an application could be either a client or a server. In a peer-to-peer environment this assumption was too constraining and was removed. Furthermore, we also enabled this duplicate suppression information to be moved from an existing replica to a new replica as part of the state.

Dealing with non determinism. Any connection from the TCP domain to the singleton will take the form of a separate socket connection at the MEAD infrastructure. These sockets are channels of communication that are distinct from MEAD's preferred GCS channel, into and out of the singleton. In fact, from MEAD's viewpoint, every TCP connection represents out-of-band communication with the singleton that can potentially compromise its consistency because the TCP messages (especially when there exist multiple TCP connections) are not totally ordered across the system. While we could opt to handle the TCP communication also in an interrupt-driven fashion, as we did with the GCS communication, the two sets of interrupts can become arbitrarily interleaved and make for tricky concurrent programming.

We chose the simpler and more controllable solution of using polling-based I/O for the TCP connections. Although this keeps the MEAD infrastructure in a "busy-wait" state, polling periodically for TCP-based I/O, this allows for the careful scheduling of when TCP communication can interleave with the GCS communication, through the appropriate adjustment of the polling frequency. Effectively, the file-descriptors associated with the TCP connections are inserted into a `select` system-call with a fixed timeout value; we note that this does change application semantics because a standard CORBA application (without replication or GCS involved) would use interrupt-driven I/O.

Because the GCS channel and the TCP sockets are managed separately by MEAD, the infrastructure underlying the singleton can differentiate between the two kinds of messages and handle the bridging seamlessly. Because the singleton must handle multiple TCP or GCS clients simultaneously, MEAD needs to perform concurrent duplicate detection-and-suppression across all of the various virtual end-to-end connections that pass through the singleton. For each incoming connection, MEAD maintains some book-keeping information, such as the last-seen request identifiers on that connection, in order to ensure state consistency.

3.3 Enhancements to Component Middleware and Deployment Engine.

Our fault tolerance enhancements span the ORB and component middleware. CIAO is an open-source[4] implementation of the OMG Lightweight CCM and Real-time CORBA [14] specifications built atop The ACE ORB (TAO) [15]. CIAO's architecture (shown in Figure 4) is designed based on (1) patterns for composing component-based middleware and (2) reflective middleware techniques to enable mechanisms within the component-based middleware to support different QoS aspects.

We also make use of the deployment facilities in the middleware. In Lightweight CCM, component assemblies are deployed and configured via the OMG D&C [16] specification, which manages the mapping of application components onto nodes in a target environment. The information about the component assemblies and

[4] CIAO is available from `www.dre.vanderbilt.edu/CIAO`.

the target environment in which the application components will be deployed are captured in the form of standard XML descriptors. To support automatic deployment and configuration of components based on their descriptors, we developed the *Deployment And Configuration Engine* (DAnCE) [17], whose architecture is shown in Figure 4.

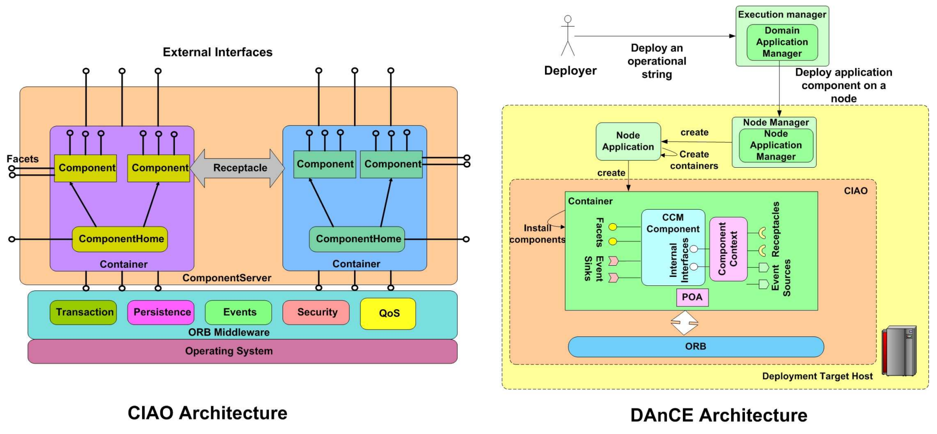

Fig. 4: CCM Implementation and DAnCE Architecture

Our middleware enhancements comprises the following:

Reconciling object reference semantics with GCS. In a GCS all members look alike to the outside world, which implies that a single object reference (IOR) should be available to clients. The concept of interoperable group reference (IOGR) does not work in GCS since the semantics of a group reference are not understood by GCS. This required our middleware to create exactly the same IOR for every replica even though they are different objects (or components). Moreover, when a new replica joins a group to maintain the quorum, we require it to have the same IOR exposed to the GCS. We modified the portable object adapter within TAO to use the `USER_ID` and `PERSISTENT_ID` POA policies. Moreover, the user id is chosen based on the unique instance name given to every replica and which corresponds to the group id used by the GCS for that replica group. We achieve this effect in a seamless manner without manual programmatic effort by delegating the job of configuring the policies on the objects (or components) using the DAnCE engine and supplying it with the right set of XML descriptors.

Reconciling procedural and OO models in state synchronization. The MEAD FT framework is a library that uses procedural programming while TAO/CIAO are OO frameworks dealing with objects/components. MEAD works only at the process level. This disparity required us to devise solutions that seamlessly interworked between MEAD and CIAO/TAO. We used DAnCE engine as the enabling technology. Within DAnCE, the NodeApplication is a process, which performs the job of an application or component server. We interfaced MEAD with the NodeApplication process and provided two global functions called `get_state` and `set_state`. MEAD cannot differentiate the state of individual components (or objects). As a result, MEAD invokes the global functions in the NodeApplication process passing it the group id of the replica group.

Within the CIAO/TAO framework, we implemented the state management functions that can be invoked on the components (objects). Rather than requiring the component developers to change their IDL descriptions of the components, which in turn will require recreating the stubs and glue code, we only require the component executor (or object servant) to implement the component-specific state management functionality.

Within the NodeApplication process the implementation of the state management global functions take care of coalescing the marshaled state from the replicas and passing it to MEAD during a get operation, and the other way round when setting the state. To do this, MEAD simply leverages the DAnCE's domain application manager, which in turn instructs all the NodeApplication processes to get/set the state during recovery.

3.4 Non-CORBA Fault tolerance: MySQL

Replicated data bases were provided by a nearly stock MySQL clustering service. In a stock MySQL clustering solution the emphasis is on data consistency. To this end if one of three cluster data bases finds itself alone it assumes that the other two data bases are still up and it has just lost connectivity to them. In order to maintain consistency in what it perceives to be a network partition it will shut itself down. This is obviously the wrong behavior if two of three pools have been destroyed the last DB should not shut itself down. Given that our fault-model assumes there will be no partitions we want the DBs to continue running no matter how many have been lost. We modified the cluster servers to not kill themselves when they detect a loss of a majority of the servers.

3.5 Putting it Together

Figure 5 shows the MLRM replicated across three pools of computational resources. The top-level elements are replicated across all three pools. This way even a large scale failure of a group of machines will not result in the failure of the MLRM functionality. Pool level MLRM elements do not need to be replicated in the same way, since there is by definition already instances in each pool. They are gateway elements though, since they have to interact with the replicated top-level elements and the node level elements.

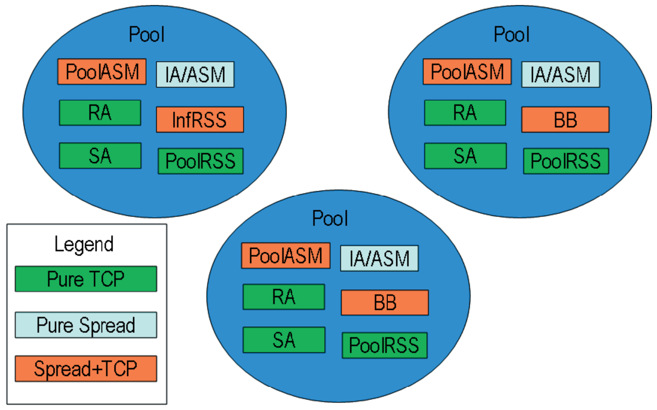

Fig. 5: Replicated MLRM

4 Experimental Results

The premise of our experiments is that there is a large scale compute platform running mission critical software. This platform is divided into clusters of computers (or data centers) that are geographically separated as shown in Figure 6. The failure scenarios we envision involve catastrophic loss of entire clusters at once. Our goal is to survive the failure by keeping the mission-critical functionality available and having extremely low down-time. We will present a brief overview of our mission-critical software. Then we will describe the experimental configuration and metrics used for evaluation. Finally, we will present our results.

The mission critical software comprises several applications, each with their own distinct characteristics. Certainly one of the most interesting aspects of these experiments is that several different fault-tolerance techniques were needed to achieve the overall level of fault-tolerance that our mission-critical software required. We used a combination of Active Replication, Warm-Passive Replication, and Database Replication (MySQL-Clustering). The first two techniques were implemented at the middleware layer. The last technique, Database Replication, was used “as is”, but had to be integrated with the actively replicated middleware elements.

4.1 Scenario A

For our first scenario, we considered a single cluster failure. The scenario called for a whole cluster to “disappear” (from the perspective of the mission-critical software’s user) instantaneously. The nature of the failure could be a major power failure or destruction of hosting facility. We simulated this failure by creating a network partition in such a way that packets sent to hosts in the failed cluster would be dropped at a router. We believe that this is an accurate simulation of

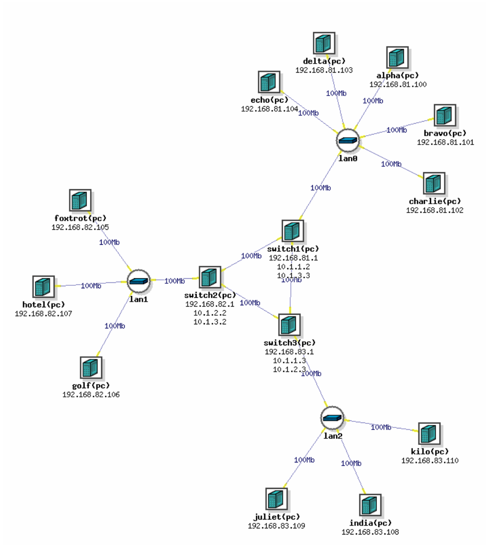

Fig. 6: Experiment Testbed and Setup

the failure, and has the advantage that we are able to determine the exact time the failure occurred (as opposed to trying to obtain a timestamp for physically pulling a plug).

Metrics We used two metrics to evaluate the results. The first metric was a boolean: *does the mission-critical functionality survive (i.e. is it still usable)*? The second metric related to the speed of recovery, with the goal being *to recover from the failure in under a second*. The recovery times for the actively replicated and passively replicated elements were derived from instrumentation inserted at the middleware layer. The recovery time for the Database replication was derived from an external program that made constant (as fast as possible) queries on the Databases. When the Database failover occurred, there is a slight increase in the time it takes to do the query, since the Database essentially blocks until it determines that one of the replicated instances is gone (and never coming back).

Results As illustrated in Figure 7, we were able to verify that the mission-critical functionality survived and was usable after the failure. While both the active replication and database replication parameters could potentially be tuned to produce even better results, the results from our experiments show that we are clearly *in the ballpark* with respect to our goal of less than 1 second recovery.

4.2 Scenario B

Our second scenario introduced cascading failures. The point of this scenario was to show that the fault-tolerant solution was robust. For this scenario we used

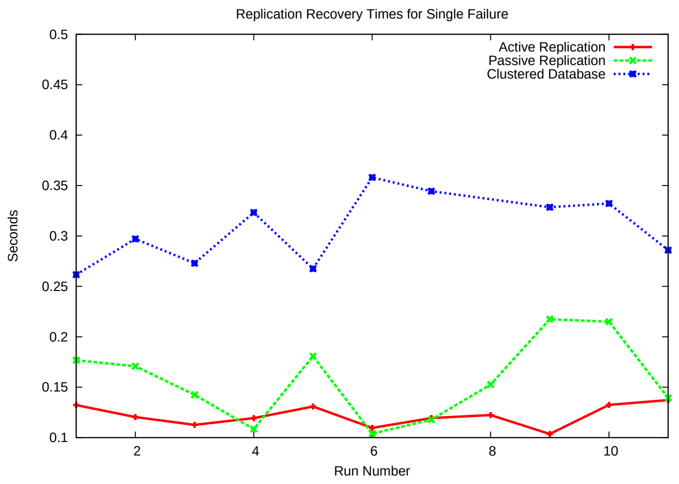

Fig. 7: Experimental Results for Scenario A

three clusters. We induced failure on one cluster, and then before the failover was complete, induced failure on another cluster.

Due to the fact that we could not inject accurately in every case we analyzed the runs after the fact and took only those runs where the failure had been injected at the right time.

Metrics The metrics used were the same metrics used in Scenario A. The only difference is that the "recovery time" is based on the time of the *first* induced failure.

Results As with Scenario A's results, we were able to verify that the mission-critical functionality survived. The recovery times are slightly higher than the single failure case. This is because we used knowledge of our fault-tolerance mechanisms to induce the second (cascading) fault at the worst possible time with respect to the recovery from the first failure.

Our results indicate that our FT solution enables fast and bounded recovery both in a single failure and cascaded failure scenarios.

5 Related Work

The Fault Tolerant CORBA standard [3] specifies reliability support through the replication of the CORBA servers, and the subsequent distribution of the replicas of every server across the nodes in the system. The idea is that, even if a replica (or a node hosting a replica) crashes, one of the other server replicas can continue to uphold the server's availability. While our mechanisms target support for both GCS and TCP clients, the consistency issues differ across the two kinds of clients

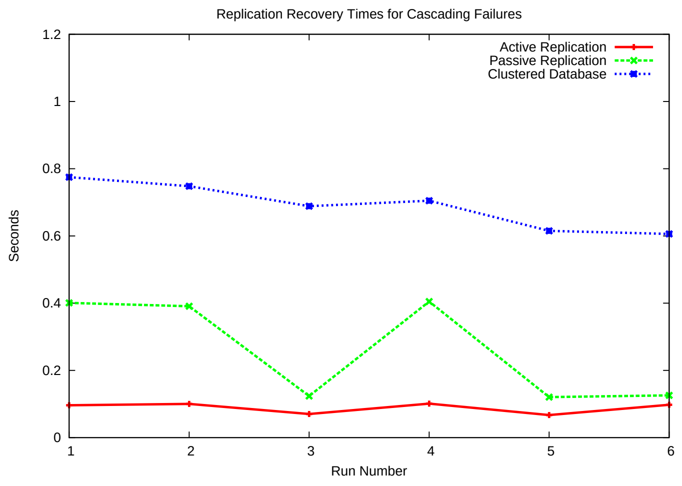

Fig. 8: Experimental Results for Scenario B

because the GCS clients can exploit totally ordered messaging guarantees while the TCP clients cannot, making them more complicated to handle.

The Fault Tolerant CORBA standard outlines a list of interfaces that constitute a fault-tolerance service and emphasizes that strong replica consistency should be provided in order to ensure effective fault-tolerance. Several fault-tolerant CORBA systems (Eternal, OGS, IRL, FTS, AQuA, DOORS, Orbix+Isis, Electra) [18] have emerged over the past decade. These systems have tended to require all of the objects of the CORBA application to be replicated and hosted over the fault-tolerant system. The Fault Tolerant CORBA standard, driven in part by the needs of vendors and real-world customers, incorporated the notion of fault-tolerance domains for scalability and ease of administration. Each fault-tolerance domain handled the replication of only its constituent objects, thereby making it possible to manage the large fault-tolerant applications by decomposing them into appropriate fault-tolerance domains.

The standard introduced the idea of a gateway to allow unreplicated clients running over non-fault-tolerant CORBA systems to invoke replicated objects that were located within fault-tolerance domains. However, the standard remains sketchy in its description of the gateway's implementation details, merely mentioning that the gateway's addressing information needs to be supplied to unreplicated clients outside the domain. Gateways [19, 20] and (predecessor) gateway-like client-side support [21] represent successful designs for supporting two-tier CORBA applications with an unreplicated client. However, to the best of our knowledge, this previous work does not include an in-depth critical look at the end-to-end consistency issues in mingling unreplicated/TCP and replicated/GCS semantics, and has also not included any empirical assessment of the attendant scalability and performance issues. The approach that we describe in this paper was born out of the lessons that we learned first-hand in implementing such a gateway [20] – thus, we aim to understand, articulate and resolve the underlying consistency challenges, as well as to quantify the actual performance overheads of the resulting system.

Our work on MEAD [11] goes beyond previous fault-tolerant CORBA systems in aiming for adaptive fault-tolerance (where the key replication properties of the system can be modified at runtime based on various system conditions), proactive fault-recovery (where pre-failure symptoms can be extracted from the system to provide advance notice of an impending failure, thereby paving the way for preemptive recovery action) and support for nondeterministic applications (where the application is analyzed for nondeterministic actions and appropriately handled, rather than forbidding programming practices such as multithreading).

GroupTransactions [22] aim to take advantage of both group communication and transactions through a new transactional model, where transactional servers can, in fact, be groups of processes. This allows for transactional applications to be built on top of computer clusters.

An e-Transaction [23] is one that executes exactly once despite failures, and is targeted at three-tier enterprise architectures with stateless middle-tier servers that are replicated. This overcomes the limitations of current transactional technologies that, for the most part, ensure at-most-once request processing, which is not sufficiently reliable. The e-Transaction abstraction builds upon an asynchronous replication scheme that provides both the liveness feature of replication, as well as the safety feature of transactions.

Another CORBA-related effort [24] aims to compare the two different kinds of systems — one with group communication and no transactions, and the other with transactions and no group communication — from the viewpoint of replicating objects for availability. Their study leads them to conclude that although transactions are effective in their own right, using group communication infrastructures to support transactional applications can lead to benefits, such as faster failover in the event of a fault.

While all of the above replication schemes refer to objects or servers, the notion of integrating group communication into a transactional model has been extended to the replication of the entire database itself [25]. This work attempts to eliminate the centralized and, therefore, unreliable approach that databases adopt today. The proposed family of replication protocols exploit group communication semantics to eliminate deadlocks, improve performance, and enhance reliability.

6 Conclusions

This paper has described an ambitious effort we undertook to make a realistic DRE system fault tolerant. We had two goals in mind when undertaking this endeavor. The first, obviously, was to make the DRE system fault tolerant. The second was to advance the state of the art in middleware support for fault tolerant DRE applications. This second goal intended to help us avoid building a one-of-a-kind point solution and to provide artifacts that would be reusable in future versions of this, and other, DRE systems.

The main conclusion we can draw from this work is that there is no one-size-fits-all fault tolerance solution. The tendency for real system builders to build hand-crafted solutions has come as much from necessity as from any immaturity of fault tolerance solutions. The development of a fault tolerance solution must start with the primary fault tolerance *requirement*, which in our case was near constant availability, or very rapid recovery of the MLRM infrastructure. These requirements lead one in the direction of a solution, in our case active replication. The characteristics of the application then start to come into play and affect the applicable techniques, for example

– Whether the application is accessible to modification
– Whether the application is nondeterministic and whether the nondeterminism is necessary or accidental
– How much state the application has and how often the state is changed
– Whether the application is always a client, always a server, sometimes a client and sometimes a server, or whether it is simultaneously a client and server
– Other characteristics, including whether it is object-based, component-based, or database-based.

Each of these, and many more factors, influence the fault tolerance techniques that are applicable and how they must be applied. The state of the art of fault tolerance technologies when we started this project was a number of available fault tolerance software bases, but which usually provided fault tolerance only to a limited subset of application types. For example, several solutions based on the Fault Tolerant CORBA specification provide fault tolerance for replicated servers in single-tiered, client-server, single language applications and were pervasive. Many of these were pervasive, forcing the entire system to be placed under control of a fault tolerance manager or on top of a group communication package. Other fault tolerance software, such as MySQL, provide replication of databases with strict transactional semantics, but no support for the fault tolerance of other elements besides the database. In contrast, few large software systems, including the DRE system we are working with, have such simple, restricted characteristics. The DRE system we describe in the paper is multi-language; with elements requiring different levels and types of fault tolerance (some needed constant availability, others less so); includes databases, clients, servers, and client/servers; and a combination of CORBA objects, CORBA components, and non-CORBA applications. This required us to make some significant advances in the state of the existing fault tolerance capabilities:

– Enabling the coexistence of multiple fault tolerance strategies (active and passive replication, as well as replicated DB elements) in the same system
– Enabling the coexistence of group communication and non group communication

- Supporting multiple ORBs within the same fault tolerant system
- Supporting peer-to-peer interactions, i.e., elements that could be clients or servers at any time and simultaneously
- Supporting multiple language applications, C++ and Java
- Supporting CORBA components
- Supporting multi-tiered applications

As much as possible, we have developed these additional capabilities to be independent of the application and to be part of the middleware fault tolerance infrastructure. Part of the significant challenge of the effort described in this paper has been the seamless integration of these new technologies to form the right fault tolerance solution for our DRE system. Our next step is to capture this integration and configuration process to form the basis for a *toolkit* approach to fault tolerance. The objective is to develop a fault tolerance toolkit, initially populated with the fault tolerance infrastructure pieces we have developed. The toolkit would support the development of fault tolerance solutions tailored to the needs of the application, constructed by integrating, configuring, and customizing the elements of the toolkit, without having to start from scratch and build one-of-a-kind solutions.

References

1. Sun Microsystems: Enterprise JavaBeans Specification. java.sun.com/products/ejb/docs.html (2001)
2. Object Management Group: CORBA Components. OMG Document formal/2002-06-65 edn. (2002)
3. Object Management Group: Fault Tolerant CORBA Specification. OMG Document orbos/99-12-08 edn. (1999)
4. Campbell, R., Daley, R., Dasarathy, B., Lardieri, P., Orner, B., Schantz, R., Coleburn, R., Welch, L.R., Work, P.: Toward an approach for specification of qos and resource information for dynamic resource management. In: Second RTAS Workshop on Model-Driven Embedded Systems (MoDES '04). (2004)
5. Balasubramanian, J., Lardieri, P., Schmidt, D.C., Thaker, G., Gokhale, A., Damiano, T.: A multi-layered resource management framework for dynamic resource management in enterprise dre systems. Journal of Systems and Software: special issue on Dynamic Resource Management in Distributed Real-time Systems (2006)
6. Wang, N., Schmidt, D.C., Gokhale, A., Rodrigues, C., Natarajan, B., Loyall, J.P., Schantz, R.E., Gill, C.D.: QoS-enabled Middleware. In Mahmoud, Q., ed.: Middleware for Communications. Wiley and Sons, New York (2003) 131–162
7. Object Management Group: Lightweight CCM RFP. realtime/02-11-27 edn. (2002)
8. Universite des Sciences et Technologies de Lille, F.: The OpenCCM Platform. `corbaweb.lifl.fr/OpenCCM/` (2003)
9. Schmidt, D.C., Natarajan, B., Gokhale, A., Wang, N., Gill, C.: TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. IEEE Distributed Systems Online **3**(2) (2002)
10. Brose, G.: JacORB: Implementation and Design of a Java ORB. In: Proc. DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems. (1997) 143–154

11. Priya Narasimhan and Tudor Dumitras and Aaron M. Paulos and Soila M. Pertet and Charlie F. Reverte and Joseph G. Slember and Deepti Srivastava: MEAD: support for Real-Time Fault-Tolerant CORBA. Concurrency - Practice and Experience **17**(12) (2005) 1527–1545
12. L. Lamport and R. Shostak and M. Pease: The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems **4**(3) (1982)
13. Yair Amir and Claudiu Danilov and Jonathan Robert Stanton: A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In: Proceedings of International Conference on Dependable Systems and Networks. (2000) 327–336
14. Object Management Group: Real-time CORBA Specification. OMG Document formal/02-08-02 edn. (2002)
15. Schmidt, D.C., Levine, D.L., Mungee, S.: The Design and Performance of Real-time Object Request Brokers. Computer Communications **21**(4) (1998) 294–324
16. Object Management Group: Deployment and Configuration Adopted Submission. OMG Document ptc/03-07-08 edn. (2003)
17. Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proceedings of the 3rd Working Conference on Component Deployment, Grenoble, France (2005)
18. Pascal Felber and Priya Narasimhan: Experiences, Approaches and Challenges in building Fault-tolerant CORBA Systems. Transactions of Computers **54**(5) (2004) 497–511
19. Pascal Felber: Lightweight fault tolerance in CORBA. In: In Proceedings of International Conference on Distributed Objects and Applications (DOA). (2001) 239–250
20. Priya Narasimhan and Louise E. Moser and P. M. Melliar-Smith: Gateways for Accessing Fault Tolerance Domains. In: Proceedings of ACM/Usenix/IFIP Middleware. (2000) 88–103
21. Alexey Vaysburd and Shalini Yajnik: Exactly-Once End-to-End Semantics in CORBA Invocations across Heterogeneous Fault-Tolerant ORBs. In: IEEE Symposium on Reliable Distributed Systems. (1999) 296–297
22. Patio-Martnez, M. and Jimnez-Peris, R. and Arvalo, S.: Group Transactions: An Integrated Approach to Transactions and Group Communication. In: Concurrency in Dependable Computing. Kluwer (2002)
23. Svend Frølund and Rachid Guerraoui: Implementing E-Transactions with Asynchronous Replication. IEEE Transactions on Parallel Distributed Systems **12**(2) (2001) 133–146
24. Little, M.C., Shrivastava, S.K.: Integrating group communication with transactions for implementing persistent replicated objects. In: Advances in Distributed Systems. (1999) 238–253
25. Bettina Kemme and Gustavo Alonso: A Suite of Database Replication Protocols based on Group Communication Primitives. In: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS). (1998) 156–163