Adding Fault-Tolerance to a Hierarchical DRE System

Paul Rubel, Joseph Loyall, Richard Schantz, Matthew Gillen

BBN Technologies
Cambridge, MA
{prubel, jloyall, schantz, mgillen}@bbn.com

Abstract. Dynamic resource management is a crucial part of the infrastructure for emerging mission-critical distributed real-time embedded system. Because of this, the resource manager must be fault-tolerant, with nearly continuous operation. This paper describes an ongoing effort to develop a fault-tolerant multi-layer dynamic resource management capability and the challenges we have encountered, including multi-tiered structure, rapid recovery, the characteristics of component middleware, and the co-existence of replicated and non-replicated elements. While some of these have been investigated before, this work exhibits all of these characteristics simultaneously, presenting a significant fault-tolerance research challenge.

1   Introduction

Fault-tolerance (FT) is an important characteristic of many systems, especially mission critical applications that are prevalent in medical, industrial, military, and telecommunications domains. Many of these applications are distributed real-time and embedded (DRE), combining the challenges of networked systems (e.g., distribution, dynamic environments, and nondeterminism) with the challenges of embedded systems (e.g., constrained resources and real-time requirements). For these systems, failure of applications or infrastructure can lead to catastrophic consequences.

As part of the DARPA ARMS program, and in conjunction with a team of researchers from several organizations, we have been developing a *Multi-Layered dynamic Resource Management* (MLRM) capability supporting a new Total Ship Computing (TSC) paradigm for the next generation of Naval surface ship [2]. This MLRM system controls the allocation of computing and communication resources to applications (some critical and others non-critical) and reallocation of resources when failures occur and when missions change, while maximizing operational capability.

Because MLRM is a critical part of the TSC infrastructure, it is important that it survive failures and damage. However, MLRM has some characteristics, typical of similar

DRE systems, that present challenges to making it fault-tolerant. In this paper, we describe our current efforts to make the MLRM fault-tolerant, concentrating on the following characteristics and challenges:

- *Hierarchical structure* – MLRM mirrors the hierarchical structure of the TSC infrastructure and must handle failures at each of the mission layer, resource pool and application layer, and resource layer.
- *Rapid recovery* – Because MLRM functionality is critical to keeping applications running and supporting ongoing missions, it is important that it be available continuously. Therefore, if MLRM fails it must recover as rapidly as possible, aiming for near zero recovery time.
- *Component middleware* – MLRM and TSC are being developed using emerging component middleware that offers many advantages, but exhibits different communication patterns than the traditional client-server model that many fault-tolerance techniques support.
- *Large numbers of elements with various degrees of fault-tolerance needs* – TSC and the MLRM subsystem itself are large distributed systems, with many interoperating elements, not all of which need to be fault-tolerant to the same degree. Traditional fault-tolerance solutions that require all elements to be part of a single approach fault-tolerance infrastructure are unsuitable.

We describe our current progress and findings in terms of each of these challenges and characteristics, and then describe our next steps toward achieving this work in progress.

2   Fitting Fault-Tolerance into a Layered DRE Structure

The MLRM architecture, illustrated in Figure 1, is hierarchical, with the following layers:

- The *Infrastructure Layer* deploys missions (consisting of application strings), assigns them to resource pools and security domains, and determines their relative priorities.
- The *Pool and Application String Layer* coordinates groups of related computing nodes (*pools*) and applications (*application strings*).
- The *Node* layer controls access to individual computing and communication resources.

The pool structure uses diversity in location and clustering to protect against large-scale damage or major system failures affecting a large portion of computing resources. With pools of computing hardware spread in different locations, the failure of one pool of resource still leaves sufficient computing capability for the critical operations.

To fit into this layered structure, we developed a top-down approach to fault-tolerance. We began developing fault-tolerance for MLRM to protect against the most catastrophic failures, so that the loss of a pool will not result in the loss of MLRM functionality. One of the functions of MLRM is to redeploy critical applications onto surviving nodes or pools in the face of a failure, but this is only possible if the MLRM functionality survives the failure. Therefore, we replicated the infrastructure layer MLRM elements across all the pools. If a pool fails, the infrastructure MLRM elements of the surviving pools take

over to initiate the actions necessary to deploy critical functionality across the remaining pools. In this case, there is no need to replicate the pool level MLRM elements, since they will still exist in the surviving pools.

3 Providing Rapid Recovery from Faults

Since MLRM has responsibility for recovering application functionality in the face of a node or pool failure, the infrastructure layer MLRM functionality must be constantly available. Therefore, the primary

Fig. 1. Our fault-tolerance capability mirrors the layered structure of the MLRM architecture.

requirement for our MLRM fault-tolerance is *speed of recovery*. Because of the very short recovery requirements, and since our fault-model is concerned with node loss rather than misbehavior, we employ a tolerance strategy that *actively* replicates components. In this scheme, each replica of the infrastructure MLRM is processing incoming messages and sending out responses. As long as one replica out of *n* of the MLRM has not failed, that replica will be able to carry out the responsibilities of the MLRM immediately and failures of *n-1* replicas can be tolerated.

We implemented active fault-tolerance for MLRM using MEAD [3] and Spread [1], both of which we customized, and in the case of MEAD, extended, and enhanced to support the features of the MLRM system. Spread provides a total order group communication system. We configured it for rapid recognition of the failure of group members. MEAD provides replication by intercepting CORBA calls and routing them through group communication, as well as code to suppress duplicate responses from replicas and recover from replica failures.

Figure 2 illustrates the results of experiments to evaluate the speed of our MLRM recovery. The experiments were conducted with three active replicas of MLRM infrastructure layer functionality[1] distributed over three pools. We failed one of the pools, by removing its network route, and measured the time for the remaining replicas, on hosts *alpha* and *hotel*, to recover from the failure.

The measured failover time includes the time needed for Spread to *detect* the failure of the group member. In all cases, the average detection+recovery time (hereafter called the *failover time*) was less than 200 ms. The mean failover time is 139 ms, with a standard deviation of 21 ms, to alpha and 128 ms, with a standard deviation of 6 ms, to hotel. The

---

[1] This first set of experiments only replicated the Infrastructure Allocator and Application String Manager-Global elements. We are in the process of replicating the Bandwidth Broker.
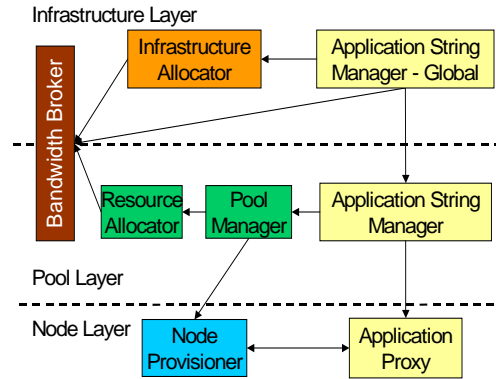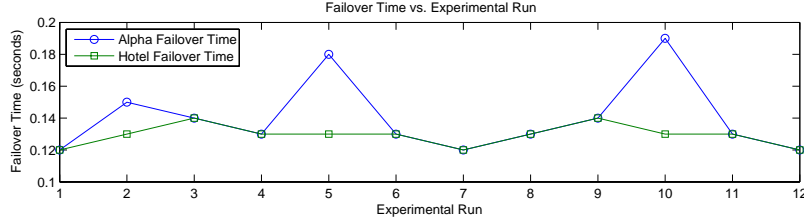
Fig. 2. Time to detect and recover from a failure of an MLRM replica

minimum failover time to both replicas is practically identical (119 and 118 ms, respectively), with a maximum of 185 ms for alpha and 135 ms for hotel.

4   Integrating Fault-Tolerance with the CORBA Component Model

Many fault-tolerance concepts, and existing code bases (including MEAD), were designed to work with replicated servers in client-server architectures, such as CORBA 2. MLRM has been developed using the CORBA Component Model (CCM, or CORBA 3), which has many advantages including lifecycle support and availability of design tools. However, there are two main challenges associated with providing fault-tolerance in CCM.

The first challenge is that the MLRM, and CCM in general, exhibits a peer-to-peer structure, where components can play the role of both client and server simultaneously. Our initial software base only supported replicated servers with duplicate suppression of responses from replicated servers back to non-replicated clients. We extended this code base to support the replication of both clients and servers, and by monitoring and controlling the CORBA message request identifiers we were able to provide the suppression of duplicate requests (from replicated clients) and responses (from replicated servers).

The second challenge is that the deployment architecture of CCM is more complicated than most CORBA 2 solutions. Before a component can be deployed using CIAO [4], an open-source C++ CCM implementation, a *Node Daemon (ND)* starts up a *Node Application (NA)*, which acts as a container for new components. The ND makes CORBA calls on the NA, instructing it to start components, which are not present at NA start up time. Note that the components, when instantiated in the NA, need to be replicated, but the NDs should not be.

To illustrate this point, consider an existing FT component when a new replica is started. Since MEAD ensures that all messages to and from one replica are seen at every replica, the existing replicas will receive an extra set of bootstrap interactions each time a new replica is started. This will not only confuse the existing replicas, but the responses from the new replica will also confuse the existing NDs. We developed a way to allow direct point-to-point interactions during the bootstrapping process and then switch to using reliable, ordered, group communications once the replicas have started.

The CCM envisions components interacting within a large-scale assembly. Architecturally, the current MLRM is made up of multiple assemblies. This decision was a pragmatic one since the ability to dynamically redeploy applications was not supported at the time by CIAO. It also allows us to set the unit of FT, the process, to the unit of CCM deployment, simplifying the process of making the MLRM FT.

Finally, since CORBA and the MLRM are multi-language solutions, our FT infrastructure needs to support multiple languages and their interactions. Currently both C++ via CIAO and Java via JacORB are supported and co-exist in our MLRM solution.

5 Limiting the Effects of the Fault-Tolerance Infrastructure

In order to keep replicas consistent, MEAD ensures that messages are reliably delivered to each replica in the same consistent order. To enforce these constraints we used Spread. Any interactions with a replica, after the initial CCM bootstrapping, pass through Spread.

In the simple case of a client interacting with a replicated server, all interactions are necessarily over Spread. The situation becomes more complex in the MLRM case, because we are (currently) replicating only the Infrastructure layer. Elements in the Pool layer will necessarily interact with the Infrastructure layer using Spread. However, since the Pool and Node layers are not replicated, they do not need the same consistency guarantees. Furthermore, from a usability perspective, we do not want to force the Spread infrastructure on the node layer, which can include hundreds of components.

The necessity of containing the use of Spread becomes even more apparent as the interactions within the system increase and more objects and components are introduced and connected. One example of this, introduced in Section 4, occurs when replicas are bootstrapped. To provide acceptable performance for components that require Spread and to more efficiently use resources, we implemented functionality that limits the use of Spread to where it is strictly necessary.

Spread is strictly necessary in replicas and each replica is required to use Spread for all its communications. Every entity that does not interact with a replicated entity does not need to use Spread. For those components that interact with both replicated and non-replicated entities, we ensure that they respond to a request in the same manner they received the request. If a request is received over TCP it is responded to over TCP and similarly for Spread. When initiating a request, MEAD compares the destination port against a list of ports on which Spread should be used. If the destination port is on this list the message will go out over Spread, otherwise it will use TCP as if MEAD were not present.

This same mechanism is used to deploy new replicas. Until a replica has been started the ND and NA interact to without group communication as neither is replicated. Once the replicated component starts, no more ND/NA interactions are necessary. In the future we envision a more dynamic method for distinguishing replicas from non-replicated objects or components that does not require up-front configuration.

6　Conclusions and Future Work

As systems become more complex and mission-critical, fault-tolerance continues to be an important part of their design and deployment. While developing a solution for providing a fault-tolerant MLRM, we continue to solve problems related to the structure of the MLRM, its fault-tolerance requirements, its underlying structure and framework, and its size and scope. An immediate next step is the rigorous evaluation of the viability, efficiency and operability of the current approach to very rapid failover for these types of DRE components. Moving forward we are working on solutions for replicating components that cannot be actively replicated, some of which must interact with network hardware on which running Spread is not an option. As CCM implementations mature, we hope to be able to better integrate fault-tolerance with components, particularly during deployment and when determining if the use of group-communication is required for particular requests. As we gain more insights into commonalities between the different implementations of components and objects, higher-level abstractions should become more apparent, providing further opportunities for improvements.

Acknowledgments

References

1. Yair Amir, Claudiu Danilov, Michal Miskin-Amir, John Schultz, and Jonathan Stanton. The Spread Toolkit: Architecture and Performance. Johns Hopkins University, Center for Networking and Distributed Systems (CNDS) Technical report CNDS-2004-1.
2. Roy Campbell, Rose Daley, B. Dasarathy, Patrick Lardieri, Brad Orner, Rick Schantz, Randy Coleburn, Lonnie R. Welch, and Paul Work. Toward an Approach for Specification of QoS and Resource Information for Dynamic Resource Management. Second RTAS Workshop on Model-Driven Embedded Systems (MoDES '04), Toronto, Canada, May 25-28, 2004.
3. P. Narasimhan, T. A. Dumitras, A. M. Paulos, S. M. Pertet, C. F. Reverte, J. G. Slember and D. Srivastava. MEAD: Support for Real-Time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, vol. 17, no. 12, 2005, pp. 1527-1545.
4. Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware, in Middleware for Communications, Qusay Mahmoud, Ed. Wiley and Sons, New York, 2003.