Reliable Distributed Real-time and Embedded Systems Through Safe Middleware Adaptation

Akshay Dabholkar, Abhishek Dubey, Aniruddha Gokhale,
Gabor Karsai, and Nagabhushan Mahadevan
Institute of Software Integrated Systems, Dept. of EECS
Vanderbilt University, Nashville, TN 37235, USA
Email: {aky,dabhishek,gokhale}@isis.vanderbilt.edu

***Abstract*—Distributed real-time and embedded (DRE) systems are a class of real-time systems formed through a composition of predominantly legacy, closed and statically scheduled real-time subsystems, which comprise over-provisioned resources to deal with worst-case failure scenarios. The formation of the system-of-systems leads to a new range of faults that manifest at different granularities for which no statically defined fault tolerance scheme applies. Thus, dynamic and adaptive fault tolerance mechanisms are needed which must execute within the available resources without compromising the safety and timeliness of existing real-time tasks in the individual subsystems. To address these requirements, this paper describes a middleware solution called Safe Middleware Adaptation for Real-Time Fault Tolerance (SafeMAT), which opportunistically leverages the available slack in the over-provisioned resources of individual subsystems. SafeMAT comprises three primary artifacts: (1) a flexible and configurable distributed, runtime resource monitoring framework that can pinpoint in real-time the available slack in the system that is used in making dynamic and adaptive fault tolerance decisions; (2) a safe and resource-aware dynamic failure adaptation algorithm that enables efficient recovery from different granularities of failures within the available slack in the execution schedule while ensuring real-time constraints are not violated and resources are not overloaded; and (3) a framework that empirically validates the correctness of the dynamic mechanisms and the safety of the DRE system. Experimental results evaluating SafeMAT on an avionics application indicates that SafeMAT incurs only 9-15% runtime failover and 2-6% processor utilization overheads at runtime thereby providing safe and predictable failure adaptability in real-time.**

***Index Terms*—Middleware, Adaptation, Fault Tolerance, Real-time, Software Health Management, Profiling**

I. Introduction

Applications that are deployed in safety-critical domains such as avionics, automotive, industrial automation are often over-provisioned in terms of the resources to handle failures in the worst-case scenarios. Moreover, they are closed in nature with precisely specified hard real-time Quality of Service (QoS) requirements of schedulability, timeliness, processor and memory allocation, and reliability. There is, however, an increasing trend towards realizing larger systems of systems that are composed predominantly from these deployed systems (*e.g.* ultra large-scale systems [1]), which we collectively term as distributed real-time and embedded (DRE) systems. The realization of DRE systems gives rise to various interdependencies between individual subsystems. Moreover, a new range of faults arise in the context of DRE systems, which must be handled to maintain the mission-critical nature of the overall DRE system in the context of the induced interdependencies.

The over-provisioning of resources in the individual subsystems is detrimental to realizing reliable DRE systems because fault tolerance solutions need resources but the individual subsystems do not have the flexibility to add new resources or modify the real-time schedules of their tasks. Redesigning and reimplementing the individual systems is not an option due to economic forces. Thus, designing fault-tolerance mechanisms for DRE systems must utilize available resources without compromising the real-time properties of the individual subsystems. Consequently, there is a need to identify unused resources at runtime that can be used for fault tolerance. The key insight we leverage for our work hinges on the existence of a *significant slack* in the over-provisioned individual subsystems. The challenge lies in identifying this slack and making effective use of it, which is the subject of this research.

Our next question is identifying the right fault tolerance mechanisms for DRE systems. Software Health Management (SHM) [2] is a technique which applies principles from system health management to software intensive systems that are ubiquitous in human society, including several safety-critical systems. Recently, these techniques have been applied to the system using a model-based approach [3]. SHM is a promising approach to providing fault-tolerance in real-time systems because it not only provides for fault detection and recovery but also effective means for fault diagnostics and reasoning, which can help make effective and predictable fault mitigation and recovery decisions. However, in these earlier works, the focus was on diagnosis of the faulty components and recovering the functionality of the system. Moreover, they handled only errors in a component that are known a priori with predefined failover strategies but did not account for system resource utilizations and availability. Thus, naively using SHM for DRE systems fault tolerance is likely to result in suboptimal runtime failure adaptations while also impacting

This work was supported in part by NSF CAREER Award CNS 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

resource utilizations.

Adaptive Fault Tolerance (AFT) [4] has been known to improve the overall reliability and resource utilizations. However, the technique has been applied only for soft real-time applications. Since they require additional resources to perform failure recovery, they can consume precious time from the hard real-time schedule of individual subsystems. To overcome the limitations in the existing SHM and AFT approaches yet being able to leverage their benefits and maintain the timeliness and safety of the real-time applications, this paper presents a middleware-based fault-tolerance solution called **Safe M**iddleware **A**daptation for Real-**T**ime Fault **T**olerance (SafeMAT). Our research on SafeMAT makes the following contributions:

- A **Distributed Resource Monitoring (DRM) framework** that provides highly configurable, fine-grained, distributed and hierarchical monitoring of system resources, such as processor, process, component and thread utilizations, that enables the selection of the best candidates to failover after failures. The DRM framework not only aids in the profiling and tuning of the system execution schedules but also provides a key component of the adaptive failure management to handle failures at runtime.
- An **Adaptive Failure Management (AFM) framework** that leverages the DRM framework to augment software health management mechanisms to provide an adaptive failure management capability. The AFM framework provides different cooperating runtime mechanisms, such as safe failure isolation and hierarchical failover algorithms, to enable the real-time applications to dynamically respond and adapt to system failures while ensuring that the system timeliness requirements are still adhered to.
- A **Performance Metrics Evaluation (PME) framework** that can profile the application execution by leveraging the DRM framework to determine the actual resource utilizations of various component tasks within their allocated scheduling quantum in the system execution period. This forms the basis for determining the existing slack in the system and leveraging it to safely provision and ensure predictability of the dynamic failure adaptation techniques provided by the AFT framework in SafeMAT. The PME framework also provides a tool to system integrators using which they can validate and tune the application component allocations and reliability to ensure safe provisioning of the necessary runtime failure adaptation mechanisms and additional new functionalities.

Paper Organization - The rest of the paper is organized as follows: Section II elicits the challenges for safe middleware adaptation; Section III presents the SafeMAT architecture and algorithms; Section IV empirically validates our approach in terms of minimal failover delays, and runtime overhead in the context of a representative DRE avionics case study; Section V compares our approach to related work in the area of real-time and fault tolerance; and finally Section VI provides concluding remarks identifying lessons learned and scope for improvements.

II. DRE Platform Assumptions and Research Challenges

This section brings out the challenges that motivate the need for the three primary vectors of the SafeMAT middleware presented in this paper. Before delving into the challenges, we present a model of the system and the underlying platform we consider in this paper.

A. Platform Assumptions

Our research focuses on a class of DRE systems where the system workloads and the number of tasks in the individual subsystems that make up the DRE system are known *a priori*. Examples of individual subsystems that make up DRE systems include tracking and sensing applications found in the avionics domain or the automobile system found in the automotive domain (*e.g.*, reacting to abnormalities sensed by tires). These systems demonstrate stringent constraints on the resources that are available to support the expected workloads and tasks. For this paper we focus on the CPU resource only.

In our research we assume that the individual subsystems of the DRE system use the ARINC-653 [5] model in their design and implementation because of its support for temporal and spatial isolation, which are key requirements for real-time systems. ARINC-653 uses fixed-priority preemptive scheduling where the platform is specified in terms of modules that are allocated per processor which in turn are composed of one or more partitions that are allocated as tasks. Each partition has its own dedicated memory space and time quantum to execute at the highest priority such that it gets preempted only when its allocated time quantum expires. Multiple components or subtasks can execute through multi-tasking within each quantum. For evaluating our design of SafeMAT and experimentation, we have leveraged an emulation [6] of the ARINC-653 specification.[1]

B. Research Challenges

Realizing the objectives of SafeMAT are fraught with a number of challenges described below. Resolving these challenges become the three primary vectors of our SafeMAT solution.

- **Challenge 1: Identifying the Opportunities for Slack in the DRE System**

As noted in Section I, DRE systems are composed often from individual legacy subsystems. Many of these subsystems comprise real-time tasks with strict deadlines on their execution times. To ensure the safety- and mission-criticality of these subsystems, they are configured with predefined execution schedules computed offline that are fixed for their execution lifetime once they are deployed in the field. This ahead-of-time system planning ensures that such subsystems will behave deterministically in terms of their expected behavior and their provided services, and the critical tasks with hard real-time

[1] We used the emulation environment since it was readily available to us, and has been used previously to demonstrate key ideas of software health management for avionics applications.

requirements will always satisfy their deadlines. To achieve this predictability, these subsystems are over-provisioned in terms of the allocated time and required capacity of resources. Naturally, for most of the time many of these resources remain under-utilized and hence provide an immediate opportunity to host the fault tolerance mechanisms needed for DRE systems. However, due to the dynamic nature of faults, the amount of slack available in each subsystem may vary at runtime thereby rendering any offline computation of slack for DRE fault tolerance useless. Therefore, there is a need to obtain a runtime snapshot of available slack in the system that then will enable the runtime execution of fault tolerance mechanisms for DRE systems. Such a monitoring capability must provide real-time information while at the same time not impose any significant overhead on the system. Section III-C presents our solution to a scalable Dynamic Resource Monitoring (DRM) capability in SafeMAT. In the context of our ARINC653-based scheduling of the DRE systems, DRM is not only able to obtain the actual CPU utilizations of the partition tasks but also of the subtasks (*i.e.*, application components) that are allocated within the partition.

- **Challenge 2: Designing Safe and Predictable Dynamic Failure Adaptation**
  Failures in DRE systems may manifest in different types and granularities. For example, some component failures may be logical or critical. The granularity of failures could be a component, group of components (subsystem), processes or processors. Moreover, the induced interdependencies in DRE systems due to composition of individual subsystems may lead to cascading failures of the dependent components (domino effect). Such an effect has the potential to increased deadline violations and over-utilization of system resources. Statically defined fault tolerance schemes will not work to completely handle these kinds of failures. Dynamic failure adaptation techniques can provide better capabilities to tolerate different kinds and granularities of failures, and can achieve better resource utilizations. However, given the criticality of hard real-time system execution, the failure adaptations that can be performed need to be safe and predictable. By utilizing the slack (which is obtained using the DRM capabilities), we can provision dynamic fault adaptation, however, we must ensure that the execution deadlines are not violated while achieving such runtime adaptations. Consequently, it is necessary to reduce the amount of recovery, which calls for failure detection and mitigation mechanisms that are fast and lightweight in terms of their space and runtime overhead as well are adaptive to the failure type and granularity, and component replica placements. Section III-D describes the adaptive fault tolerance mechanism supported by SafeMAT.
- **Challenge 3: Validating System Safety in the Context of DRE System Fault Tolerance**
  Although it may be feasible to design dynamic fault tolerance techniques for DRE systems by leveraging the slack, there is no easy approach to validate the safety and correctness of the resulting system and it is difficult to develop a mathematical proof of correctness of the system due to its dynamic nature. Thus, there is a need for a scalable and accurate capability that can validate the the overall DRE system for safety and predictability. SafeMAT provides a framework to profile a DRE system to validate if the real-time properties are met in the context of faults that can be artificially injected into the system. Section III-E describes such a framework that provides empirical validation of the system safety and predictability.

The rest of this paper presents our SafeMAT middleware that resolves these three challenges.

III. Design of SafeMAT

This section presents our SafeMAT solution to provide adaptive and dynamic fault tolerance to DRE systems. Since SafeMAT is designed to extend an existing emulation environment for ARINC-653, we first briefly describe the underlying system and the existing fault management approach. Subsequently, we describe our SafeMAT solution.

A. The ARINC-653 Component Model Middleware

The emulation middleware we use in our research is called the ARINC-653 Component Model (ACM) middleware, which essentially implements the CORBA Component Model [7] abstraction over the ARINC-653 emulation environment. ACM components interact with each other via well-defined patterns, facilitated by ports: asynchronous connections (event publishers & consumers) and/or synchronous provided/required interfaces (facets/receptacles). ACM allows the developers to group a number of ARINC-653 processes into a reusable component. Since this framework is geared towards hard real-time systems, it is required that each port be statically allocated to an ARINC-653 process whereas every method of a facet interface be allocated to a separate process.

ACM provides a design-time graphical modeling environment to enable a developer to assemble the components of the application, deploy them into ARINC-653 partitions (essentially OS processes) of ARINC-653 modules (essentially the processors), and configure various real-time properties of the components. A runtime middleware honors these decisions. The ACM middleware comprises multiple different functionalities. Of interest to us in this research is the *Module Manager (MM),* which is a controller responsible for providing *temporal partitioning* among partitions.[2] For this purpose, each module is bound to a single core of the host processor. Using offline analysis, the MM is configured with a fixed cyclic schedule computed from the specified partition periods and durations. It is specified as offsets from the start of the hyper period, duration and the partition to run in that window. Once configured and validated, the MM implements the schedule using the `SCHED_FIFO` policy of the Linux kernel and manages the execution and preemption of the partitions. The MM is also responsible for transferring the inter-partition messages across the configured channels. In case of a distributed system, there can be multiple MMs each bound to a processor core that are controlled hierarchically by a system-level module manager.

[2]Partitions are mapped to Linux processes.

*1) **Software Health Management in ACM**:* We have extended and augmented the ACM software health management framework [3] with resource-aware adaptive fault tolerance (AFT). ACM supports the notion of Software Health Management (SHM), which provides incremental fault mitigation strategies and operates at two levels. The first and basic level of protection is provided by component-level health management (CLHM), which is implemented in all components. It provides a localized timed state machine with state transitions triggered either by a local anomaly or by timeouts, and actions that perform the local mitigation. The second and global level is called system-level health management (SLHM). The SLHM comprises an aggregator of alarms that are received from individual CLHMs. The Aggregator feeds these alarms to a diagnostics engine, which is configured with a failure propagation graph to reason about the root cause of failures. The decisions are then fed to a fault mitigation capability called a *Deliberative Reasoner* [8].

*2) **Fault Handling in ACM and SafeMAT**:* An ACM component can be in one of the following three states: `active` (where all ports are operational), `inactive` (where none of the ports are operational) and `semi-active` (where only the consumer and receptacle ports are operational, while the publisher and facet ports are disabled). We focus on fail-stop failures within hard DRE systems that prevent clients from accessing the services provided by hosted applications. Failures can be masked by recovering and failing over to redundant backup replica components. Due to hard real-time constraints and to avoid state synchronization overhead, we use *semi-active replication* [9] to recover from fail-stop processor failures. In semi-active replication, one replica—called the primary—handles all client requests in active state. Backup replicas are in semi-active state where they process client's requests but do not produce any output.

We consider two main sources of failure for each component port (a) logical failure - internal software, concurrency (deadline violations due to lock timeouts) and environmental faults, and latent error in the developer code to implement the operation associated with the port or (b) a critical failure, such as process/processor failures, or undetected component failures. By convention, to recover from logical failures, we failover to similar backup replicas with identical interfaces but alternate implementations (from different vendors/developers). In case of critical failures, we failover to identical backup replicas or to alternate backup replicas if available. Also by convention, alternate backup replicas can be deployed within the same partition whereas identical backup replicas are always deployed to different partitions in the same module or different modules of ACM.

B. SafeMAT Architecture

We have designed the **Safe** **M**iddleware **A**daptation for Real-**T**ime Fault **T**olerance (SafeMAT) middleware to safely provision adaptive failure mitigation and recovery mechanisms in DRE systems that is resource-aware and leverages the benefits of software health management. The design of SafeMAT is driven by a holistic approach to answering the following three questions that emerge in fault tolerance for DRE systems:

*1) **How to be resource-aware?**:* To answer this question requires fine-grained information on the resource utilization in the system, which can then be used in the adaptive decisions to deal with faults. The Distributed Resource Monitoring (DRM) framework in SafeMAT described in Section III-C provides this capability.

*2) **How to deal with failures in the system of systems context by being aware of resources?**:* To answer this question requires a dynamic fault tolerance capability that can be adaptive to account for resource availabilities. The Adaptive Failure Management (AFM) framework in SafeMAT described in Section III-D provides this capability.

*3) **How to ensure that the solutions do not compromise the safety and timeliness of existing real-time systems?**:* To answer this question requires a capability to validate that the dynamic and adaptive fault tolerance mechanisms will not compromise on the safety and timeliness of the already deployed systems. The Performance Metrics Evaluation (PME) framework in SafeMAT described in Section III-E provides this capability.

Figure 1 illustrates the architectural components of SafeMAT and their interactions. It depicts the underlying ARINC-653 Component Middleware solution upon which SafeMAT is designed and implemented.

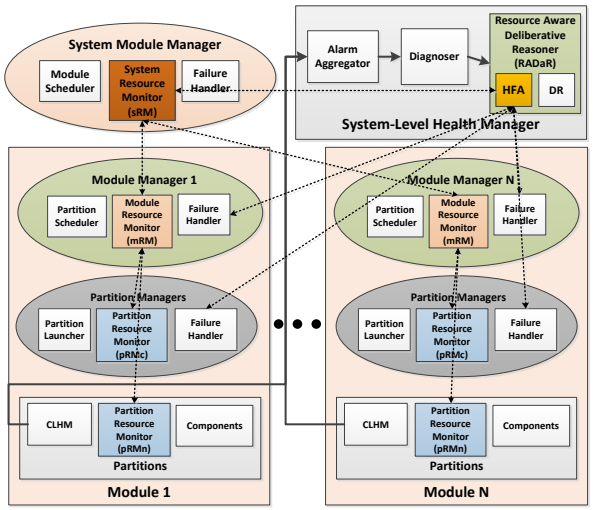

Fig. 1. SafeMAT Architecture

SafeMAT has been architected in the form a hierarchy of cooperating components implemented atop ACM. At the topmost level resides the System Module Manager along with the SLHM that hosts the System Resource Monitor (sRM) and the Resource-Aware Deliberative Reasoner (RADaR), respectively. At the second level are the different Module Managers that are deployed on each computing processor core or machine, each hosting a Module Resource Monitor (mRM). At

the third level are the different Partition Managers responsible for managing each partition, each hosting a Partition Resource Monitor (pRM). Each of the managers consequently have Failure Handlers to detect the failures in the partitions or modules and notifying them to the RADaR. The logical failures in components are notified by the respective CLHMs (from the ACM framework) residing in each application component. The different monitors form the DRM framework whereas the RADaR along with the various Failure Handlers form the AFM framework in SafeMAT. SafeMAT extends ACM by providing an additional level of lower-level fault mitigation in the form of a partition manager and its resource monitor (pRM). Doing so helps to isolate failures in partitions and mitigate partition faults by taking actions right away instead of involving the module manager.

C. Distributed Resource Monitoring

The Distributed Resource Monitoring (DRM) framework resolves Challenge 1 of Section II by providing a highly configurable and flexible distributed, hierarchical framework for monitoring the health and utilizations of system resources at various granularities, such as processor, process, component and thread. The framework comprises a distributed hierarchical network of a single System Resource Monitor (sRM) controlling multiple distributed Module Resource Monitors (mRM) that in turn control multiple Partition Resource Monitors (pRM) local to them in client-server configurations. The sRM resides in the system module whereas the mRMs are always deployed within the Module Managers and the pRMs are deployed within the individual partitions and their Partition Managers. The pRMs are of two types depending on their configured modes (a) *pMRc* in the COMPUTE mode and (b) *pMRn* in the NOTIFY mode.

1) ***Configurability in DRM****:* It is possible to configure the DRM framework using different strategies, depending on the overall system configuration and amount of system resources available. These strategies include `reactive` and `periodic` monitoring strategies that can be used in conjunction with different granularities of monitoring system resources ranging from processes to threads. The reactive monitoring strategy is the least resource consuming since the CPU utilizations are computed only when instructed by the RADaR (in case of a failure). The periodic monitoring strategy is the most resource consuming since the monitors compute utilizations periodically and keep the historic record of the utilizations to provide a better prediction regarding the utility of the resources. In the periodic strategy, the mRM periodically sends utilizations of all components to the sRM so that the information is readily available but may not be the most current one. The periodic strategy is also useful for profiling the resource utilizations during the profiling and tuning of the system execution characteristics in Section III-E. Finally, it is also possible to configure the DRM framework to supply only the utilizations of the specific entities that RADaR is interested in.

2) ***Discovering Resource Allocations****:* The DRM framework is also capable of discovering the runtime deployment and allocations of components to specific partitions and modules at runtime thereby obviating the need to configure the framework manually thereby enabling fast monitoring. It infers the assignments of the different subtasks to their components as well as allocations of components to their partitions when the monitors initialize their state. The pRMn runs within the partition in the NOTIFY mode where it does not compute the resource utilizations but only sends the mappings of the deployed components and their subtasks. These mappings are collated by the mRM and sent to the sRM which maintains the global allocations of subtasks to components, deployment of components to partitions, and the assignments of partitions to their modules. This capability enables the application of the DRM framework more generally to other types of systems where the allocations and deployments can change at runtime. Once the component deployment and allocations are learned by the sRM, it updates them with the primary-backup information about the components, component groups, and modules.

3) ***Resource Liveness Monitoring****:* The DRM framework has been additionally entrusted with monitoring the health of its own monitors by periodically making the monitors in the lower level send their health status to the upper level monitors. This monitoring capability is auxiliary to the existing signal handlers that also detect partition and partition manager failures thereby creating a more robust dual health monitoring capability. Thus, if the health status beacon is not received from the pRMn and pRMc by the Module Manager and Partition Manager then it is assumed that the Partition (process), and the Partition Manager (process) have crashed respectively. Similarly, it is assumed the module (processor/-core) has crashed if the mRM has not reported its health status beacon. Every time a failure is detected by the parent entity, the failure status is sent to the RADaR. Thus, the major advantage of SafeMAT over ACM is that while the SHM framework in ACM can only detect logical component failures, the DRM framework in SafeMAT can detect critical module, partition and component failures.

D. Resource-Aware Adaptive Failure Mitigation

To perform resource-aware failure adaptation and address Challenge 2 of Section II, we have developed the Adaptive Failure Mitigation (AFM) engine that leverages the DRM framework and augments the ACM-SHM framework through different cooperating runtime mechanisms, such as hierarchical failover and safe failure isolation. The AFM is designed as a collection of different components including the Failure Handlers and RADaR that integrate the *Hierarchical Failure Adaptation (HFA)* algorithm we developed with the Deliberative Reasoner (DR) [8] of the SLHM. The Failure Handlers are responsible for detecting process and processor failures and the simultaneous logical and critical component failures that have occurred but not reported to the HFA. The Failure Handlers along with the DRM framework and the HFA algorithm work together to provide quick and efficient failure adaptation at

runtime.

1) Failover Strategies: The type of failover strategy employed by the runtime failure adaptation mechanism is highly dependent on the failure type (*i.e.*, logical or critical), the failure granularity (*e.g.*, component, subsystem, partition or module), and the primary-backup deployment topology. The primaries can constitute individual components or groups of components (also called subsystems) and also the modules themselves. The graphical modeling capability provided by ACM can be used to model the deployment of primaries and backups. For logical failures, backups of application components with alternate implementations can be deployed in the same partition. However, for critical failures, backups with identical implementations are deployed in separate partitions within the same or remote modules (processors).

Due to the different primary-backup deployment possibilities, it is necessary to implement adaptive failover mechanisms that take into account the failure type, granularity and deployment topology that can enable the ability to failover and recover the application component(s) at the component, subsystem, process and processor levels. Moreover, to remain resource-aware, our algorithm chooses the best candidates at each level for failover by ranking the backups dynamically in increasing order of either their processor or partition or component utilizations for which we leverage the DRM framework.

2) Enabling Hierarchical Failure Adaptation (HFA): We have developed a Hierarchical Failure Adaptation (HFA) algorithm that adapts its failover targets depending upon the failure type, granularity and the primary-backup deployments. The algorithm is invoked whenever any of the DRM or the ACM-SHM frameworks detect a failure. In order to provide quick and efficient failover once the ACM Alarm Aggregator and the Failure Handlers detect a failed primary (component/partition/module), the sRM proactively pre-computes the sorted list of least utilized backups and the message is sent to the RADaR already containing the failed primaries piggybacked with the sorted list of failover target backups. The least utilized resource indicates maximum available slack. It then hands over the control to the SLHM.

It is the responsibility of the SLHM to determine as to when to activate the failure recovery mechanisms which is dependent upon the number of failures the system can withstand that have been programmed in advance within the ACM-SHM framework. It is also dependent upon the time taken by the system to stabilize till all alarms/errors are collected, which is usually a hyperperiod long in duration. Additionally, the AFM failure handlers and the DRM liveness monitoring is capable of detecting simultaneous module, partition, logical and critical component failures and are intelligently mitigated by the HFA algorithm in an hierarchical fashion.

3) The HFA Algorithm: At the core of the HFA algorithm (Figure 2) are three functions: `DetermineFailover`, `DRWrapper`, and `Restart`. DetermineFailover is a function that determines how best to choose a failover target component and rewire it with the rest of the application. On a failure, HFA first detects the failure type (module/partition/component group/critical/logical). If it's a module failure (M_F), the algorithm fails over to the least utilized identical module and calls REWIRE on all the components in that module. If it's a partition failure (P_F), the algorithm invokes the DRWrapper function for each component deployed in that partition. Otherwise a component failure (L_F/(C_F)) is assumed and the DRWrapper function is called for that component. DRWrapper then calls the DeliberativeReasoner function to determine group failure (G_F) *i.e.,* if the component has any dependent components that will also require failover and it selects the least utilized backup target group of components and finally calls DetermineFailover on each component in the failed group.

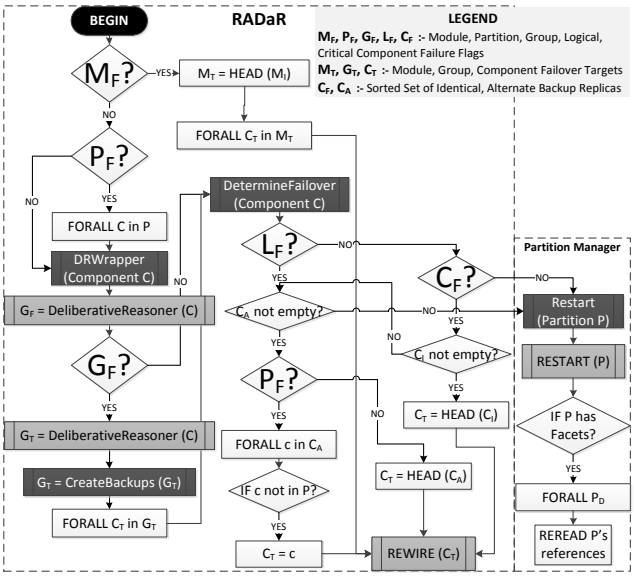

Fig. 2. The HFA Algorithm

In case of logical failure (L_F), DetermineFailover function checks if alternate backup replica is available. Otherwise, it checks for critical failure (C_F), and if true selects the least utilized identical backup replica if available. If not available, it checks if alternate backup replica is available. If not available, it restarts that partition to provide degraded QoS. If available, it checks for a simultaneous partition failure (P_F), in which case it selects the least utilized identical replica in a different partition. If not a critical or logical failure, it restarts the partition. DetermineFailover handles the simultaneous partition failure as a special case where it has occurred simultaneous with a logical component failure. In case of a simultaneous critical component failure, it does not need to handle this special case as identical backup replicas are always deployed on a different partition as primary. If the restarted partition contained facets, the Restart function ensures that the dependent partitions reread the restarted partition's new component references.

E. Pre-deployment Application Performance Evaluation

The real-time system execution schedule specifies the period of execution along with the allocated start and end times of

Algorithm 1 The Hierarchical Failover Adaptation (HFA) Algorithm

Input:
1: M, P, G, C : Module, Partition, Group, Component Failed Primaries
2: M_F,P_F, G_F, C_F, L_F : Flags for Module, Partition, Group, Critical and Logical component Failures
3: M_I, G_I, C_I, C_A : Sorted List of Identical & Alternate Backup Replicas
Output:
4: M_T, G_T, C_T : Failover Target Backup Replicas

Begin *HFA*
5: **if** M_F **then**
6: $M_T \leftarrow HEAD(M_I)$
7: **for all** $C_T \in M_T$ **do**
8: $REWIRE(C_T)$
9: **end for**
10: **else if** P_F **then**
11: **for all** $C \in P$ **do**
12: $DRWrapper(C)$
13: **end for**
14: **else**
15: $DRWrapper(C)$
16: **end if**
End

Begin *DRWrapper* (Component C)
1: $G_F \leftarrow DeliberativeReasoner(C)$
2: **if** G_F **then**
3: $G_T \leftarrow DeliberativeReasoner(C)$
4: $G_I \leftarrow CreateBackups(G_T)$
5: $G_T \leftarrow HEAD(G_I)$
6: **for all** $C_T \in G_T$ **do**
7: $DetermineFailover(C_T)$
8: **end for**
9: **else**
10: $DetermineFailover(C)$
11: **end if**
End

Begin *DetermineFailover* (Component C)
1: **if** L_F **then**
2: $CheckAlternate$ (C)
3: **else if** C_F **then**
4: **if** $C_I \neq \emptyset$ **then**
5: $C_T \leftarrow HEAD(C_I)$
6: **else**
7: $CheckAlternate$ (C)
8: **end if**
9: **else**
10: $Restart(P)$
11: **end if**
12: $REWIRE(C_T)$
End

Begin *CheckAlternate* (Component C)
1: **if** $C_A \neq \emptyset$ **then**
2: **if** P_F **then**
3: **for all** $c \in C_A$ **do**
4: **if** $c \ni P$ **then**
5: $C_T \leftarrow c$
6: **end if**
7: **end for**
8: **else**
9: $C_T \leftarrow HEAD(C_A)$
10: **end if**
11: **else**
12: $Restart(P)$
13: **end if**
End

Begin *Restart* (Partition P)
1: $RESTART(P)$
2: **if** P has provided interfaces **then**
3: **for all** $p \in P_d$ **do**
4: $REREAD(P$'s references)
5: **end for**
6: **end if**
End

the system tasks forming the scheduling quantum within the system execution time period (P). To address Challenge 3 of Section II, we have developed an application Performance Metrics Evaluation (PME) framework that can profile the application execution times and CPU utilizations by leveraging the DRM framework to measure the actual utilizations of various component tasks within their allocated scheduling quantum in the system execution period. The profiling of a system's resource utilization during execution, both in the presence and absence of failures, helps in determining post-failover processor utilization of the application and SafeMAT components. We measure the approximate worst case execution times (WCETs) of the SafeMAT adaptation mechanism to estimate the additional runtime overhead incurred. This can also help in safely predicting whether the application is capable of recovering within the hard real-time deadline. Moreover, the fine grained performance evaluation of the application component subtasks can also provide the basis for the system integrator for determining the slack in the system and thereby alter the task allocations within the application execution schedules to enable provisioning the necessary runtime adaptation mechanisms and additional new/upgraded functionalities.

F. SafeMAT Implementation

SafeMAT has been implemented atop the ACM real-time emulation middleware. It is implemented in around 5,000 lines of C/C++ source code excluding the ACM code. The *Partition Manager* is implemented as a separate process that gets spawned by the *Module Manager* for each partition

that needs to be spawned. The *Module Manager* sends the necessary partition information through environment variables and command line parameters to the *Partition Manager* which in turn spawns the partition with the right parameters and the same environment variables set.

The DRM uses the client-server paradigm and can be configured with two different monitoring strategies – `reactive` and `periodic`. The communication between the mRM and the pRMs is established through plain UDP sockets for performance. We did not employ TCP sockets as we assume the closed network that the avionics systems operate on have high reliability and high bandwidth performance with a small and bounded network propagation delay. The DRM computes processor, process and thread utilizations from the corresponding `/proc/stat`, `/proc/<PID>/stat` and `/proc/<PID>/task/<TID>/stat` Linux data structures.

To realize the PME framework, we profile SafeMAT component's longest execution time across various execution instances and use that as an approximation for the WCETs, and we measure the actual online CPU utilization percentages within each execution quantum of the hyperperiod by analyzing the timing logs generated by the Module Manager and the Partitions and the performance logs generated by the DRM framework, respectively over a large number of iterations. To achieve this we can configure the DRM to periodically collect the CPU utilizations only at the end of each hyperperiod. We compare these to the actual measured CPU utilization between those times to the duration of the quantum to get an idea of the slack that is available within each quantum.

IV. Empirical Evaluation of SafeMAT

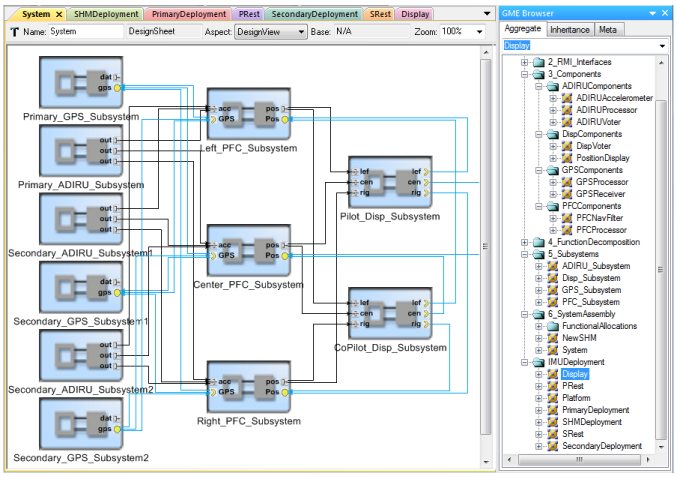

Fig. 3. IMU System Assembly.

To measure the performance of the various SafeMAT adaptive mechanisms, we used a representative DRE system called the Inertial Measuring Unit (IMU) [10] from the avionics domain. IMU is rich and large enough to provide a large number of components and redundancy possibilities that stem from the composition of its subsystems comprising the Global Positioning System (GPS), the Air Data Inertial Reference Unit (ADIRU) [11], the flight control (PFC) subsystem, and the Display subsystem. Figure 3 shows the IMU system assembly comprising primary subsystems of GPS and ADIRU, and their two secondary semi-actively replicated backup replica subsystems connected to redundant actively replicated PFC and Display subsystems. The ADIRU subsystem is designed to withstand 2 failures of its 6 Accelerometers. The GPS and the ADIRU subsystems feed the 3D location coordinates and acceleration values, respectively, to each of the PFC subsystems that integrate the acceleration values over the 3D coordinates computing the next coordinate position and outputting them to the Display subsystem. The GPS subsystems and ADIRU subsystems run at a frequency of 0.1 Hz and 1 Hz respectively. The PFC fetches the GPS data at a slower but accurate rate of 0.1 Hz whereas the Display subsystem fetches the data from the PFC subsystem at a rate of 1 Hz. Thus, the hyperperiod of the IMU is 10 seconds (LCM of 1 and 10).

A. Evaluating SafeMAT's Utilization Overhead

We use SafeMAT's PME framework to determine the overhead imposed by the SafeMAT's fast failure adaptation capability by measuring the CPU utilizations of its components. Measuring the actual utilizations at the end of each execution hyperperiod is an indicator of the slack available for accommodating failure adaptation mechanisms. Since SafeMAT builds over ACM, we executed 100 iterations of the IMU system each for the plain vanilla ACM-SHM and the SafeMAT adaptation failure recovery mechanisms. We artificially introduced failures at 15, 20, 30, 35 iterations in the GPS Processor, Accelerometers 6, 5 and 4, respectively such that the values output by them are exceedingly high (*i.e.* deviate from the expected trend). Once Accelerometer 4 fails at iteration 35, the system begins to malfunction and the Display starts receiving erroneously high acceleration values. At this moment the SafeMAT failure adaptation starts executing and makes the ADIRU and GPS primary subsystems failover to one of their semi-active secondary subsystems depending upon their overall least average utilizations. In this execution scenario the `Primary_ADIRU_Subsystem` fails over to the `Secondary_ADIRU_Subsystem2` whereas the `Primary_GPS_Subsystem` fails over to the `Secondary_GPS_Subsystem1`. Figure 4 shows that the SafeMAT does not add significant utilization overhead (2-6%) over the existing ACM-SHM imposed utilizations (26-73.26%).

B. Evaluating SafeMAT-induced Failover Overhead Times

To qualitatively measure SafeMAT's runtime failover overhead times we measure the worst-case execution times (WCETs) of the SafeMAT's components based on two main parameters: (1) the impact of component replica placements relative to their primaries and (2) the number of nested components within the component group that need failover.

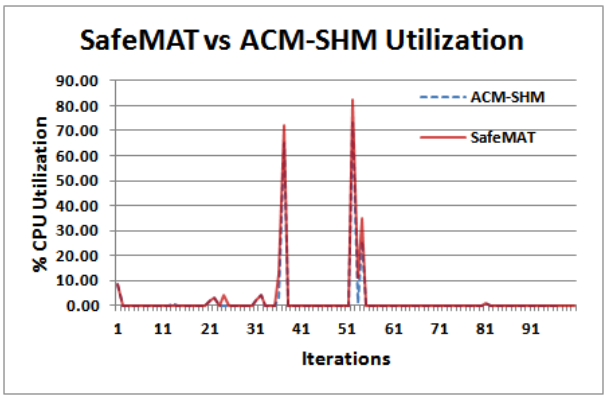

Fig. 4. SafeMAT Utilization Overhead

We measure the failover overhead (T_{FO}) as:

$$T_{FO} = T_{Diag} + T_{DR} + \sum_{i=1}^{m}\left(T_{mRM} + \sum_{j=1}^{p} T_{pRM}\right) + T_{sRM} + T_{HFA}$$

where

m - number of modules
p - number of partitions within each module
T_{Diag} - WCET for Failure Diagnosis
T_{DR} - WCET for Deliberative Reasoning
T_{sRM} - WCET for the sRM to collect utilizations
T_{mRM} - WCET for each mRM to collect utilizations
T_{pRM} - WCET for each pRM to collect utilizations
T_{HFA} - WCET for Hierarchical Failover Algorithm

1) Impact of Component Replica Deployments: To measure the impact of component replica deployments, we focused on the GPS subsystem from the IMU case study.

We created different deployment scenarios by altering the placements of the component replica by either placing them either within the same partition as primary, or a different partition in the same module or a different partition within a different module. We executed the GPS subsystem with the existing vanilla ACM-SHM recovery mechanisms in place and with the new SafeMAT failure adaptations enabled. We have considered the WCETs of both ACM-SHM and SafeMAT in this case. As shown in Table 5, SafeMAT incurs comparable execution times to the existing ACM-SHM execution times as this scenario has been evaluated on a per component basis. The times go up as the replica partitions move further away from the primaries. The high recovery overhead per component are due mainly to the unavoidable network latency to collect the utilizations. However, the minuscule overhead on the order of a few milliseconds are very insignificant in this case and will not cause deadline violations when there is a large amount of slack available, which is usually the case. Therefore, this is not a cause of concern as shown in the next evaluation where we progressively increase the number of components that need failover – a scenario that is more common in real systems.

2) Impact of Component Group Size: To measure the impact of size of the group of components that require failover, we measure the overhead incurred by SafeMAT for the GPS and ADIRU subsystems where the number of components increase from just 2 to 13. As shown in the evaluation Table 6,

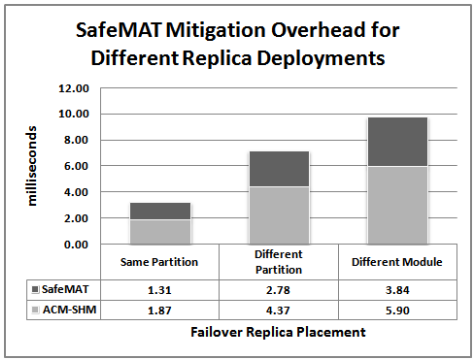

Fig. 5. SafeMAT Mitigation Overhead for Different Replica Deployments

when the number of components increase, the SafeMAT overhead costs gets amortized over larger number of components. The effective additional runtime overhead incurred by SafeMAT's adaptive mechanisms becomes significantly less (9-15%) compared to the ACM-SHM's diagnostic and reasoning overhead. SafeMAT's overhead is largely dependent on the size of the recovery group, deployment complexity of the components within the recovery group, and the amount of network communication required within the DRM as shown in the T_{FO} equation. However, it does not grow exponentially, as recovery group size increases. The more the number of components that need failover, the more the amount of utilization data that can be bundled together in the network messages that are sent by the DRM monitors to RADaR. Conversely, the smaller the number of components affected, the greater the overhead incurred by SafeMAT due to the network communication that is mandatory even for relatively small number of messages exchanged.

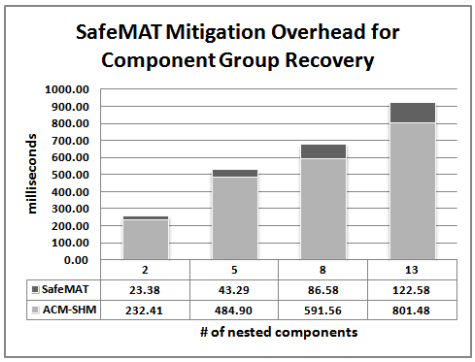

Fig. 6. SafeMAT Mitigation Overhead for Component Group Recovery

C. Discussion: System Safety and Predictability

Compared to the vanilla ACM-SHM mechanisms, SafeMAT adds negligible runtime utilization overhead without overloading the system while performing better failure recovery within the available utilization slack. Moreover, by selecting the least-utilized failover targets, SafeMAT maintains more available post recovery slack within the system compared to ACM-SHM, while potentially improving the task response times as well. Figure 7 shows that there was no noticeable impact on the Display jitter values using SafeMAT over vanilla ACM-

SHM and therefore the response times remained largely unaffected while at the same time failure recovery was superior. Moreover, there were no missed real-time deadlines for the application tasks. Moreover, SafeMAT adds negligible runtime failover overhead thereby maintaining the predictability of the overall system Thus, these results illustrate that SafeMAT maintains the safety of the system and also the predictability.

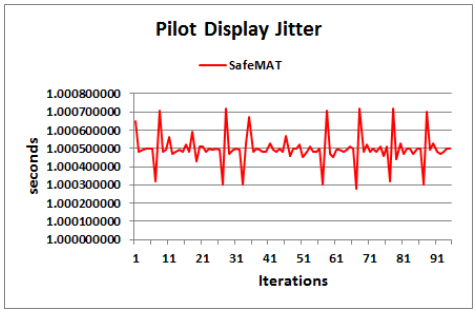

Fig. 7. Application Display Jitter (Hyperperiod = 1 sec)

V. Related Work

In this section we discuss the existing body of research in the area of adaptive fault tolerance in distributed real-time and embedded systems and compare and relate our work on SafeMAT. We categorize adaptive fault tolerance research in following areas:

Dynamic Scheduling: Common methodologies to leverage the slack in execution schedule have focused on dynamic scheduling depending upon the runtime conditions. The Realize middleware [12] provides dynamic scheduling techniques that observes the execution times, slack, and resource requirements of applications to dynamically schedule tasks that are recovering from failure, and make sure that non-faulty tasks do not get affected by the recovering tasks.

Resource-aware Adaptations: The DARX framework [13] provides fault-tolerance for multi-agent software platforms by focusing on dynamic adaptations of replication schemes as well as replication degree in response to changing resource availabilities and application performance. [14] proposes adaptive fault tolerance mechanisms to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. Research performed in AQUA [15] dynamically adapts the number of replicas receiving a client request in an ACTIVE replication scheme so that slower replicas do not affect the response times received by clients. Eternal [16] dynamically changes the locations of active replicas by migrating soft real-time objects from heavily loaded processors to lightly loaded processors, thereby providing better response times for clients. FLARe [4] proactively adjusts failover targets at runtime in response to system load fluctuations and resource availability. It also performs automated overload management by proactively redirecting clients from overloaded processors to maintain the desired processor utilization at runtime. [17] focuses on an adaptive dependability approach by mediating interactions between middleware and applications to resolve constraint consistencies while improving availability of distributed systems.

Real-time fault-tolerant systems: IFLOW [18] and MEAD [19] use fault-prediction techniques to reduce fault detection and client failover time to change the frequency of backup replica state synchronization to minimize state synchronization during failure recovery, and by determining the possibility of a primary replica failure and redirecting clients to alternate servers before failures occur, respectively. The Time-triggered Message-triggered Objects (TMO) project [20] considers replication schemes such as the primary-shadow TMO replication (PSTR) scheme, for which recovery time bounds can be quantitatively established, and real-time fault tolerance guarantees can be provided to applications. FC-ORB [21] is a real-time Object Request Broker (ORB) middleware that employs end-to-end utilization control to handle fluctuations in application workload and system resources by enforcing desired CPU utilization bounds on multiple processors by adapting the rates of end-to-end tasks within user-specified ranges. Delta-4/XPA [22] provided real-time fault-tolerant solutions to distributed systems by using the semi-active replication model. Other research [23] uses simulation models to analyze multiple checkpointing intervals and their effects on fault recovery in fault-tolerant distributed systems.

Need for Safe Fault Tolerance: For the hard real-time DRE systems, applying dynamic load balancing, dynamic rate and scheduling adjustments, adaptive replication and redundancy schemes add extraneous dynamism and therefore potential unpredictability to the system behavior. Altering the redundancy strategies require altering the real-time schedules which is not acceptable for hard real-time systems that are strictly specified. Constantly redirecting clients upon overload and promoting backups to primaries adds unnecessary resource consumptions for fixed priority systems. Such approaches do not attempt to minimize the number of resources used; their goal is to maintain service availability and desired response times for the given number of resources in passively replicated systems. However, in hard real-time systems exceeding the RMS bound of 70% of the processor utilization is not a concern as the tasks are guaranteed to not be preempted until their allocated quantum is over. So as long as task utilizations are guaranteed to be under 100% processor load, their deadlines and profiled WCETs are guaranteed to be satisfied. In SafeMAT we guarantee through exhaustive application performance profiling by establishing runtime utilization and failover overhead bounds that the dynamic failure adaptations will not violate the real-time deadlines and overload the resources. Moreover, as the system resources are over-provisioned we use semi-active replication which subsumes the need for expensive state-synchronization and load balancing mechanisms.

VI. Conclusion

Mission-critical hard real-time applications being in-service for many years, have too rigid execution schedules to incorporate additional evolving domain requirements in the form

of new functionalities and better failure adaptation techniques even if their resources are over-provisioned to ensure their safety and predictability. While, existing SHM techniques are predictable, they are too static and do not offer the best case failure adaptation in real-time. In order to evolve these systems and improve their predictability, reliability and resource utilizations, it is necessary to discover the existing slack within their execution schedules and utilize it to safely provision additional and efficient dynamic failure adaptation mechanisms.

In this paper, we presented a dynamic, safe middleware adaptation technique and a performance metric evaluation framework that provided a fast and adaptive failover through flexible and configurable fine-grained resource monitoring and an hierarchical failure adaptation algorithm that is not only resource-aware but also took into account the failure type, failure granularity, the relative component replica placements. Our approach manifested in the form of the SafeMAT middleware and the PME framework. We also rigorously evaluated our adaptive middleware by measuring the runtime utilization and the execution overhead for different replica deployments as well as an increasing number of components.

REFERENCES

[1] S. E. Institute, "Ultra-Large-Scale Systems: Software Challenge of the Future," Carnegie Mellon University, Pittsburgh, PA, USA, Tech. Rep., June 2006.

[2] A. Srivastava and J. Schumann, "The case for software health management," in *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 3–9.

[3] A. Dubey, G. Karsai, and N. Mahadevan, "Model-based software health management for real-time systems," in *Aerospace Conference, 2011 IEEE*, march 2011, to appear. Draft available at http://isis.vanderbilt.edu/sites/default/files/PaperSubmission.pdf.

[4] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, and D. C. Schmidt, "Adaptive Failover for Real-time Middleware with Passive Replication," in *Proceedings of the 15th Real-time and Embedded Applications Symposium (RTAS '09)*, San Francisco, CA, Apr. 2009, pp. 118–127.

[5] ARINC, "ARINC specification 653-2: Avionics application software standard interface part 1 - required services," ARINC Incorporated, Annapolis, Maryland, USA, Tech. Rep., May 2010.

[6] A. Dubey, G. Karsai, and N. Mahadevan, "A component model for hard real-time systems: CCM with ARINC-653," *Software: Practice and Experience*, vol. 41, no. 12, pp. 1517–1550, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1083

[7] *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 ed., Object Management Group, Jan. 2008.

[8] A. Dubey, N. Mahadevan, and G. Karsai, "A deliberative reasoner for model-based software health management," in *The Eighth International Conference on Autonomic and Autonomous Systems*, 2012, to appear.

[9] A. M. Déplanche, P. Y. Théaudière, and Y. Trinquet, "Implementing a semi-active replication strategy in chorus/classix, a distributed real-time executive," in *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 90.

[10] A. Dubey, N. Mahadevan, and G. Karsai, "The inertial measurement unit example: A software health management case study," Institute for Software Integrated Systems, Vanderbilt University, Tech. Rep., 02/2012 2012.

[11] M. McIntyre and C. Gossett, "The boeing 777 fault tolerant air data and inertial reference system-a new venture in working together," in *Digital Avionics Systems Conference, 1995., 14th DASC*, Nov. 1995, pp. 178 –183.

[12] V. Kalogeraki, P. M. Melliar-Smith, and L. E. Moser, "Dynamic Scheduling of Distributed Method Invocations," in *21st IEEE Real-time Systems Symposium*. Orlando, FL: IEEE, Nov. 2000.

[13] O. Marin, M. Bertier, and P. Sens, "Darx: A framework for the fault-tolerant support of agent software," in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2003, p. 406.

[14] O. Gonzalez, H. Shrikumar, J. A. Stankovic, and K. Ramamritham, "Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling," in *RTSS '97*, San Francisco, CA, USA, 1997, p. 79.

[15] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "An Adaptive Quality of Service Aware Middleware for Replicated Services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1112–1125, 2003.

[16] V. Kalogeraki, P. M. Melliar-Smith, L. E. Moser, and Y. Drougas, "Resource Management Using Multiple Feedback Loops in Soft Real-time Distributed Systems," *Journal of Systems and Software*, 2007.

[17] L. Froihofer, K. M. Goeschka, and J. Osrael, "Middleware support for adaptive dependability," in *Middleware*, 2007, pp. 308–327.

[18] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, and R. E. Strom, "Utility-Driven Proactive Management of Availability in Enterprise-Scale Information Flows." in *Proceedings of ACM/Usenix/IFIP Middleware*, 2006, pp. 382–403.

[19] S. Pertet and P. Narasimhan, "Proactive recovery in distributed corba applications," in *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, p. 357.

[20] K. H. K. Kim and C. Subbaraman, "The pstr/sns scheme for real-time fault tolerance via active object replication and network surveillance," *IEEE Trans. on Know. and Data Engg.*, vol. 12, no. 2, 2000.

[21] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos, "FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization controlstar, open," *Journal of Systems and Software*, vol. 80, no. 7, pp. 938–950, 2007.

[22] D. Powell, "Distributed Fault Tolerance: Lessons from Delta-4," *IEEE Micro*, vol. 14, no. 1, pp. 36–47, 1994.

[23] P. Katsaros and C. Lazos, "Optimal object state transfer - recovery policies for fault tolerant distributed systems," in *Proc. of DSN. (2004)*.