SafeMAT: Safe Middleware-based Adaptation for Predictable Fault-Tolerant Distributed Real-time and Embedded Systems

Akshay Dabholkar[1 a], Aniruddha Gokhale[a,*], Abhishek Dubey[a], Gabor Karsai[a], Nagabhushan Mahadevan[a]

[a] *Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37235, USA*

Abstract

In distributed real-time and embedded (DRE) systems, the composition of 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 be able to execute within the resource constraints of DRE systems without compromising the safety and timeliness of existing real-time tasks in the individual subsystems. Software Health Management (SHM) is a promising technique for DRE systems to recover from failures predictively, however, since current SHM approaches can deal with component failures known at design-time only and do not consider resource utilizations and availability, their use in DRE systems result in sub-optimal failure adaptations. On the other hand, Adaptive Fault Tolerance (AFT) approaches are known to provide dynamic fault tolerance, however, they require additional resources, which may not be available in resource-constrained DRE systems. To realize the benefits of SHM and AFT in a way that is resource-aware and preserves the real-timeliness of existing applications, this paper describes a middleware solution called **Safe** **M**iddleware **A**daptation for Real-**T**ime Fault **T**olerance (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 to execute the 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 can be used for the dual purposes of identifying the slack in the system and also for empirically validating 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 thereby providing safe and predictable failure adaptability in real-time.

Keywords: Middleware, Adaptation, Fault Tolerance, Real-time, Software Health Management, Profiling

1. Introduction

Applications found in domains such as avionics, automotive, and industrial automation are safety-critical, which calls for over-provisioning of resources to handle worst-case scenarios, such as failures. Most often these systems are closed in nature, *i.e.*, task and workloads are fixed, with precisely specified hard real-time quality of service (QoS) requirements for which the algorithms and implementations for system schedulability, resource allocation, and system reliability undergo rigorous validation and verification. Recent trends, however, indicate that these individual safety-critical systems are increasingly being interconnected to become part of larger systems. For example, intelligent transportation systems are enabling automobiles to be connected to each other as well as to a variety of infrastructure elements. In this paper we refer to these systems of systems as distributed real-time and embedded (DRE) systems. The realization of DRE systems gives rise to various interdependencies between individual subsystems. Moreover, the degree of uncertainty in the overall system increases due to the induced interdependencies between the individual subsystems. A key class of such uncertainties includes a whole new set of faults that were not considered earlier in the design of individual subsystems of the DRE system, which must be now be handled to maintain the mission-critical nature.

Unfortunately, the over-provisioning of resources in the individual subsystems, which was a necessity to handle worst-case scenarios, becomes detrimental to realizing reliable DRE systems because fault tolerance solutions to handle the new class of faults will need additional resources. However, 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 deployed individual systems is not an option. Thus, the only feasible way to designing fault-tolerance mechanisms for DRE systems is to opportunistically utilize available resources without compromising the real-time properties of the individual subsystems. Opportunistic use of resources is inherently a runtime propery, which implies the need to identify unused resources at runtime that can be used for DRE system fault tolerance. The key insight we leverage in our work hinges on the existence of a *significant slack*, which we hypothesize will exist to varying degrees due to the over-provisioned nature of individual subsystems. The key challenge lies in identifying this slack at runtime in a timely manner and making effective use of it.

The next we face is identifying the right fault tolerance mechanisms for DRE systems that can execute effectively when resources are used opportunistically. Software Health Management (SHM) [1] 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 as shown in [2]. Recently, these techniques have been applied using a model-based approach [2] and a follow on work that included a goal-based deliberative search reasoner was presented in [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.

*Corresponding Author

Email addresses: aky@dre.vanderbilt.edu (Akshay Dabholkar), a.gokhale@vanderbilt.edu (Aniruddha Gokhale), dabhishe@isis.vanderbilt.edu (Abhishek Dubey), gabor.karsai@vanderbilt.edu (Gabor Karsai), nag@isis.vanderbilt.edu (Nagabhushan Mahadevan)

However, there exist some limitations to existing work in SHM in directly applying these techniques to DRE systems. First, the earlier works have focused primarily on diagnosis of the faulty components and recovering the functionality of the system. Second, they handled only those errors in a component that were known *a priori* for which predefined failover strategies were designed. Third, these techniques were not concerned with system resource utilizations and resource availability during failover. Thus, despite the promise, naively using SHM for DRE systems fault tolerance is likely to result in suboptimal runtime failure adaptations while also impacting resource utilizations.

Adaptive Fault Tolerance (AFT) [4] is known to improve the overall reliability and resource utilizations. However, the technique has been applied for systems with soft real-time requirements. Moreover, since these techniques often require additional resources to perform failure recovery, if applied naively to DRE systems, they can consume precious time and resources from the hard real-time schedules of individual subsystems. Yet, opportunistic use of resources call for adaptive mechanisms in DRE systems fault tolerance. To overcome the limitations highlighted above when considering SHM and AFT in isolation while still availing of their benefits and maintaining 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 2 elicits the challenges for safe middleware-based adaptation; Section 3 presents the SafeMAT architecture and algorithms; Section 4 describes details on SafeMAT implementation; Section 5 empirically validates our approach in terms of minimal failover delays, and runtime overhead in the context of a representative DRE avionics case study; Section 6 compares our approach to related work in the area of real-time and fault tolerance; and

finally Section 7 provides concluding remarks identifying lessons learned and scope for improvements.

2. DRE System Model and Research Challenges

This section brings out the challenges that motivate the need for the primary vectors of the SafeMAT middleware presented in this paper. Before delving into the challenges, we present a model of the system and details of the underlying platform upon which we build our solution for this research.

2.1. System Model and 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, the automobile system found in the automotive domain (*e.g.*, reacting to abnormalities sensed by tires), conveyors systems in industrial automation (*e.g.*, periodic monitoring and relaying of health of physical devices to operator consoles), or resource management in the software infrastructure for shipboard computing domain. These systems impose 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.

Our research assumes 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 described next.[3]

2.2. Research Challenges

Section 1 highlighted the need for resource-aware and safe adaptive fault tolerance for DRE systems that also incorporated principles of software health management. Realizing these objectives is fraught with a number of challenges, which are presented below. The three primary vectors of our SafeMAT solution stem from the need to resolve these challenges.

- **Challenge 1: Identifying the Opportunities for Slack in the DRE System** - As noted in Section 1, 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

[3] 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.

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 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 3.4 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 3.5 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 3.6 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 challenges.

3. Design and Implementation of SafeMAT

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 in a way that is resource-aware and leverages the benefits of software health management. SafeMAT addresses the three key research challenges outlined in Section 2.2. This section first describes the underlying ARINC-653 Component Model middleware upon which SafeMAT is designed followed by a detailed design description of SafeMAT.

3.1. The ARINC Component Model (ACM) Framework

The ACM framework provides system-level health management by leveraging various failure reasoning techniques that are based on the notion of interacting components. We provide an overview of the different parts of the toolkit, which is necessary to understand the design and evaluation of SafeMAT.

3.1.1. The ARINC-653 Component Model (ACM)

ACM combines the CORBA Component Model [7] with ARINC-653 [5]. 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 single reusable component. Since this framework is geared for hard real-time systems, it is required that each port is statically allocated to an ARINC-653 process whereas every method of a facet interface is allocated to a separate operating system Environmentprocess.

3.1.2. The ACM Modeling Environment

The ACM modeling environment captures (1) the component's interaction ports, conditions associated with the ports, (2) the real-time properties (priority, periodicity, deadline, worst case execution time etc.) and resource requirements (CPU, stack size) of the ports and the component, the data and control flow within the component, and (optionally) a local component-level health management strategy (CLHM) part of a two-level Health Management [8] using a domain-specific modeling language and associated tools. The modeling tool allows the specification of the platform in terms of the modules (processors) and the partitions (processes) within each module. The integrator can specify the deployment of each component (group of threads) into an appropriate partition such that the *temporal partitioning* concerns are satisfied. Lastly, the integrator can specify whether a Software Health Management (SHM) module should be generated for the assembly or not. Tools included with the modeling environment generate glue code that is responsible for implementing the ports, binding each port with an ARINC-653 process and the integration code and configuration files.

3.1.3. The ACM Middleware

The ACM middleware shown in Figure 1 is composed of layers that are instantiated and configured for runtime. These layers are described next.

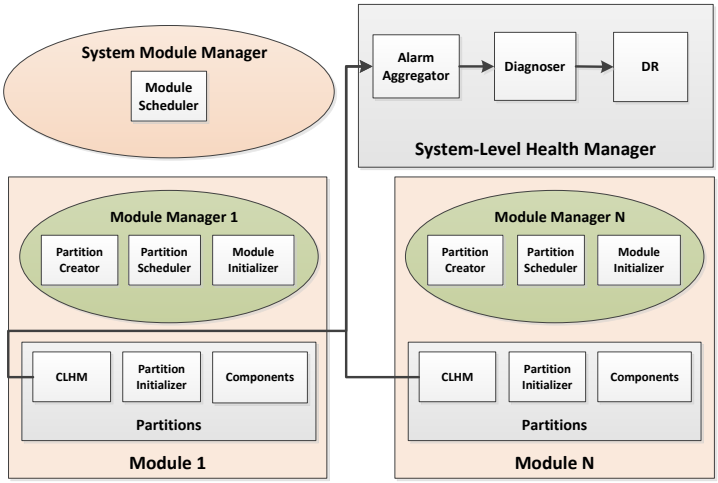


Figure 1: The ACM Middleware

The Module Manager (MM) is the main controller responsible for providing *temporal partitioning* among partitions (i.e., Linux processes). For this purpose, each module is bound to a single core of the host processor. The module manager 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 module manager implements the schedule using the `SCHED_FIFO` policy of the Linux kernel and manages the execution and preemption of the partitions. The module manager is also responsible for transferring the inter-partition messages across the configured channels. Figure 2 shows the example execution time line of a module with two partitions and a hyper period of 2 seconds.

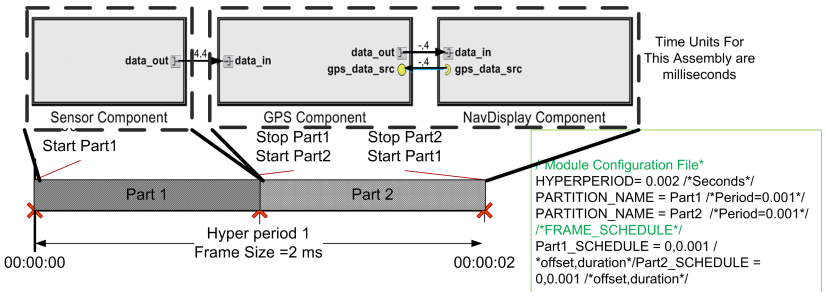


Figure 2: A module configuration and the time line of events as they occur

In case of a distributed system, there can be multiple module managers each bound to a processor core that are controlled hierarchically by a system level module manager.

The APEX Partition Scheduler is instantiated for each partition using the APEX services emulation library that implements a priority-driven preemptive scheduling algorithm using Linux `SCHED_FIFO` scheduler. It initializes and schedules the ARINC-653 processes inside the partition based on their periodicity and priority. It ensures that all processes, periodic as well as aperiodic, finish their execution within the specified deadline.

TAO Object Request Broker (ORB) [9] is an implementation of the real-time CORBA standard. The ORB thread is executed as an aperiodic ARINC-653 process within its respective partition. For controllability, the ORB runs at a lower priority than the partition scheduler does. Since ARINC does not allow dynamic creation of processes at run-time, the ORB is configured to use a predefined number of worker threads (i.e. ARINC-653 Processes) that are created during initialization.

Component and Process Layer is a layer that provides the glue code, generated from the definitions of components and their interfaces specified in the modeling environment in order to map the concepts of component model into the concepts exposed by the ARINC Emulator layer and the TAO ORB layer. The system developer provides the functional code. This layer also consists of CLHMs that are special processes that can take mitigation actions, if required.

3.1.4. Software Health Management (SHM) in ACM

Software Health Management (SHM) in ACM happens at two levels as shown in Figure 3. The first level of protection is provided by a component-level health management (CLHM) strategy, 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 System-Level Health Manager (SLHM) is at the second, top level in our health management strategy. The deployment of the SLHM requires the addition of three special SLHM components to an ACM assembly: the *Alarm Aggregator*, the *Diagnosis Engine*, and the *Deliberative Reasoner*.

The *Alarm Aggregator* is responsible for collecting and aggregating inputs from the component-level health managers (local alarms and the corresponding mitigation actions). This information is collected using a moving window, which is two hyperperiods long. The events are sorted based on their time of occurrence and then sent to the *Diagnosis Engine*. The *Diagnosis Engine* is initialized by a Timed Failure Propagation Graph (TFPG) [10] model that captures the failure-modes, discrepancies (possibly indicated by the alarms), and the failure propagations from failure modes to discrepancies and from discrepancies to other discrepancies, across the entire system [2, 11]. The reasoner uses this model to isolate the most plausible failure source: a software component that could explain the observations, *i.e.*, the alarms triggered and the CLHM commands issued. The result, *i.e.*, the list of faulty components is reported to the next component that provides the system level mitigation: the *Deliberative Reasoner*.

*3.1.5. **Fault Model and Fault Handling in ACM***

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

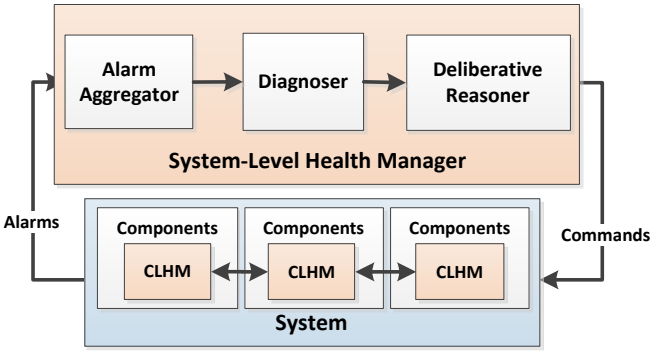


Figure 3: ACM Software Health Management Architecture.

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* [12] 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.

ACM (and hence SafeMAT) considers two main sources of failure for each component port (a) logical failure resulting from the 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 fail over to similar backup replicas with identical interfaces but alternate implementations (from different vendors/developers) with the hypothesis that the same logical error may not exist in an alternate implementation. In case of critical failures, we fail over 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 must always be deployed to different partitions in the same module or different modules of ACM (and hence SafeMAT).

3.2. SafeMAT Architecture

Figure 4 illustrates the architectural components of SafeMAT and their interactions. It depicts the underlying ACM middleware solution upon which SafeMAT is designed and implemented. The design of SafeMAT is driven by a holistic approach to answering the following questions on 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

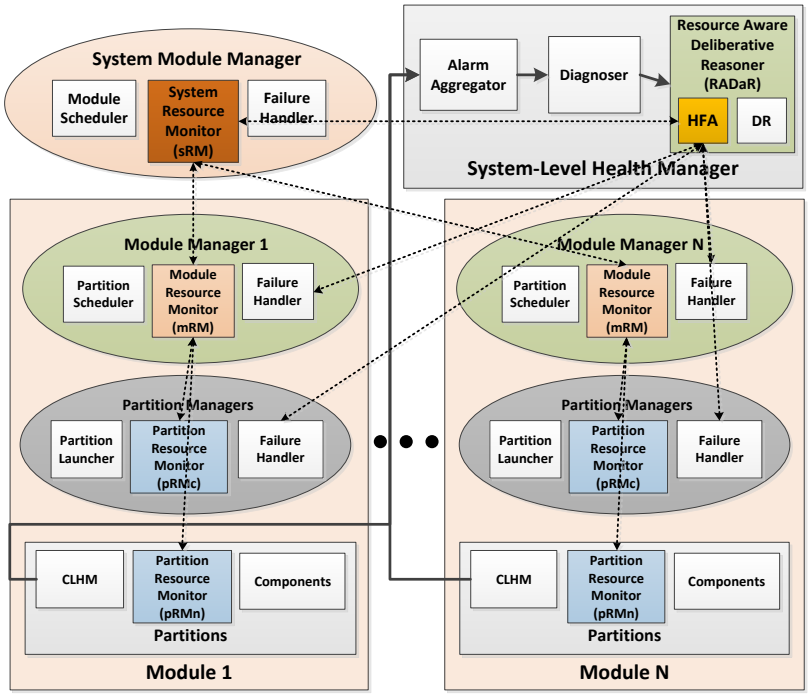


Figure 4: SafeMAT Architecture

decisions to deal with faults. The Distributed Resource Monitoring (DRM) framework in SafeMAT described in Section 3.4 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 3.5 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 3.6 provides this capability.

SafeMAT extensions to ACM have been architected in the form of a hierarchy of cooperating components. As shown in Figure 4, at the topmost level, SafeMAT extends ACM's System Module Manager with a System Resource Monitor (sRM) and failure handlers. Moreover, it introduces the Resource-Aware Deliberative Reasoner (RADaR) to ACM's SHLM. At the second level are the different Module Managers supplied by the original ACM that are deployed on each computing processor core or machine, each

hosting a Module Resource Monitor (mRM), which are newly introduced in SafeMAT. At the third level, SafeMAT introduces new architectural elements to ACM in the form of different Partition Managers that are responsible for managing each partition, each hosting a Partition Resource Monitor (pRM). The design details of the partition manager are shown in Figure 5.

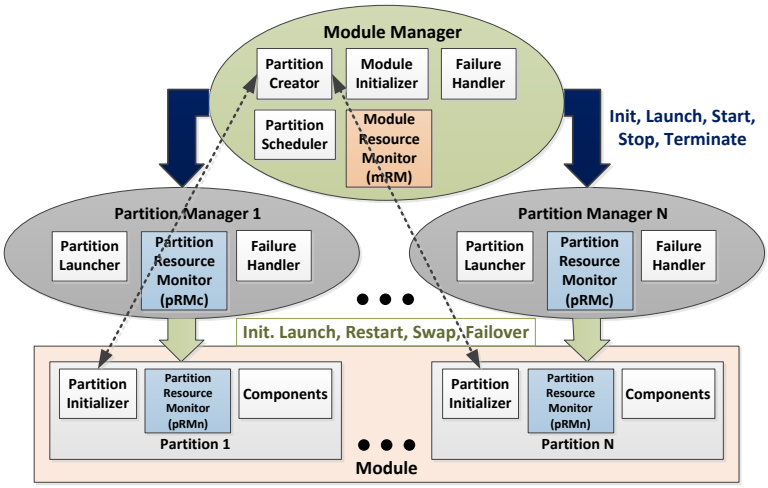


Figure 5: Partition Manager

Each of the managers include Failure Handlers to detect the failures in their respective partitions or modules and notifying them to the RADaR. The logical failures in components are notified by the respective CLHMs (from the original ACM framework) residing in each application component. The different monitors developed for SafeMAT form the core of the DRM framework whereas the RADaR along with the different Failure Handlers form the core of 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) (see Section 3.3). Doing so helps to isolate failures in partitions and mitigate partition faults by taking actions at the partition-level itself instead of involving the module manager. The following sections provide justification for these new architectural entities supplied by SafeMAT.

3.3. Isolating the Impact of Failed Partitions

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). As with any multiprocess system, processes can fail due to external factors such as driver faults, buffer overruns, segmentation faults, etc. It is necessary to enhance the safety of the real-time application by preventing failed partition processes from affecting the real-time schedule. The *Module Manager* handles the scheduling and execution of the partitions. Thus, whenever a partition process fails, it needs to ensure a quick recovery of that partition in

a way that will not affect the real-time application schedule. However, in order to achieve this, the *Module Manager* needs to stop its scheduler and focus on restarting and initializing the partition. So in order to detect and isolate the effect of the partition failure and relieve the *Module Manager* from handling the partition recovery, we have developed a *Partition Manager* as shown in the Figure 5. The *Partition Manager* is instantiated for each partition and is responsible for handling the execution and failure management of each individual partitions. *Partition Manager* coordinates with the RADaR described in Section 3.5 for managing the partition failures and their recovery.

Algorithm 1 Algorithm for the `Partition Manager`

```
1: procedure PARTITION MANAGER(P)                                  ▷ Set of all partitions.
2:     Set scheduling policy to SCHED_FIFO
3:     Set CPU affinity to a single core.
4:     for channel ∈ CHANNELS do
5:         for p ∈ P do
6:             if SRC(channel) ∈ SRC_SP(p) or SRC(channel) ∈ SRC_QP(p) then
7:                 p.SRCCHANNEL.append(channel)
8:             end if
9:         end for
10:    end for
11:    for p ∈ P do
12:        OFFSETS.add(OFFSET(p), p)
13:    end for
14:    Sort OFFSETS in ascending order based on the time value.
15:    for p ∈ P do
16:        childprocess =fork(EXECUTABLE(p))
17:        Wait on handshake from childprocess subject to a timeout
18:        if timeout then
19:            Shutdown Module
20:            Exit
21:        end if
22:    end for
23:    for entry in OFFSETS do
24:        T1 ← currenttime
25:        T2 ← T1 + DURATION(entry.Partition)
26:        Send SIGCONT to EXECUTABLE(entry.Partition)    ▷ SIGCONT is a POSIX
   signal
27:        clock_nanosleep(T2) ▷ Use a high-resolution clock such as clock real-time in Linux.
28:        Send SIGSTOP to EXECUTABLE(entry.Partition)    ▷ SIGSTOP is a POSIX
   signal
29:        for c ∈ entry.Partition.SRCCHANNEL() do
30:            c.fire()    ▷ Move message from source port of the channel to all the destination
   ports
31:        end for
32:    end for
33: end procedure
```

Algorithm 1 illustrates the responsibilities of the partition manager. Whenever, it detects a partition failure, it restarts the partitions if instructed by the RADaR. Moreover,

it also hosts the pRM to enable computation the resource utilization of the partition process and its constituent component threads. It also ensures that if the partition is restarted then it does not need to re-perform the synchronization with the *Module Manager* in order to save time and be ready and initialized before its next scheduling quantum arrives.

3.4. Distributed Resource Monitoring: the DRM Framework

The Distributed Resource Monitoring (DRM) framework resolves Challenge 1 of Section 2 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*, which are applicable in the COMPUTE mode, *i.e.*, during the period of ongoing computations, and (b) *pMRn*, which are applicable in the NOTIFY mode, *i.e.*, during the period of ongoing notifications.

3.4.1. Configurability in the 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 explained in Section 3.6. Finally, it is also possible to configure the DRM framework to supply only the utilizations of the specific entities that RADaR is interested in. Table 1 shows the different valid configurations at the different levels of DRM.

Table 1: DRM Configuration and Compatible Strategies

sRM	**mRM**	**pRM (Partition Manager)**	**pRM (Partition)**
ONDEMAND	REACTIVE	REACTIVE, COMPUTE	NOTIFY
PERIODIC	REACTIVE	REACTIVE, COMPUTE	NOTIFY

3.4.2. Discovering Resource Allocations

The DRM framework is also capable of discovering the deployment and allocations of components to specific partitions and modules at runtime thereby obviating the need to configure the framework manually and 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 sends the mappings of the deployed components and their subtasks but does not compute the resource utilizations. 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. The pRMc computes the resource utilizations and is configured within the partition in the COMPUTE mode. 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 for the components, component groups, and modules.

3.4.3. Resource Liveness Monitoring: Overcoming Single Point of Failures

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 enhancement 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 also.

3.5. Resource-Aware Adaptive Failure Mitigation: the AFM Framework

To perform resource-aware failure adaptation and to address Challenge 2 of Section 2, we have developed the Adaptive Failure Mitigation (AFM) framework 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) [3] of the SLHM. Figure 6 shows the basic flowchart for the HFA algorithm. The details of the algorithm are provided below. 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.

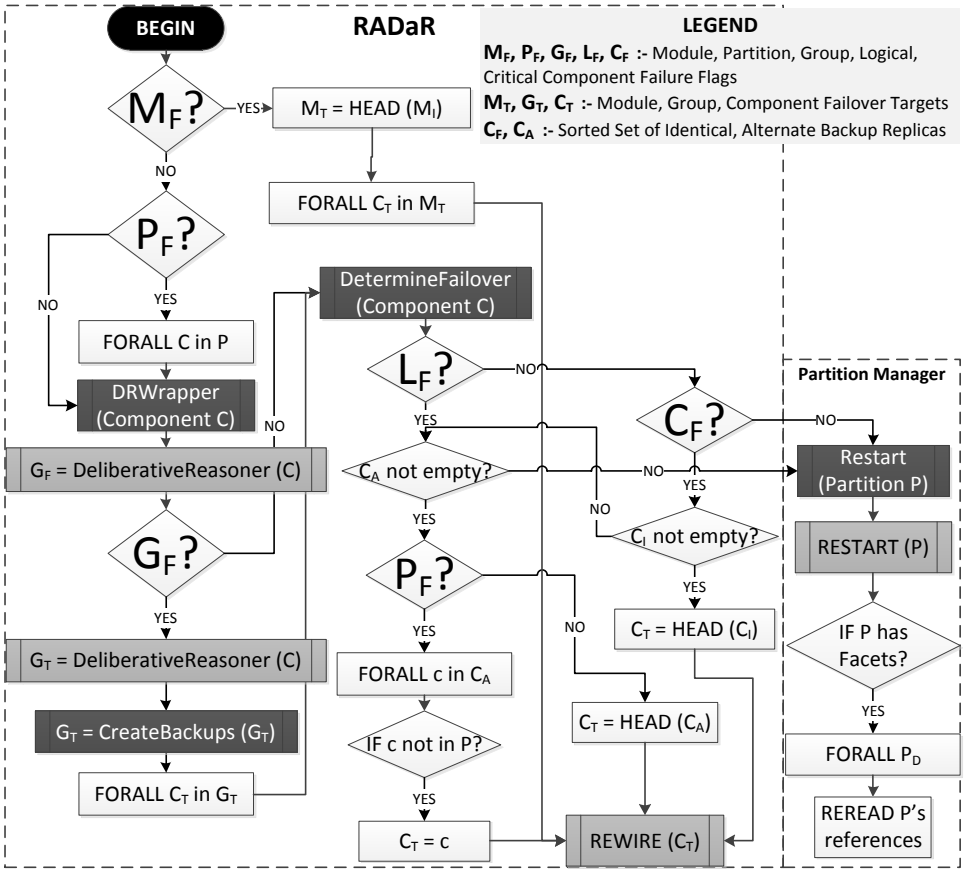


Figure 6: The HFA Algorithm

3.5.1. 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 ACM modeling paradigm allows various deployment scenarios for the primary components and their backups as shown in the Figure 7.

For instance, the application component primaries and their corresponding backups can be deployed within the same module or can be spread across multiple modules. Moreover, they can either be deployed within the same partition or different partitions depending whether they are identical instances or alternate implementations of the primary replica. If the backups are an identical replica then by convention they are never deployed within the same partition as they are meant to handle critical failures that

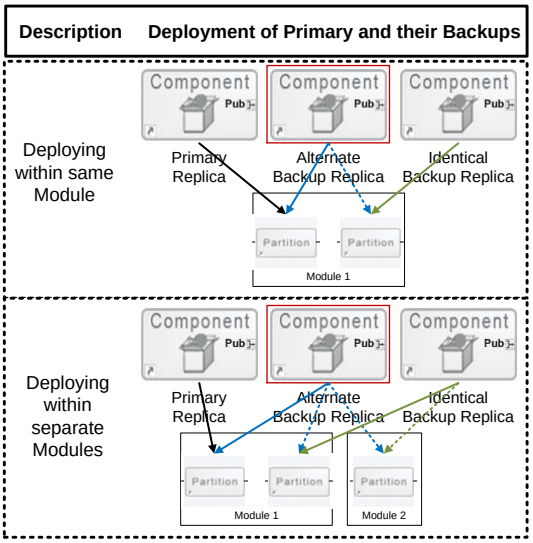


Figure 7: Backup Deployment Scenarios

usually result in the process or the processor crashing. However, backups with alternate component implementations can be deployed within the same partition as they are meant to handle latent errors in the component's implementation logic.

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 fail over 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.

3.5.2. Enabling Hierarchical Failure Adaptation (HFA)

We have developed the 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 or 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 with the listing of 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. Algorithm 2 depicts the details of the HFA algorithm.

Algorithm 2 The Hierarchical Failover Adaptation (HFA) Algorithm

```
1:  procedure HFA(M, P, G, C)                                   ▷ Input Parameters
2:      M, P, G, C : Module, Partition, Group, Component Failed Primaries
3:      M_F, P_F, G_F, C_F, L_F : Flags for Module, Partition, Group, Critical and Logical
    component Failures
4:      M_I, G_I, C_I, C_A : Sorted List of Identical & Alternate Backup Replicas   ▷
    Output Parameters
5:      M_T, G_T, C_T : Failover Target Backup Replicas
6:      if M_F then
7:          M_T ← HEAD(M_I)
8:          for all C_T ∈ M_T do
9:              REWIRE(C_T)
10:         end for
11:     else if P_F then
12:         for all C ∈ P do
13:             DRWrapper(C)
14:         end for
15:     else
16:         DRWrapper(C)
17:     end if
18: end procedure
```

1. **Lines (1-6):** The failed primaries alongwith the type of failure is provided by the Failure Handlers whereas the sRM provides the least utilized backups. The RADaR maintains the current status of the set of backup replicas available in sorted order based on their utilizations and it updates and rewires the failover target backups after the algorithm finishes it's execution.
2. **Lines (8-14):** If a module (processor) failure is detected, the Failure Handlers in the System Module Manager instruct the sRM to gather the utilizations of the backup modules and provide the module with the least utilization and finally report the failure to the RADaR.
3. **Lines (15-21):** If a process (partition) failure is detected, the algorithm checks if it was due to a component or logical failure and call the subroutine DetermineREWIRE (C) that handles the further failover
4. **Lines (21-23):** In case of a critical component failure, again the control is handled over to the DetermineREWIRE (C) subroutine.
5. *DetermineREWIRE (Component C)*

   a. **Lines (1-4):** In case of a logical component failure and alternate backup replicas if

```
19: procedure DRWrapper(C)                                   ▷ Component
20:     G_F ← DeliberativeReasoner(C)
21:     if G_F then
22:         G_T ← DeliberativeReasoner(C)
23:         G_I ← CreateBackups(G_T)
24:         G_T ← HEAD(G_I)
25:         for all C_T ∈ G_T do
26:             DetermineFailover(C_T)
27:         end for
28:     else
29:         DetermineFailover(C)
30:     end if
31: end procedure
```

```
32: procedure DetermineFailover(C)                           ▷ Component
33:     if L_F then
34:         CheckAlternate (C)
35:     else if C_F then
36:         if C_I ≠ ∅ then
37:             C_T ← HEAD(C_I)
38:         else
39:             CheckAlternate (C)
40:         end if
41:     else
42:         Restart(P)
43:     end if
44:     REWIRE(C_T)
45: end procedure
```

available, the one with the least utilized component utilization is selected or else if available, one with the least utilized identical backup replicas is selected

b. **Lines (5-7):** If it is not a logical failure and there are no more backup replicas are left then the partition is restarted in the hope that failure will not recur.

c. **Line (8):** The Deliberative Reasoner is called to figure out if the dependent group of components also need failover. In this case the dependent group can be a entire subsystem.

d. **Lines (9-12):** If Group failure is detected, the failover target component group determined by the Deliberative Reasoner is updated with head of the list of alternate or identical backup components and a rewire command is issued to the group.

e. **Lines (13-14):** If only the primary component needs failover, it's least utilized backup component is rewired.

6. *RESTART (Partition P)*

a. **Lines (1-4):** Restart the partition and if it has any component facets then the partitions with the dependent component receptacles are instructed to reread the

```
1: procedure CheckAlternate(C)                                  ▷ Component
2:     if C_A ≠ ∅ then
3:         if P_F then
4:             for all c ∈ C_A do
5:                 if c ∋ P then
6:                     C_T ← c
7:                 end if
8:             end for
9:         else
10:            C_T ← HEAD(C_A)
11:        end if
12:    else
13:        Restart(P)
14:    end if
15: end procedure
```

```
1: procedure Restart(P)                                         ▷ Partition
2:     RESTART(P)
3:     if P has provided interfaces then
4:         for all p ∈ P_d do
5:             REREAD(P's references)
6:         end for
7:     end if
8: end procedure
```

component references.

At the core of the HFA algorithm 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 is 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 is 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 selects the least utilized backup target group of components and finally calls DetermineFailover on each component in the failed group.

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

```
1: procedure CreateBackups(G)                                   ▷ Group
2:     if L_F then
3:         if C_A ≠ ∅ then
4:             for all C1 ∈ G do
5:                 for all C2 ∈ C1_A do
6:                     if P_F then
7:                         if C2 ∈ P then
8:                             continue
9:                         end if
10:                        G_I ← G_I ∪ C2
11:                    end if
12:                end for
13:            end for
14:        end if
15:    else if C_F then
16:        if C_I ≠ ∅ then
17:            for all C1 ∈ G do
18:                for all C2 ∈ C1_I do
19:                    G_I ← G_I ∪ C2
20:                end for
21:            end for
22:        end if
23:    end if
24:    SORT(G_I)
25: end procedure
```

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.

3.6. Pre-deployment Application Performance Evaluation: the PME Framework

The real-time system execution schedule specifies the period of execution along with the allocated start and end times of the system tasks forming the scheduling quantum within the system execution time period (P). To address Challenge 3 of Section 2, 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.

4. Implementation of SafeMAT

SafeMAT has been implemented atop the ACM hard real-time ARINC-653 emulation middleware (implemented on Linux). It is implemented in around 5,000 lines of C/C++ source code excluding the ACM code. We describe the implementation details of the individual frameworks of SafeMAT in the rest of this sectionimpl.

4.1. Partition Manager

We have implemented the *Partition Manager* as a separate Linux 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. In order for the partitions to correctly synchronize back with the *Module Manager*, the *Module Manager*'s PID is also set as one of the environment parameters along with the partition name, and the boolean indicating whether the partition is being restarted. The *Partition Manager* implements signal handlers as a means of handling partition failures. Whenever the failure handlers detect a partition failure, they check its exit status after receiving a `SIGCHLD` and if it is an abnormal termination, the *Partition Manager* restarts the partition and also sets the boolean to true. If the boolean is set to true then the partitions do not need to re-perform the synchronization with the *Module Manager* and can quickly recover in time before their next scheduling quantum in the next hyperperiod. Furthermore, in order to handle partition restarts as described in the HFA algorithm in Section 3.5, if the facet side partition needs restarting the facet reference is reread for the receptacle side partition. This can be achieved through catching the invalid object reference exception and/or by sending a message to the partitions.

4.2. Distributed Resource Monitoring (DRM) Framework

We have developed the DRM using the client-server paradigm that can be configured with two different communication 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 bounded network propagation delay. The sRMs are in charge of configuring the mRMs and pRMs with the communication strategies so that the clients need to worry about correctly configuring all the monitors and thereby alleviating the need for configuration checking before deployment. This is achieved by making the mRM always initiate the first communication to setup and configure the monitors with the right strategy, the CPU number on which the module is deployed, the name of the partition to be

monitored. The port at which they are expected to receive the messages is set through the environment variables while spawning the Partition Managers which forwards this information to the partitions that configure the pRMs. Additionally each of the monitors of the DRM framework are also programmed to perform their health monitoring by periodically sending their health status beacons to the their immediate parents through `ALIVE` socket messages. This aids in the detection of the partition (process) and module (processor) failures.

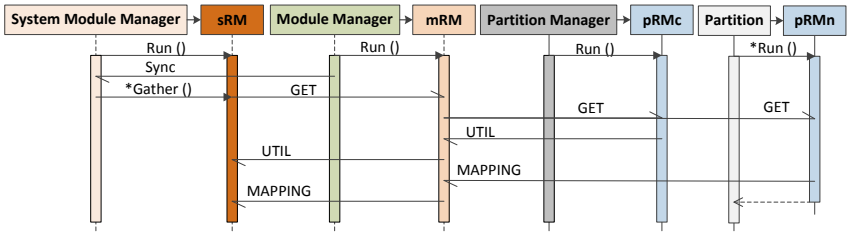


Figure 8: Distributed Resource Monitoring (DRM) Communication Sequence

The pRMc computes the processor, process and thread utilizations from the `/proc/stat`, the corresponding `/proc/<PID>/stat` and `/proc/<PID>/task/<TID>/stat` files on Linux. In the ACM emulated middleware we associate the module, partition and component utilizations with the respective processor, process and thread group utilizations. The partition and component utilizations are always computed as a fraction of the processor utilizations on which they are deployed on to reflect their true utility to the AFM engine. To allow for efficient querying, the mRMs and the sRM maintain the mappings of the component allocations to their partitions and modules so that the AFM engine can selectively query the utilizations of specific components, partitions and modules. The PIDs are reported back by the pRMs to their corresponding mRM within the *Module Manager* when the partitions notify their initialization statuses to the *Module Manager* through the Linux message queues `/dev/mqueue/<Q-NAME>`.

The pRMns within the partition communicate with the partition initialization logic and obtains the list of components assigned and deployed within that partition. The partition initializer also reports the corresponding ARINC-653 process (Linux Pthreads) identifiers (TIDs) that execute the different ports and methods within a component. This mapping of the components to their corresponding TIDs is reported back by the pRMns to their corresponding mRM. Once the mRM has the necessary partition PIDs and the component to TID mappings, it enables the sRM to report CPU utilizations on a per component or a per process or a per module basis whenever queried by the SLHM components. The sequence of communications that occur between the sRM, mRM, pRMc, and pRMn components is shown in Figure 8.

4.3. Adaptive Failure Mitigation (AFM) Engine

The Diagnoser and Deliberative Reasoner components from the SLHM framework have been extended by integrating the HFA algorithm and DRM frameworks. We have

developed the Resource Aware Deliberative Reasoner (RADaR) by improving the reasoning algorithm employed by the Deliberative Reasoner (DR) within the SLHM framework to compute component failover targets by considering the CPU utilizations, failure type, failure granularity and the deployment topology. We have incorporated the failure detection of the partitions and modules through failure handlers and DRM health status monitoring. Additionally, in order to detect logical or component failures in case of simultaneous partition or module failures, the RADaR traverses the history of any failures that were caught by the Alarm Aggregator and the output files generated out by the CLHMs that indicate the failure types. This gives us the capability to handle both logical component failures as well as critical process and processor failures simultaneously within the same framework. The HFA algorithm provides a wrapper over the DR's reasoning algorithm.

We integrated the HFA algorithm in the decision making part of the deliberative reasoning algorithm that gets executed each time the DR gets invoked with the failed components. First it uses the DR's dependency tracking phase to figure out the dependent group of component's that require failover and the failover candidates initially generated by the DR. The DR achieves this through a search of the component's assembly specification and deduces the failed component's dependencies and determines whether the dependent components also need recovery. When the DR comes up with the initial failover target component or a group of components, they may not be necessarily the best candidates. We select the best failover target for the failed component by executing the HFA algorithm on the initial result of the DR and manipulate the DR's output with the better candidates provided by our algorithm. The HFA algorithm achieves this by querying the sRM for the sorted rank lists of failover target backup replica components based on their relative utilizations.

4.4. Application Performance Metrics Evaluation (PME) Framework

We profile the SafeMAT component's actual WCETs and 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 out 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. The analysis of the timing log files is performed by parsing the standard tags such as `START_*`, `STOP_*` corresponding to the start and stop times of the various processing blocks using Python scripting. 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. We particularly profile the utilizations of the Health Management and SafeMAT framework components to verify that the utilizations do not reach 100% so that they are able to finish their decision making within the allocated quantum . This ensures that the recovery from failures is made as fast as possible.

5. Empirical Evaluation of SafeMAT

To measure the performance of the various SafeMAT adaptive mechanisms, we used a representative DRE system called the Inertial Measuring Unit (IMU) [13] 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) [14], the flight control (PFC) subsystem, and the Display subsystem. Figure 9 shows the IMU system assembly comprising primary subsystems of GPS and ADIRU, and their two secondary backup replica subsystems connected to redundant PFC and Display subsystems.

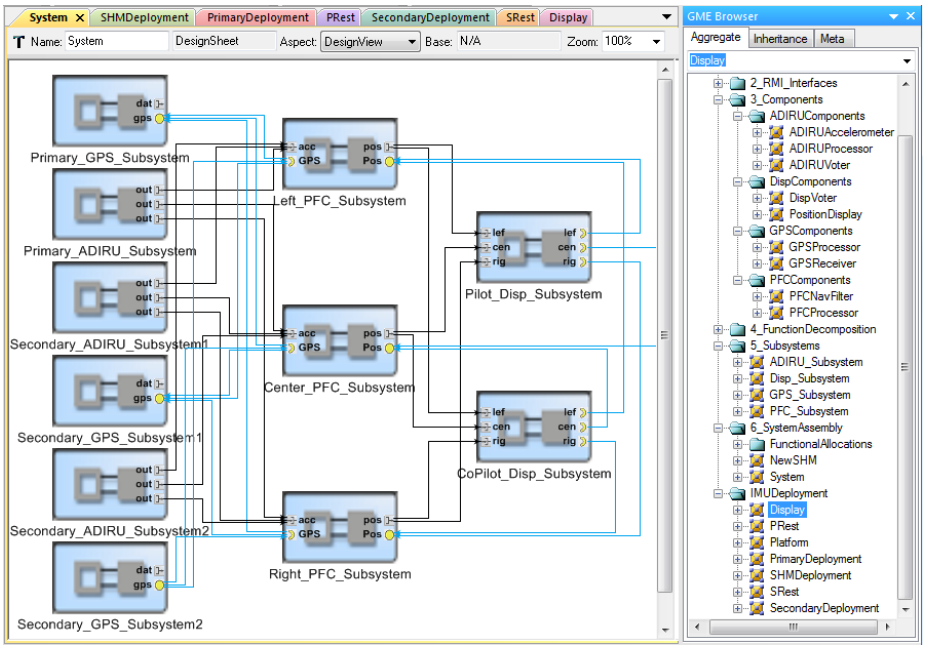


Figure 9: IMU System Assembly.

When the GPS processor has an updated position, it sends a pulse out of its publisher port and the subscriber GPS Receiver can asynchronously detect it and fetch the data coordinates. The ADIRU subsystem comprises actively replicated six Accelerometers, four Processors, and three Voters and is designed to withstand two Accelerometer failures. The six Accelerometers feed acceleration values to each of the four Processors which compute the body acceleration data and fed it to each of the three Voters. In turn the Voters choose the middle value and output it to the PFC subsystem. The GPS Processor and the ADIRU Voter feed the 3D location coordinates and acceleration values, respectively, to each of the PFC subsystem that integrates the acceleration values over the 3D coordinates and computes the next coordinate position and outputs it to the Display subsystem which further votes and chooses one of the three coordinate values received. The Secondary GPS and ADIRU subsystems are semi-actively replicated.

The GPS subsystems and ADIRU subsystems run at a frequency of 0.1 Hz and 1 Hz, respectively. The PFC fetches the GPS data 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). Deployment information

of all the subsystems is not shown in this paper for similar reasons. However, we discuss the impact of various primary-backup deployments on the overall runtime adaptation overhead added by SafeMAT by going into the deployment details of the standalone adaptation of the GPS Subsystem in Section 5.2.[4]

5.1. 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.

Figure 10 shows the 3D-position (X, Y, Z coordinates) values received by the `Pilot_Display_Subsystem` getting out of sync between iterations 30 and 40, and recovering after the SafeMAT failure adaptation takes place. The perturbation is caused by the erroneous acceleration values because the IMU is solely capable of operating using just the acceleration values without the need for continuous GPS input. GPS coordinates are used to just supply the initial coordinates for the integration over the acceleration values computed by the PFC subsystem.

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 11 shows that the SafeMAT does not add significant utilization overhead (2-6%) over the existing ACM-SHM imposed utilizations (26-73.26%).

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

[4]The details of each subsystem and their deployments have been omitted in this paper for the lack of space, which can be found in [13].

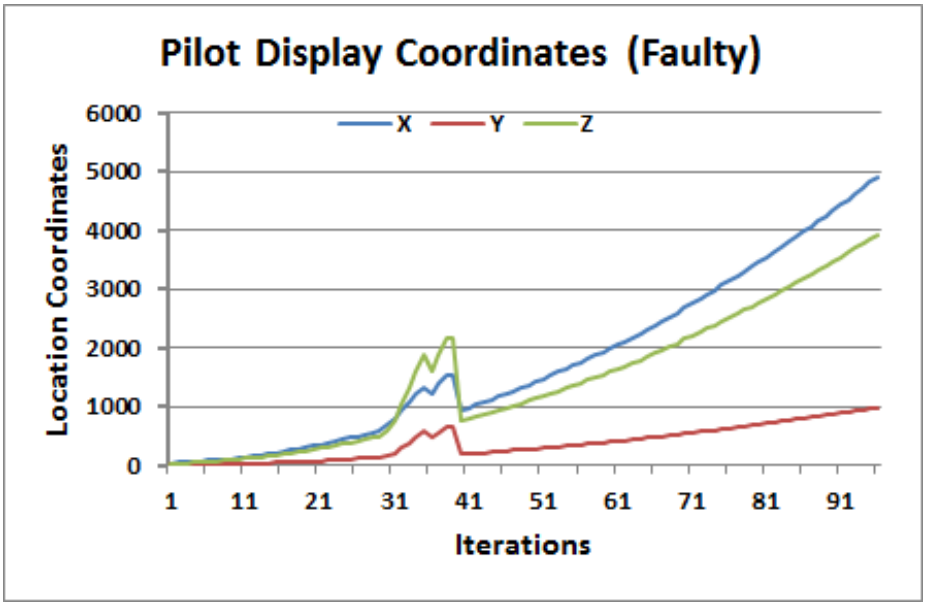


Figure 10: Application Recovery after Failover

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

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

5.2.1. Impact of Component Replica Deployments

To measure the impact of component replica deployments, we focused on the GPS subsystem from the IMU case study. Figure 12 shows the assembly for the BasicSP system with a redundant set of Sensor and GPS components (Sensor2 Sensor3, GPS2, GPS3). Sensors publish an event every 2 sec for their associated GPS. The GPS consumes the event published by its sensor at a periodic rate of 2 sec. Afterwards, it publishes an event, which is sporadically consumed by the Navigation Display. Thereafter, the NavDisplay component updates its location by using getGPSData facet of the GPS Component. In the initial setup of the assembly, the Sensor, GPS, and NavDisplay components are used and hence set to be in *active mode*. The redundant Sensor and GPS (Sensor2 Sensor3, GPS2, GPS3) are not used. The GPS2 & GPS3 is set to a *semi-active mode*, leaving

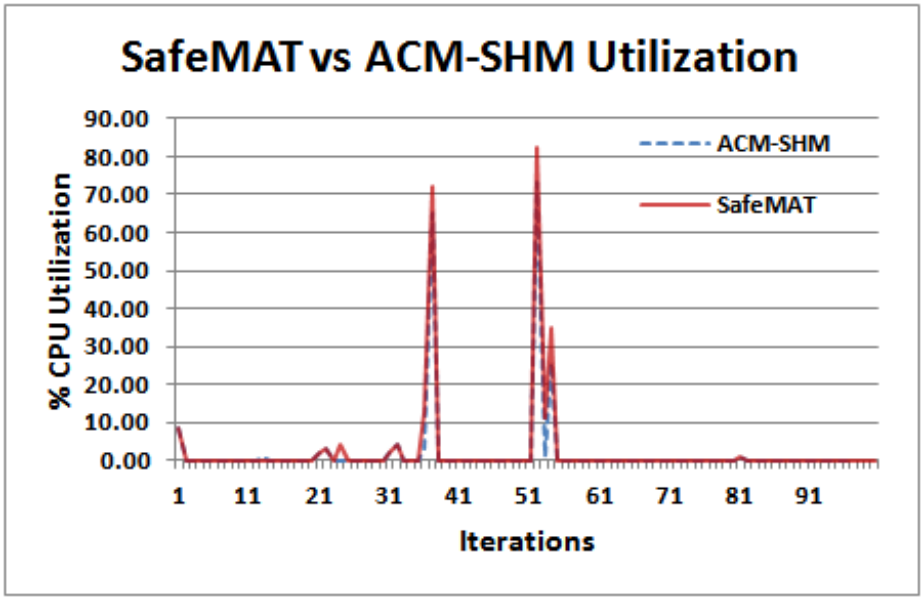


Figure 11: SafeMAT Utilization Overhead

the Sensor2 & Sensor3 components in active mode. This would allow the GPS2 & GPS3 to keep track of the current state (by being in semi-active mode where the GPS2's and GPS3's consumers are active) but not affecting NavDisplay.

We created different deployment scenarios by altering the placements of the component replica by either placing them either within the same partition as primary (Figure 13a), or a different partition in the same module (Figure 13b) or a different partition within a different module (Figure 13c). 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 14, 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 (See Figure 15. 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.

5.2.2. 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 16, when the number of components increase, the SafeMAT overhead costs gets

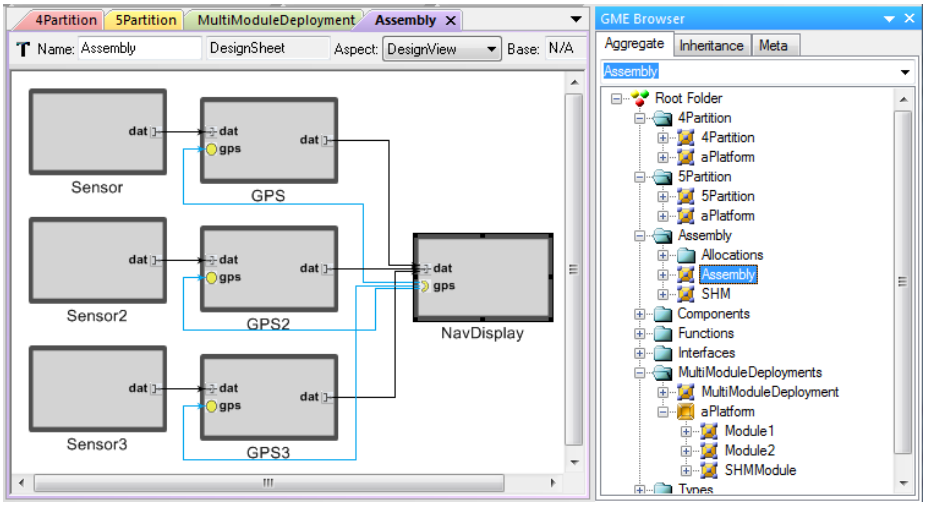
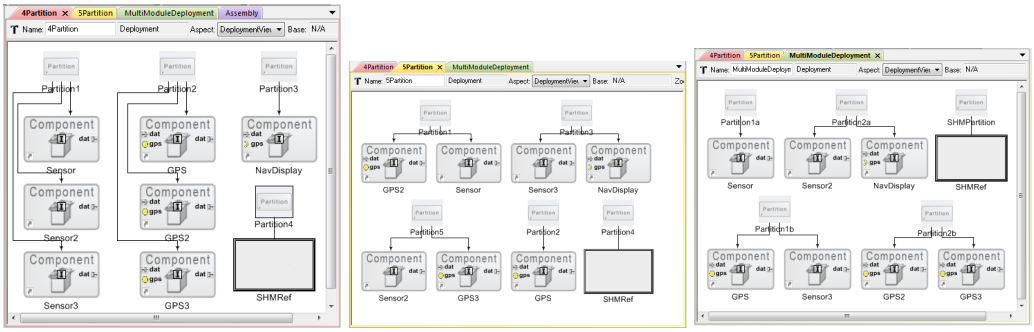


Figure 12: GPS (BasicSP) Subsystem Assembly



(a) Same Partition as Primary (b) Different Partition as Primary (c) Different Module as Primary

Figure 13: Different Component Replica Deployments

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.

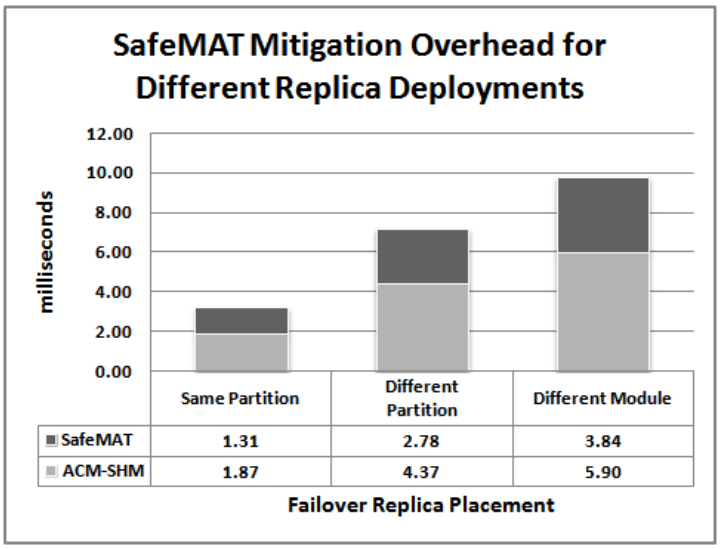


Figure 14: SafeMAT Mitigation Overhead for Different Replica Deployments

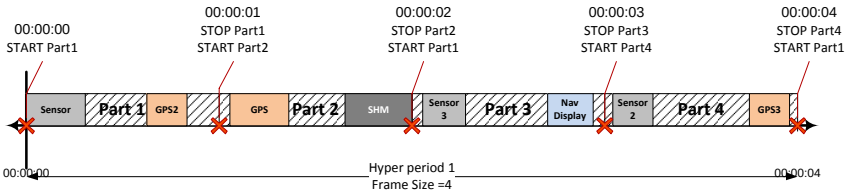


Figure 15: Slack in GPS Schedule

5.3. 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 17 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. Furthermore, 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.

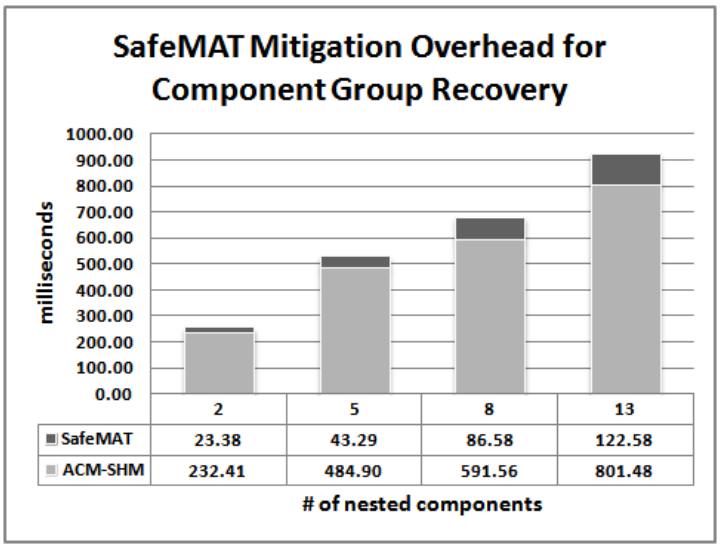


Figure 16: SafeMAT Mitigation Overhead for Component Group Recovery

6. Related Work

In this section we discuss 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 prior work along the following three different dimensions and subsequently summarize the limitations in related work that motivated the research on SafeMAT.

6.1. Resource-aware Adaptations

The DARX framework [15] 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. In [16], an adaptive fault tolerance mechanism is proposed to choose a suitable redundancy strategy for dynamically arriving aperiodic tasks based on system resource availability. Research performed in AQUA [17] 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 [18] 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. In [19], an adaptive dependability approach is presented which mediates interactions between middleware and applications to resolve constraint consistencies while improving availability of distributed systems.

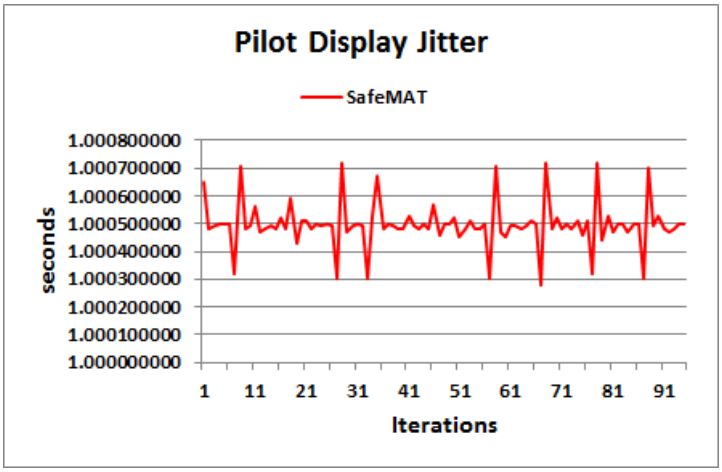


Figure 17: Application Display Jitter (Hyperperiod = 1 sec)

6.2. Real-time Fault-tolerant Systems

Our SafeMAT work is influenced by the Time-aware fault Tolerance (TAFT) [20] work in that we leverage the architectural patterns in SafeMAT. While TAFT was applied to CORBA/C++-based systems, we extended the work to systems that following the ARINC-653 model with partition scheduling, and also provide comprehensive software health management. IFLOW [21] and MEAD [22] 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 [23] 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 [24] 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 [25] provided real-time fault-tolerant solutions to distributed systems by using the semi-active replication model. Other research [26] uses simulation models to analyze multiple checkpointing intervals and their effects on fault recovery in fault-tolerant distributed systems. The ROAFTS architecture [27] is designed to support adaptive fault tolerance in both process-structured and object-structured distributed real-time applications. ROAFTS considers those fault tolerance schemes for which recovery time bounds can be easily established and provides quantitative guarantees on the real-time fault tolerance of these schemes.

6.3. Dynamic Scheduling

Common methodologies to leverage the slack in execution schedule have focused on dynamic scheduling depending upon the runtime conditions. The Realize middleware [28]

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.

6.4. Limitations in Prior Work and 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.

7. Conclusion

Mission-critical hard real-time applications that are 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 for worst-case scenarios. While, existing software health management (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 and 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 is manifested in the form of the SafeMAT middleware. 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.

The underlying task model for DRE systems in SafeMAT and its performance evaluation are conducted assuming the ARINC-653 task model. ARINC-653 is a standard to

support hard real-time avionics applications. Since DRE systems are often represented by a mix of different criticality levels, there is a need to segregate these applications and ease verification. This is achieved through the partitioning scheme supported by ARINC-653 that provides both temporal and spatial isolation among the applications.

Acknowledgments

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.

References

[1] A. Srivastava, 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.

[2] A. Dubey, G. Karsai, N. Mahadevan, Model-based software health management for real-time systems, in: Aerospace Conference, 2011 IEEE, 2011, pp. 1–18.

[3] A. Dubey, N. Mahadevan, G. Karsai, A deliberative reasoner for model-based software health management, in: The Eighth International Conference on Autonomic and Autonomous Systems, IARIA, St. Maarten, The Netherlands Antilles, 2012, pp. 86–92.

[4] J. Balasubramanian, S. Tambe, C. Lu, A. Gokhale, C. Gill, 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, 2009, pp. 118–127.

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

[6] A. Dubey, G. Karsai, N. Mahadevan, A Component Model for Hard Real-time Systems: CCM with ARINC-653, Software: Practice and Experience 41 (12) (2011) 1517–1550. doi:10.1002/spe.1083.
URL http://dx.doi.org/10.1002/spe.1083

[7] N. Wang, D. C. Schmidt, C. O'Ryan, An Overview of the CORBA Component Model, in: G. Heineman, B. Councill (Eds.), Component-Based Software Engineering, Addison-Wesley, Reading, Massachusetts, 2000.

[8] A. Dubey, G. Karsai, N. Mahadevan, Towards model-based software health management for real-time systems., Tech. Rep. ISIS-10-106, Institute for Software Integrated Systems, Vanderbilt University (August 2010).
URL http://isis.vanderbilt.edu/node/4196

[9] Institute for Software Integrated Systems, The ACE ORB (TAO), www.dre.vanderbilt.edu/TAO/ (Vanderbilt University).

[10] S. Abdelwahed, G. Karsai, N. Mahadevan, S. C. Ofsthun, Practical considerations in systems diagnosis using timed failure propagation graph models, Instrumentation and Measurement, IEEE Transactions on 58 (2) (2009) 240–247.

[11] N. Mahadevan, A. Dubey, G. Karsai, Application of software health management techniques, in: Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, ACM, ACM, New York, NY, USA, 2011.

[12] A. M. Déplanche, P. Y. Théaudière, 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, IEEE Computer Society, Washington, DC, USA, 1999, p. 90.

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

[14] M. McIntyre, 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, 1995, pp. 178 –183. doi:10.1109/DASC.1995.482827.

[15] O. Marin, M. Bertier, 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, IEEE Computer Society, Washington, DC, USA, 2003, p. 406.

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

[17] S. Krishnamurthy, W. H. Sanders, M. Cukier, An Adaptive Quality of Service Aware Middleware for Replicated Services, IEEE Transactions on Parallel and Distributed Systems 14 (11) (2003) 1112–1125. doi:http://doi.ieeecomputersociety.org/10.1109/TPDS.2003.1247672.

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

[19] L. Froihofer, K. M. Goeschka, J. Osrael, Middleware support for adaptive dependability, in: Middleware, 2007, pp. 308–327.

[20] E. Nett, M. Gergeleit, M. Mock, An Adaptive Approach to Object-Oriented Real-Time Computing, in: IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE Computer Society, Kyoto, Japan, 1998, p. 342. doi:http://doi.ieeecomputersociety.org/10.1109/ISORC.1998.666806.

[21] Z. Cai, V. Kumar, B. F. Cooper, G. Eisenhauer, K. Schwan, 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.

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

[23] K. H. K. Kim, 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. 12 (2). doi:dx.doi.org/10.1109/69.842258.

[24] X. Wang, Y. Chen, C. Lu, X. Koutsoukos, FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization controlstar, open, Journal of Systems and Software 80 (7) (2007) 938–950.

[25] D. Powell, Distributed Fault Tolerance: Lessons from Delta-4, IEEE Micro 14 (1) (1994) 36–47. doi:dx.doi.org/10.1109/40.259898.

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

[27] K. H. Kim, ROAFTS: A Middleware Architecture for Real-time Object-Oriented Adaptive Fault Tolerance Support, in: High Assurance Systems Engineering Symposium (HASE '98), IEEE CS, 1998, pp. 50–57.

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