SAFER: System-level Architecture for Failure Evasion in Real-time Applications

Junsung Kim, Gaurav Bhatia, Ragunathan (Raj) Rajkumar
Electrical and Computer Engineering, Carnegie Mellon University
{*junsungk, gnb, raj*}*@ece.cmu.edu*

Markus Jochim
Research and Development, General Motors
markus.jochim@gm.com

***Abstract*—Recent trends towards increasing complexity in distributed embedded real-time systems pose challenges in designing and implementing a reliable system such as a self-driving car. The conventional way of improving reliability is to use redundant hardware to replicate the whole (sub)system. Although hardware replication has been widely deployed in hard real-time systems such as avionics, space shuttles and nuclear power plants, it is significantly less attractive to many applications because the amount of necessary hardware multiplies as the size of the system increases. The growing needs of flexible system design are also not consistent with hardware replication techniques.**

To address the needs of dependability through redundancy operating in real-time, we propose a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features to tolerate fail-stop *processor* and *task* failures for distributed embedded real-time systems. To detect such failures, SAFER monitors the health status and state information of each task and broadcasts the information. When a failure is detected using either *time-based failure detection* or *event-based failure detection*, SAFER reconfigures the system to retain the functionality of the whole system. We provide a formal analysis of the worst-case timing behaviors of SAFER features. We also describe the modeling of a system equipped with SAFER to analyze timing characteristics through a model-based design tool called SysWeaver. SAFER has been implemented on Ubuntu 10.04 LTS and deployed on Boss, an award-winning autonomous vehicle developed at Carnegie Mellon University. We show various measurements using simulation scenarios used during the 2007 DARPA Urban Challenge. Finally, we present a case study of failure recovery by SAFER when node failures are injected.

***Keywords*-fault-tolerance; real-time; task-level replication; distributed; embedded; hot standby; cold standby.**

I. Introduction

Advances in distributed embedded real-time systems have enabled a variety of different applications such as industrial control systems, avionic systems and automotive systems which are tightly coupled with the physical world. Such applications need to satisfy strict timing constraints based on operating characteristics, making timing guarantees an essential requirement. Furthermore, system reliability is of high importance for safety-critical applications that interact with the physical world. However, a trend towards increasing complexity in distributed embedded real-time systems poses challenges in designing a reliable system such as a self-driving car depicted in Figure 1 [1].

A conventional way of improving reliability is adding redundant hardware to replicate the whole (sub)system. All hardware replicas run identical copies of the same software, and voting is used to choose the correct output among the

Figure 1. Boss: an award-winning autonomous vehicle developed at CMU

candidates; however, this approach becomes less attractive to many systems with cost and space constraints. Hardware replication also tends to reduce the system flexibility. We advocate a software replication approach that provides many of the same benefits as hardware replication while maintaining lower costs and a higher level of flexibility.

Either *active* or *passive* replication can be used for supporting fault tolerance on distributed embedded real-time systems. Maintaining active replicas using techniques like virtual synchrony [2] is less appealing due to its heavy resource requirements on CPU and network bandwidth [3]. Hence, we focus on achieving system reliability using passive (*primary-backup*) [4] replication, where we support two types of backups, *hot standby* and *cold standby*.

Providing a predictable timing bound to handle failures is essential to bound Time-To-Recovery (TTR). With bounded TTR, the system can be designed to be functional even under the presence of failures. We consider real-time tasks, where each task τ_i must complete its jobs within D_i units of time relative to their release times. According to their TTR requirements, we classify each task into one of three classes: (1) *Hard Recovery Task*, which should be able to recover and complete within its original deadline D_i, (2) *Soft Recovery Task*, which has more relaxed TTR requirements such as (say) $2D_i$ instead of D_i, and (3) *Best-Effort Recovery Task*, whose recovery is optional depending on the amount of available resources.

In this paper, we assume a *fail-stop* failure model [5]. In the fail-stop failure model, a failed component loses its capability of generating any data, and the lack of data from a task or hardware subsystem can be used to signify failure. Tolerating fail-stop failures on distributed embedded real-time systems is significant because fail-stop failures such as thermal shutdown tend to happen frequently.

With safety-critical real-time systems in mind, we target multiple goals. Most importantly, *no single point of fail-*

1052-8725/12 $26.00 © 2012 IEEE
DOI 10.1109/RTSS.2012.74

ure is permitted. In other words, a task/processor failure should not lead to system failure. Secondly, *failure recovery within a guaranteed duration* should be achieved. Since cyber-physical systems are usually tightly connected to the physical world, failure recovery without predictable timing behavior could return unpredictable results in the physical world. Apart from these two goals, *predictive fault discovery and notification, resource isolation, ease of use of abstraction, ease of application development, and sensor/actuator control* are other factors considered.

We propose a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features to tolerate fail-stop *processor* failures and *task* failures for distributed embedded real-time systems in a *timely* manner. To detect failures, SAFER monitors the health status and state information of each task and broadcasts the information. When a failure is detected using either *time-based failure detection* or *event-based failure detection*, SAFER reconfigures the system to retain the functionality of the whole system using task-level fault-tolerance techniques. More specifically, SAFER provides the following features: (a) Each task can have zero, one or more backup(s), (b) Each backup can be either a hot standby or a cold standby, (c) Failure detection and recovery latencies can be guaranteed, (d) A primary and each of its backup(s) are always allocated to run on independent processor boards to avoid common failure modes, (e) State transfer is managed for seamless recovery from failures.

To integrate SAFER with our fault-tolerant task allocation schemes [6, 7] and verify that the system works well with SAFER, we have implemented new features in a model-based development tool, SysWeaver developed at CMU [8]. Our analysis engine based on the formal timing analysis given in Section IV-E has been added to SysWeaver, and we have added a simulation capability of SAFER features under the presence of failures. Specifically, by injecting failures, we can simulate the timing behavior of the system and verify its operation with different models and system parameters.

SAFER has been implemented as a proof of concept on Ubuntu 10.04 LTS and deployed on Boss, an award-winning self-driving car developed at CMU [1]. We provide a case study showing the quality of mobile robotics algorithms running on the vehicle. The results are measured using the autonomous driving simulation scenarios used during the 2007 DARPA Urban Challenge. We also provide extensive measurement results in TTR and the overhead of SAFER.

The major contributions of this paper are as follows:

1) The design, implementation and evaluation of a real-time fault-tolerant distributed architecture, SAFER, to provide task-level fault-tolerance techniques[1].
2) The analyses of the worst-case timing behaviors of SAFER features.
3) Modeling of a system equipped with SAFER to analyze timing characteristics through a model-based design tool, SysWeaver.
4) A case study showing the quality of mobile robotics algorithms of Boss in the presence of failures after the SAFER layer is integrated.

[1]This contribution extends our previous work [9], where we described how a preliminary version of SAFER works. Since then, we have improved how SAFER 1) detects failures, 2) transfers state information of a primary for its cold standbys, and 3) recovers failures when cold standbys are used.

Organization: The rest of this paper is organized as follows. Section II presents the related work. Section III represents the overall system and the failure model. Section IV describes the architecture of SAFER and its implementation, and it also provides detailed analysis of the SAFER layer on failure detection and recovery time. Section V describes a modeling technique on SysWeaver to find proper system parameters for SAFER. The evaluation results and the case study on Boss will follow in Section VI and VII, respectively. We conclude our paper in Section VIII.

II. Related Work

Fault-tolerant distributed embedded systems have been extensively studied in the literature. The ISIS system [10] is a well-known software system that supports fault-tolerance services. FT-CORBA (Fault-Tolerant CORBA) [11, 12, 13] has been used in various applications to design and implement a fault-tolerant distributed system, and practical experiences on two different FT-CORBA infrastructures are described in [14]. CORBA-based fault-tolerant middleware services are also surveyed in [15]. There are other replication-based recovery services such as Arjuna [16], REL [17] and IFLOW [18], which are not based on CORBA. One clear distinction between the existing work and SAFER is that SAFER provides the framework to support timely failure recovery in a generic distributed embedded system.

There have also been efforts on building real-time fault-tolerant systems. MEAD [19] provides a proactive fail-over framework using a failure prediction method to overcome the unpredictable nature of failure occurrences and support somewhat predictable timing behavior. FLARe [20] is designed and implemented to support fault-tolerance for distributed soft real-time applications. SAFER differs from the above-mention systems in that SAFER is built on a publish-subscribe model rather than a client-server model. In addition, SAFER provides predictable timing characteristics of failure detection and recovery when real-time systems become SAFER-enabled. SAFER also provides a flexible failure detection and recovery infrastructure. SAFER detects failures using heartbeat signals (*time-based*) as well as OS (Operating Systems) signals (*event-driven*). A primary can use hot standby and/or cold standby as a backup.

Fault-tolerant scheduling in distributed embedded real-time systems has also been widely studied in [21, 6, 7, 22, 23]. Our fault-tolerant task partitioning algorithms, especially R-BATCH [6] and R-FLOW [7], are integrated into SysWeaver to support fault-tolerant task allocation and simulate the timing behavior of different allocations.

III. System and Failure Model

We assume that the system comprises p nodes communicating via messages over a network, where each node has a processor executing real-time periodic tasks each represented by (C_i, T_i, D_i), where each task τ_i computes for a maximum of C_i time-units every T_i time-units within a deadline D_i. Those tasks are scheduled under the DMS (Deadline-Monotonic Scheduling) [24] policy enforced by the Linux/RK resource reservation schemes on each processor [25]. Tasks use the Publish-Subscribe architecture to communicate with each other so that any task on the system can be configured to be recoverable. The network has an upper bound on message delivery and is completely connected. In other words, a message is eventually delivered within a known delay bound, and the network is assumed not to partition[2].

1) Failure Model: Tasks, processors or nodes on the system are subject to fail-stop failures, where they fail by crashing and do not generate incorrect outputs. In other words, tasks running on a live processor/node are assumed to always emit correct outputs. Therefore, in order for the system to continue to correctly operate, recovery and restoration processes might be required. These failures may also happen concurrently.

The system network may experience occasional omission failures, i.e., it may suffer from intermittent packet loss. This implies that the network does not fail completely. This again can be realized (say) by using redundant links.

2) Task Classification: Distributed embedded real-time systems run multiple tasks across different nodes with strict timing constraints. We provide support for task-level replication schemes when processors or tasks fail in a fail-safe manner. In [7], we have investigated an application flow to consider the end-to-end time delay of a chain of distributed real-time applications. To meet the timing requirements of different application flows, we have to consider several components such as time delay on each task, node and communication link. We aim to limit the fail-over time on each task to yield a reliable system. Depending on the fail-over time requirement, we classify tasks into three different categories [6]: *Hard Recovery Task*, *Soft Recovery Task* and *Best-Effort Recovery Task* defined in Section I. Particularly, a hard recovery task is the most challenging because the task may have little time left for re-execution to meet its original deadline if the failure occurs near the end of the current job.

3) Hot Standby and Cold Standby: We use passive replication (primary-backup [26]), which has been mostly used for soft real-time systems [20]. To support the hard recovery tasks described above, we provide two different types of backups, *hot standby* and *cold standby*:

- *Hot standby*: A hot standby runs concurrently with its primary. Depending on the reliability requirements, multiple hot standbys may coexist on different nodes. Only the primary emits its outputs. Its hot standbys simultaneously run on different nodes, receive the same input as the primary, but they do not generate any outputs. When a primary fails, one of the hot standbys is promoted to be the primary and starts generating outputs. Since it has been already running, only interface redirection from a null output device to the active channel needs to happen. This enables the use of hot standbys to recover hard recovery tasks[3].
- *Cold standby*: A cold standby is a dormant task which is triggered to run when a failure is detected. When its primary is running, its binary resides in the system memory, and it does not consume any CPU resources; however, state information from the primary task computations are periodically sent to its cold standby node(s). On failure of the primary or specified number of hot standby, the cold standby becomes active and starts from the last check-pointed status. The cold standbys can be used for recovering soft recovery tasks and best-effort recovery tasks.

Our major goal of SAFER is to provide ρ_i backups for safety-critical real-time tasks to tolerate ρ_i fail-stop failures, which may happen concurrently. One way of determining ρ_i for all tasks is described in [6].

IV. The Architecture of SAFER

The overall architecture of the SAFER layer is illustrated in Figure 2. The SAFER layer is composed of SAFER daemons, one running on each processor, and a library supporting a task execution environment. The library enables any task launched on the SAFER layer to be periodically executed, with configurable parameters. The daemons have a master-slave architecture, and the master SAFER daemon controls the slave SAFER daemons responsible for managing tasks on each node and monitoring its health status. The configurable parameters for each task are given to the library when the task is launched by a SAFER daemon. For the underlying communication layer, an inter-process communication primitive can be used.

SAFER uses multiple techniques for recovering from processor or task failures. For a task failure without a processor failure, SAFER tries to re-spawn the failed task several times. If the task fails to run, the re-spawning process can be restarted with a different software state or with new inputs because the failure might have been caused by the current software state or/and the current inputs. This re-spawning process can be configured for different applications. The SAFER layer also supports task-level replication techniques, where *selective* tasks on failed processors are recovered on other live processors. Replicas must therefore be placed on independent nodes, a constraint that is referred to as

[2]We are currently exploring network architectures where redundant links will make network partitioning highly unlikely. Such network redundancy is the topic of ongoing study and is beyond the scope of this paper.

[3]A time synchronization service is important to support hot standbys. SAFER has its own time synchronization module, which will be described in Section IV.

a *placement constraint* [6]. The major benefit of using selective task-level recovery is its flexibility. Since we can selectively recover tasks, we can increase the reliability of highly critical tasks by adding more hot/cold standbys for those tasks. We can also efficiently manage the available computing resources by not replicating less-critical tasks, thus enabling an affordable solution.

SAFER provides process group management which is applicable to the SAFER daemons and application tasks. SAFER has two different types of groups: one that is formed by the SAFER daemons and the other formed by each application. The SAFER daemons have a master-slave relationship and manage tasks on each machine. Hence, when the master SAFER daemon fails, one of the slave SAFER daemons will be promoted to become the master SAFER daemon. Also, any application task that has at least one hot standby forms a group that includes the primary and its hot standbys. When the primary fails, one of the hot standbys will become the primary. Therefore, one generic group management method can be used for the SAFER daemons and application tasks in the system.

A. Group Membership Protocol

The process group management is done using a group membership protocol [27, 28]. Our group membership protocol is implemented as a separate thread in the SAFER library used by both the SAFER daemons and application tasks. The membership protocol of SAFER is specifically designed to provide predictable timing behavior and deterministic recovery times. In this membership protocol, all members send heartbeat messages to the master, where the master of a group can be either the master SAFER daemon or the primary of an application group. The master broadcasts its own health status including the list of group members to the group members. Based on this information, only the master will decide who is in the group. The master can detect the failure of a group member if no heartbeat messages are received from that member. The failure of the master can be detected by the group members due to the absence of status messages, and one of the group members will be promoted to become the master by following a pre-determined sequence of group members. If the failed master rejoins the group, it will broadcast a message to obtain group information and send a message to the current primary.

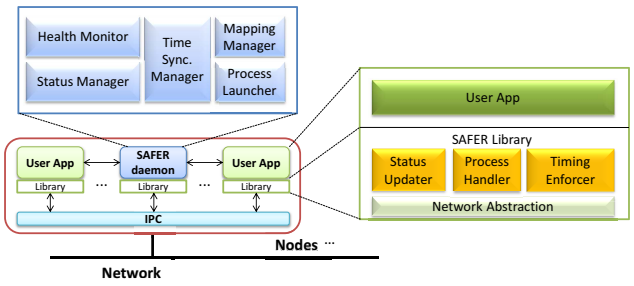


Figure 2. The overall architecture of SAFER

After successful state transfer, the current primary will be demoted, and the rejoined primary takes the role of the primary. If a failed group member rejoins, it will be detected through its heartbeat messages by the current primary, which in turn will add it to the group. These steps are important to provide consistent group management when simultaneous failures occur. Suppose an application group is composed of a primary and two hot standbys. If the primary fails and the heartbeat messages from the first hot standby to the second hot standby are lost, each hot standby may try to become the primary. When they see each other, the pre-determined sequence decides who has precedence. Therefore, we can guarantee that we have only one primary in the group, hence maintaining consistency.

B. The SAFER Library

As illustrated in Figure 2, the SAFER library is a task execution environment composed of a status updater, process handler, timing enforcer and network abstraction. User threads and the SAFER daemons run on top of the library.

1) Status Updater: The SAFER group membership protocol is implemented as a separate thread in the status updater. The status updater supports task-level replication techniques by managing state information between the primary (the master SAFER daemon) and its backups (the slave SAFER daemons). The role of the status updater changes based on whether a task it monitors is a primary (e.g. the master SAFER daemon) or a backup (e.g. the slave SAFER daemon). The status updater of the primary task periodically sends a heartbeat message, its internal state information and the list of group members to its hot/cold standbys, where the update period is configurable. The status updater at the backup node sends out heartbeat signals to the primary such that the primary can decide who is in the group. The rejoining process is also dealt with by the status updater. Therefore, the status updater enables the members in the group to agree upon the availability of each node.

2) Process Handler: The process handler of the SAFER library promotes a backup to be the primary when it receives the corresponding request from the master SAFER daemon or the status updater. When a backup is promoted, the new primary starts generating outputs for use and confirms its promotion to the requester. It must be noted that a hot standby is always running and its outputs are filtered by the network abstraction of the library in Figure 2 under the control of the process handler.

3) Timing Enforcer: The timing enforcer enables tasks to have guaranteed and protected access to required processing resources in a timely manner based on Linux/RK [25]. In Linux/RK, a shared resource is reserved and enforced by the following parameters: computation time C every T time-units within a deadline D. We refer to these parameters $\{C, T, D\}$ as explicit parameters of our reservation model. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval.

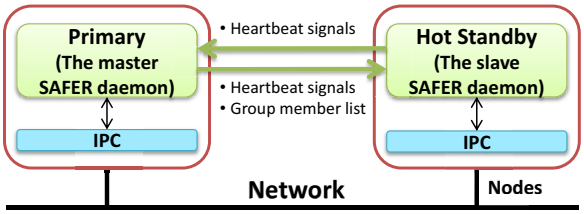


Figure 3. The primary-backup architecture

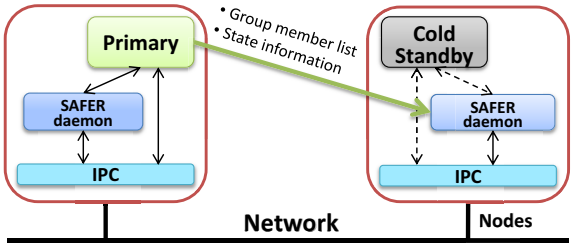


Figure 4. The cold standby operation

C. The SAFER Daemon

As illustrated in Figure 2, a SAFER daemon is composed of a health monitor, a status manager, a time synchronization manager, a mapping manager and a process launcher.

1) Health Monitor: The health monitor of the master SAFER daemon monitors the health status of the other daemons and their processors. Once the daemon detects and notifies the failure of a processor, other SAFER daemons can trigger the recovery procedures using the process launcher and process handler unless it is already recovered by the hot standbys of the tasks running on the failed processor.

2) Status Manager: The status manager tracks the current status of tasks running on its own node and issues failure notification if there is a task failure (say due to a segmentation fault) by capturing the OS signal, which may trigger the failed task re-spawning process on the local node. If there is any cold standby on the processor where the daemon runs, the status manager stores the state information of that cold standby from its primary as depicted in Figure 4. It should be noted that the daemons themselves cannot have cold standbys because the daemon should operate as long as the processor is alive.

3) Time Synchronization Manager: The SAFER layer offers a global time service using a service similar to NTP [29] used for time synchronization over the Internet. The master SAFER daemon behaves as a time server, and each slave becomes a client for this service and listens to messages from the time server. This service is essential to synchronize all the daemons so that failure recovery occurs within the given timing requirement[4] between the primary task and its hot/cold Standbys. This also enables the timing enforcer of the SAFER library to have a smaller penalty in resource scheduling.

4) Process Mapping Manager and Launcher: The process mapping manager and launcher are responsible for automatically deploying tasks on the nodes of the SAFER layer based on system configuration parameters. The system configuration includes information about where tasks are allocated and the resource requirements of each task. It also contains the location of the primary and hot/cold standbys if the tasks are selected to have backups. The process mapping manager maintains and updates the system configuration information. Changes to this information can occur due to processor failures, resource demand changes, task completions, and so forth. Based on the up-to-date information from the process mapping manager, the process launcher loads tasks on different processors. The process mapping manager and launcher can be connected to a user-interface application that provides a global view of the system with the current health status of each task on each node. As an example, the information from the process mapping manager and launcher are visualized on TROCS [30], the operator interface of Boss [1].

[4]Please see Section IV-E.

D. Failure Detection and Recovery

Heartbeat signals from the status updater of each task will be used for detecting task/processor failures. The status updater of the SAFER library at a backup node will decide the failure of the primary if heartbeat messages of its primary are missed a specified number of times. We call this failure detection scheme as *time-based failure detection*. A task failure may be directly detected by the status manager of the SAFER daemon by catching a signal generated by the OS when a task has unexpectedly failed. Then, an appropriate recovery will be initiated. We name this failure detection scheme as *event-driven failure detection*. It should be noted that event-driven detection cannot be used for processor failure detection.

The recovery from a failure is done by using either the task re-spawning process or the task-level replication techniques of hot and cold standbys. The re-spawning process relaunches the failed task with a different software state and/or a random back-off time to apply new inputs, which could be different from the inputs causing the failure. This process can be suspended by either the limited number of retries or the task recovery, where the task recovery can be achieved by either the re-spawning process or one of the backups on different processors.

A hot standby receives the same inputs as the primary with no failure, and the user threads of the hot standby run normal operations except that the output from them is filtered by the network abstraction[5]. In the presence of any task failure detected by a hot standby, one of the live backups will promote itself to become the primary based on the predetermined precedence information. Then, it will send a notification to the SAFER daemons.

In the case of a cold standby, without a failure, a cold standby node daemon periodically receives and stores the

[5]We do not generate outputs from hot standbys because we assume the fail-stop failure model. To relax the failure assumption model so that we can check if the outputs from the primary are valid, the network abstraction can be modified to compare the results of the primary with the results of its hot standbys.

state information of the primary coming from the status updater of the primary. The disadvantage of using a cold standby is that the recovery latency could be long when there is a failure detected by the master SAFER daemon. Conversely, since it runs only on demand, it saves computing resources in the absence of failures.

E. Worst-Case Analysis and Admission Control

Assigning a proper standby to a task is of vital importance to meet the fail-over requirements of the given set of tasks. Regardless of the detection methods we provide, we have to consider the worst-case behavior to meet the fail-over requirements. We explicitly consider the time-based failure detection method. Event-based failure detection takes no longer than time-based failure detection and will not be analyzed directly.

For this analysis, we assume a synchronous network [27], and we define a time delay d to represent the maximum network delay of the heartbeat signal packets. We let $T_{heartbeat}$ denote the interval between two consecutive heartbeat signals. Then, the status updater of the SAFER library at a backup node decides the death of the primary unless it hears a heartbeat signal from a processor within $d + kT_{heartbeat}$, where k is an adjustable positive integer based on the underlying communication medium and protocol.

We consider a set of tasks, Γ composed of n tasks, τ_1, τ_2, ..., τ_n. Each task is in one of three sets: *Hard Recovery Task* set, Γ_H, *Soft Recovery Task* set, Γ_S, and *Best-effort Recovery Task* set, Γ_B. A task τ_i is represented by (C_i, T_i, D_i, μ_i) [6]. τ_i will compute for a maximum of C_i time-units every T_i time-units within a relative deadline D_i, and μ_i denotes the ratio of *recovery instance* to its deadline. The recovery instance is defined as the time instance when the failed job is completed by the new primary, which used to be the backup. For example, if $\tau_i \in \Gamma_H$ or $\mu_i = 1$, the failed job should be recovered within D_i. R_i denotes the worst-case response time of τ_i in the absence of a failure, and we assume an implicit deadline of $D_i = T_i$ in this paper. s_i denotes the worst-case slack-to-recovery time of τ_i, i.e., the time duration between the failure and the recovery instance. If τ_i runs on processor, P_l, the response time of τ_i on P_l is represented as $R_i^{P_l}$. τ_i can have $n^H(\tau_i)$ hot standbys and $n^C(\tau_i)$ cold standbys. Then, $\tau_{i,j}^H$ denotes the j^{th} hot standby of τ_i, and $\tau_{i,k}^C$ is the k^{th} cold standby of τ_i. Either $\tau_{i,0}^H$ or $\tau_{i,0}^C$ can represent τ_i. We also have a set of processors, P, composed of p processors, P_1, P_2, ..., P_p, and $\tau_i \in P_l$ means that τ_i is running on P_l. Let Π represent the allocation information. We use Π_{ij}^H to represent the processor allocated to τ_{ij}^H. Π_{ik}^C is the processor that contains τ_{ik}^C.

On SAFER, the following theorems are satisfied.

Theorem 1: Given τ_i and its $n^H(\tau_i)$ hot standbys, at least one of the hot standbys will detect and recover[6] the failure (only) of the primary if $T_{heartbeat} \leq \frac{(s_i - d)}{k}$, where d is a network delay. The primary is marked as failed if k consecutive heartbeat signals are missing.

Proof: The worst case of recovering a failure happens when a task fails just before the released job completes. Under this circumstance, the slack-to-recovery time is minimized as $s_i = \mu_i T_i - R_i^{\Pi_i}$. Then, the worst-case slack-to-recovery time s_i should be greater than the failure detection time, $d + kT_{heartbeat}$ so that one of the hot standbys becomes the primary and sends out the computed outputs from the failed job. Hence, $T_{heartbeat} \leq \frac{(s_i - d)}{k}$ is satisfied. ■

Theorem 2: Given τ_i and its $n^C(\tau_i)$ cold standbys, at least one cold standby will detect and recover the failure of the primary if $T_{daemon} \leq \frac{(s_i - R_i^{\Pi_{i,j}^C} - d - d_S)}{k}$, where T_{daemon} is the period of the SAFER daemon, d_S is the time required for copying and processing the state information from the daemon and j is determined based on the precedence sequence.

Proof: A similar proof to that of Theorem 1 can be applied. First, we have to consider the response time of the cold standby on $\Pi_{i,j}^C$ because the cold standby will start running after the node receives the fault notification. This will further decrease s_i. Also, the period of the daemon

Algorithm 1 System-Schedulability-Test-(Γ, P, Π)

Input: Γ: a taskset, P: a set of processors and Π: allocation information between Γ and P
Output: Schedulability of Γ on P with Π

```
for i = 1 to n do
    ▷ Do the response time test for τ_i and its standbys
    R_i ← the response time of τ_i on Π_i.
    if R_i ≤ D_i then
        ▷ The primary is schedulable, so check its standbys
        for j = 1 to n^H(τ_i) do
            R_j ← the response time of τ^H_{i,j} on Π^H_{i,j}.
            if R_j ≤ D_i then
                ▷ This hot standby is schedulable.
        ▷ Check its slack-to-recovery time
        if R_i ≤ μ_i D_i − kT_heartbeat − d then
            ▷ The primary and hot standbys are recoverable
        for j = 1 to n^C(τ_i) do
            R_j ← the response time of τ^C_{i,j} on Π^C_{i,j}.
            if R_j ≤ D_i then
                ▷ This cold standby is schedulable.
                ▷ Check its slack-to-recovery time
                if R_i ≤ μ_i D_i − R_j − kT_heartbeat − d − d_S then
                    ▷ Mark this cold standby recoverable
if all tasks schedulable and recoverable then
    return TRUE
else
    return FALSE
```

[6]In order to fully recover within the given value in this theorem, the hot standby should always keep the outputs until the next period starts. Otherwise, the recovery can be off by the period of the hot standby.

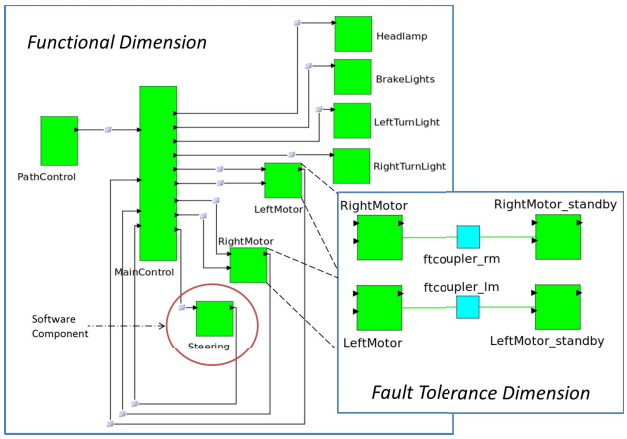


Figure 5. Fault-Tolerance Dimension in SysWeaver

should be applied instead of $T_{heartbeat}$ from the SAFER library, since the cold standby is managed by the SAFER daemons. Then, $T_{daemon} \leq \frac{(s_i - R_i^{\Pi^C_{i,j}} - d - d_S)}{k}$ is satisfied. ■

These two theorems provide significant information to task partitioning algorithms because these properties should be applied to the admission test of hot standby and cold standby along with the relevant schedulability-based admission tests. Based on these two theorems, a schedulability test is given in Algorithm 1.

V. SysWeaver Integration

SysWeaver is a model-based design, integration, and analysis framework introduced by de Niz et. al [8] for embedded real-time systems. It explicitly captures the para-functional behaviors such as timeliness and dependability, and their impact on the functional aspects of a system. It uses a view-based representation of various system dimensions such as Functional, Timing, Fault-Tolerance and Deployment. Different domain experts can work on each of these different dimensions, while SysWeaver resolves dependencies among the different views automatically. The framework enables the use of analysis plugins and task-level system simulation capabilities to evaluate and verify system properties along with automatic system code generation. This provides the ability to automatically generate distributed system-oriented glue code which ties together the distributed functional code.

We use SysWeaver to model and capture the SAFER fault-tolerance properties of the system in the Fault-Tolerance Dimension. This view enables us to model the fault-tolerance strategy as well as give us a component-level mapping of the backups running in the system. Figure 5 shows an example of the Fault-Tolerance Dimension in SysWeaver. A system model is created within SysWeaver and we then use fault-tolerance task allocation algorithms described in our previous work [6, 7] to provide a fully deployed system.

As part of the verification of the SAFER layer, we have implemented the theoretical analysis as an analysis plugin in SysWeaver. Using the system properties represented in the different dimensions, the analysis engine provides an evaluation of the fault-tolerance techniques and analytic results that are then fed back into the system model. To evaluate these analytical results, we then use the distributed system simulation engine within SysWeaver. The simulation engine uses task level properties such as execution time and task priorities along with communication delays to simulate the system response. A fault injection framework has been integrated into the simulation engine that enables us to inject faults and evaluate fault-based mode changes for tasks and captures system response times and deadlines of primary tasks in the presence of faults. The analytical results from SysWeaver for SAFER will be shown in the next section.

VI. Evaluation

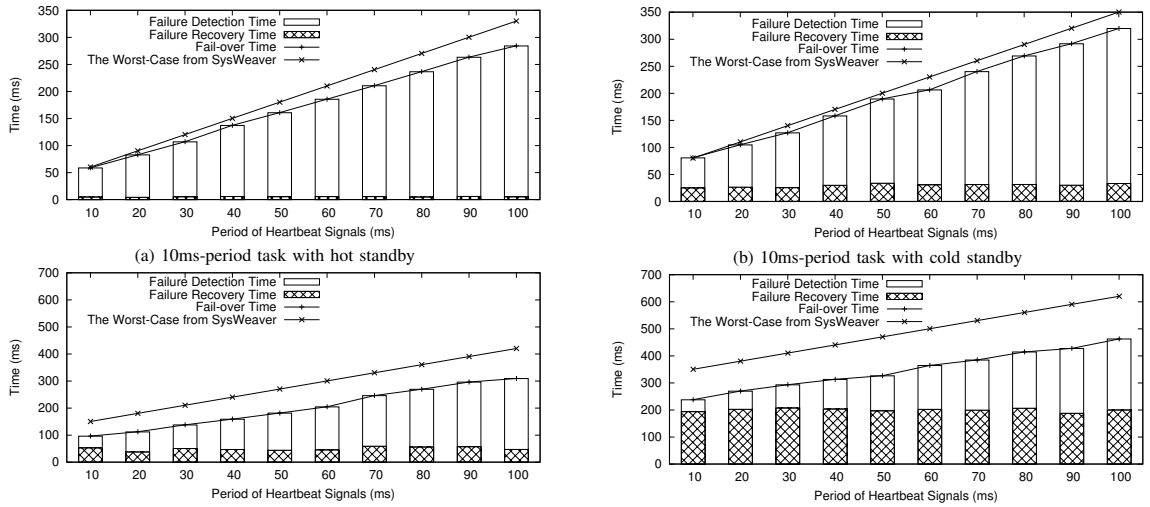
A proof-of-concept implementation of SAFER runs on Linux and x86 hardware. SAFER is deployed and has been running on Boss [1]. To measure the performance of the SAFER layer with the presence of a failure, we have built a cluster composed of three Intel Quad-Core machines. We ran a scenario used to test Boss during the competition in 2007 without the perception system. The artificial intelligence algorithms for behavior and planning along with vehicle control were run on the cluster. By injecting processor failures through a script, we measured the fault detection and fault recovery times for different tasks with different periods. The fail-over time is the summation of failure detection time and recovery time. For a hot standby, the fault detection time is the time duration between when a failure happens and when the backup detects the failure. Its fault recovery time is the time duration between when the backup detects the failure and when the failed task is completely recovered. For a cold standby, the fault detection time is the time duration between when a failure happens and when the master SAFER daemon detects the failure; its fault recovery time is the time duration between when the master SAFER daemon detects the failure and when the failed task is completely recovered, where the recovery procedure includes state information copy, initialization and one re-execution.

For the measurements, we chose two tasks, `BehaviorTask` and `LocalPlannerTask`, which have $10ms$ and $100ms$ as their respective periods. Since Boss has a set of harmonic tasks composed of 10ms-period tasks, 50ms-period tasks and 100ms-period tasks, those two tasks are appropriate to understand the behavior of Boss with SAFER. We measured the fail-over time with a hot standby and a cold standby[7] with two different detection methods, time-based detection and event-based detection. Each point corresponds to the average of 50 iterations.

A. Time-based Failure Detection

Figure 6 shows the fail-over time measurements when the time-based detection is used for detecting a failure,

[7]The fail-over time measurements with the re-spawning process are not described in this paper because the fail-over time with cold standby is longer than the time with the re-spawning process due to the network delay. If the process fails to recover the failure after several retries, it could take longer; however, it is beyond the scope of this paper.



(a) 10ms-period task with hot standby

(b) 10ms-period task with cold standby

(c) 100ms-period task with hot standby

(d) 100ms-period task with cold standby

Figure 6. Fail-over time measurements when time-based detection is used

while the period of heartbeat signals from the primary varies from $10ms$ to $100ms$. The hot standby (the master SAFER daemon for cold standby) declares a failure if it misses three continuous heartbeat signals from the primary with a delay d of $20ms$ that includes network and node-side delay. From the data in the figures, it is seen that the failure detection time highly depends on the period of the heartbeat signals of the primary. The failure detection time linearly increases as the period of heartbeat signals increases because the waiting time from the backup linearly increases. The worst-case of the failure detection time calculated in SysWeaver is also depicted. As shown in the figures, our measurements in average case are under the worst-case.

The recovery time of a task is related to its task period because the process handler of its hot (cold) standby should be able to receive the command from the status updater thread (the master SAFER daemon). For the cold standby measurements, the recovery time is much longer because of state recovery and re-execution to recover the failure. The state recovery could depend on each application. For example, the state recovery of the 100ms-period task (`LocalPlannerTask`) in Figure 6d takes a long time because the task has a large amount of state information and it has a long initialization sequence.

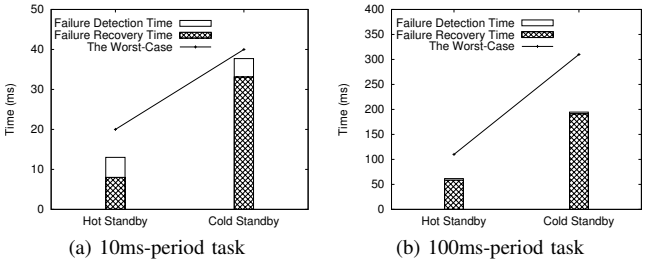
B. Event-based Failure Detection

Figure 7 shows the measurements when event-driven detection is used when the period of SAFER daemon is $10ms$. Since the local SAFER daemon detects local task failure, the failure detection time is hugely reduced. The local SAFER daemon captures the signal from the task with failure and reports it to the master SAFER daemon. Therefore, the failure detection time is measured as $5ms$ in average. If the period of SAFER daemon increases, the failure detection time will also be increased. The recovery time of a task is related to its period as shown previously. The worst-case from SysWeaver is also depicted as solid lines in Figure 7a and 7b.

C. Overheads of SAFER

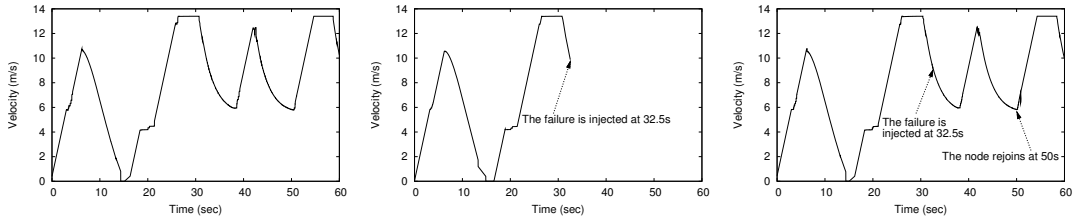
We have analyzed the worst-case timing behavior using SysWeaver and measured the average-case timing behavior on our simulation cluster. We must also consider the overheads such as additional CPU utilization, network bandwidth and memory consumption that are imposed by the SAFER services. When hot standbys are used, the CPU utilization increases linearly as the number of hot standbys increases. Hot standbys also add to the network load by sending regular heartbeat signals. We limited the size of a heartbeat message to 132 $bytes$ in order to minimize the impact of heartbeat messages. Since the failure detection time heavily depends on the period of heartbeat messages, the trade-off between the network load and the failure detection time should be considered. In the case study described in Section VII, for example, 25.78 $Kbps$ and 2.578 $Kbps$ of additional network load are added for `BehaviorTask` and `LocalPlannerTask`, respectively.

The network load is also increased by state transfer requirements for cold standbys. Minimizing the amount of data to transfer reduces network congestion and is aided by the support provided by SAFER to easily configure the specific state information to transfer. The period of data transfer is



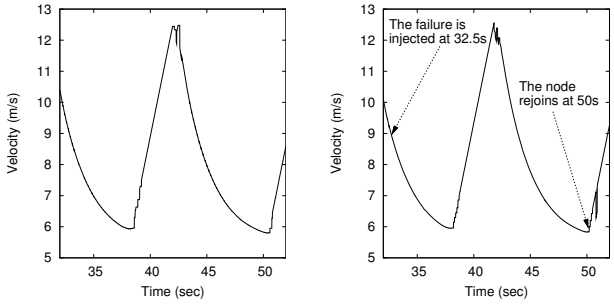
(a) 10ms-period task (b) 100ms-period task

Figure 7. Fail-over time measurements when event-based detection is used



(a) Normal velocity trace without a failure injection (b) Failure injected at $32.5s$ with SAFER disabled (c) Failure injected at $32.5s$ with SAFER enabled

Figure 8. The velocity trace of Boss measured from `ControllerTask`



(a) No failure injection (b) With SAFER enabled

Figure 9. The scaled version of Figure 8a and 8c

another dimension to consider. If cold standbys are used in the case study, 112.89 $Kbps$ and 41.29 $Kbps$ of the network load are added when 1024 $bytes$ for `BehaviorTask` and 4096 $bytes$ for `LocalPlannerTask` are required for state transfer. Furthermore, the SAFER daemons storing state information from primaries also consume more CPU resources. The computation time of daemons increases by 2 to 5 percent[8] depending on the number of cold standbys that each SAFER daemon manages. Cold standbys also consume memory resources even if they are dormant in system memory.

VII. Case study on Boss

The SAFER layer has been heavily tested on Boss. As a case study, we show results from a scenario used during the 2007 DARPA Urban Challenge. The scenario uses the layout of our test track located at Robot City in Hazelwood, Pittsburgh, PA, where we test our self-driving car at intersections, stop signs, U-turns, two-lane roads, curvy roads and parking lots. In the scenario illustrated in Figure 10, Boss will start from the point depicted at the bottom left of the figure. Boss will follow the road, cross an intersection governed by stop signs, increase the velocity while following the straight road and make turns on the curves. The same scenario file is also used on the real vehicle, but only the tasks that interact with the sensors are replaced with the simulated tasks.

The scenario is composed of nine tasks, excluding the SAFER daemons: `BehaviorTask`, `ControllerTask`, `LocalPlannerTask`, `MissionPlannerTask`, `RoadBlockageDetector`, `Planner3DTask_1`, `Planner3DTask_2`, `RobotClient` and `ServerTask`. `MissionPlannerTask` decides where to go. `BehaviorTask` decides the behavior such as turning, lane changing and intersection traversal, and `LocalPlannerTask` sends commands to the vehicle actuators such as accelerator, brake and steering wheel. `ControllerTask` receives those actuator commands and directly interfaces with the vehicle hardware. On the simulation cluster, this task runs in simulation mode. `Planner3DTask_1` and `Planner3DTask_2` are mainly used in unstructured driving conditions such as parking and are therefore not heavily used in this scenario. To test SAFER with this scenario, we replicated `BehaviorTask` and `LocalPlannerTask` using hot standbys. The period of heartbeat signals is fixed at $10ms$, and the period of the SAFER daemon is also $10ms$. We run the scenario while Boss navigates the map. In reference to Figure 8, we inject a failure by disconnecting one of the three cluster machines at $t = 32.5s$ after Boss turns left at the intersection. Then, the disconnected machine rejoins the cluster at $t = 50s$ when Boss is at the top left corner.

Figure 8 shows the velocity profile of Boss measured from `ControllerTask`. The velocity is an important variable to visualize the behavior of a self-driving car. During normal operations, as depicted in Figure 8a, Boss stops at the intersection controlled by the stop sign at $t = 15s$, hence the velocity becomes zero. The velocity graph shows valleys when Boss decreases its speed before and along curves at $t = 39s$ and $t = 50s$. Boss then increases its speed along the straight road. The velocity cannot be sent to `ControllerTask` without `BehaviorTask` and `LocalPlannerTask` running. With SAFER disabled, therefore, Boss completely stops when a node failure is injected as depicted in Figure 8b. Figure 8c shows the case with SAFER enabled. The node failure is injected at $t = 32.5s$, and the node rejoins at $t = 50s$. As can be seen, there is very little if any behavioral difference at the

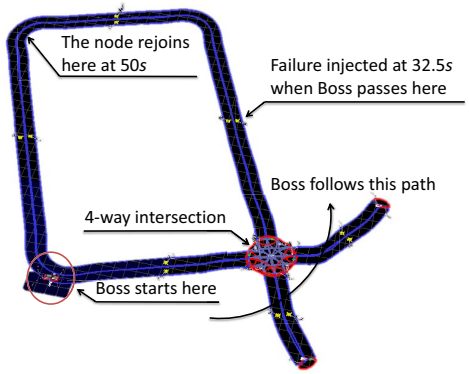


Figure 10. The map Boss follows during the simulation

[8]This value could be different on different configurations.

level of driving. At a microscopic level, the graphs zoomed into the time interval $[32, 52]$ are shown in Figure 9. It can be seen that the differences between Figure 9a and 9b are subtle as SAFER detects and recovers from the failure within the original deadline. However, the velocity profile shows a small amount of jitter during the process rejoining step as the new primary becomes hot standby again when the old primary rejoins.

We have leveraged the benefits of SAFER on the vehicle. Boss is equipped with various sensors, but the Velodyne HDL-64E is the most critical sensor on the vehicle due to its wide field of view (360° horizontal by 26.8° vertical). This provides enough data to reconstruct a good three-dimensional view of the world around the vehicle. After the competition, we saw that the processing board running the task for Velodyne frequently crashed. With the SAFER layer in place, this undesirable situation can be avoided. We also sometimes suffered from over-heated processing nodes in the harsh environment. When the densely deployed processing nodes are over-heated, cooling time is required not to damage the hardware. SAFER is also able to handle this unwanted event. Although SAFER is a generic framework, it may be limited by physical constraints. For example, a machine that is used exclusively to interface with a sensor may not be recovered even with SAFER when the machine fails.

VIII. Conclusion

We have proposed a layer called SAFER (System-level Architecture for Failure Evasion in Real-time applications) to incorporate configurable task-level fault-tolerance features using hot standbys and cold standbys in order to tolerate fail-stop processor and task failures for distributed embedded real-time systems. SAFER is implemented on Ubuntu 10.04 LTS and integrated into a self-driving car developed at Carnegie Mellon. The formal analyses of the worst-case timing behavior are also provided, and our analysis engine is integrated with a model-based design tool, SysWeaver. We have presented measurements along with analytical results from the driving simulation scenarios used during the 2007 DARPA Urban Challenge. A case study on Boss has also shown that there is no noticeable behavioral difference even when node failure is injected and the failed node later rejoins. Future work to be done includes supporting graceful degradation based on load and resource changes and providing a generic infrastructure to recover from different types of sensor and actuator failures.

References

[1] C. Urmson, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, 25(1):425–466, June 2008.

[2] K. Birman, et al. Exploiting virtual synchrony in distributed systems. In *the 11th ACM symposium on operating systems principles*, 1987.

[3] M. Broy. Challenges in automotive software engineering. In *Proc. of the 28th International Conference on Software Engineering*, 2006.

[4] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[5] R. Schlichting and F. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[6] J. Kim, et al. R-BATCH: Task partitioning for fault-tolerant multiprocessor real-time systems. In *Proc. of the 10th IEEE International Conference on Computer and Information Technology (CIT)*, 2010.

[7] J. Kim, et al. An autosar-compliant automotive platform for meeting reliability and timing constraints. In *SAE 2011 World Congress*, 2011.

[8] D. de Niz, et al. Model-based development of embedded systems: The sysweaver approach. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[9] J. Kim, et al. SAFER: System-level Architecture for Failure Evasion in Real-time Applications. In *Proc. of the 4th Workshop on Adaptive and Reconfigurable Embedded Systems*, 2012.

[10] K. Birman. Replication and fault-tolerance in the isis system. In *Proc. of the 10th ACM symposium on operating systems principles*, 1985.

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

[12] B. Natarajan, et al. DOORS: Towards high-performance fault tolerant CORBA. In *Proc. of the IEEE International Symposium on Distributed Objects and Applications*, 2000.

[13] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proc. of the USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX Association, 1995.

[14] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, pages 467–511, 2004.

[15] S.M. Sadjadi and P.K. McKinley. A survey of adaptive middleware. *Michigan State University Report MSU-CSE-03-35*, 2003.

[16] G.D. Parrington, S.K. Shrivastava, S.M. Wheater, and M.C. Little. The design and implementation of arjuna. *Computing systems*, 1995.

[17] T. Friese, et al. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Systems Aspects in Organic and Pervasive Computing*. 2005.

[18] Z. Cai, et al. Utility-driven proactive management of availability in enterprise-scale information flows. In *Proc. of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, 2006.

[19] P. Narasimhan, et al. MEAD: support for Real-Time Fault-Tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, 2005.

[20] J. Balasubramanian, et al. Adaptive failover for real-time middleware with passive replication. In *Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009.

[21] J.J. Chen, et al. Real-time task replication for fault tolerance in identical multiprocessor systems. In *Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[22] P. Emberson and I. Bate. Extending a task allocation algorithm for graceful degradation of Real-Time distributed embedded systems. In *Real-Time Systems Symposium, 2008*, pages 270–279, 2008.

[23] J. Balasubramanian, et al. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[24] N.C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[25] S. Oikawa, et al. Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In *Proc. of the 5th IEEE Real-Time and Embedded Technology and Applications Symposium*, 1999.

[26] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[27] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 1991.

[28] R. Rajkumar and M. Gagliardi. High availability in the real-time publisher/subscriber inter-process communication model. In *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS)*, 1996.

[29] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991.

[30] M. McNaughton, et al. Software infrastructure for an autonomous ground vehicle. *Journal of Aerospace Computing, Information, and Communication*, 5(1):491 – 505, December 2008.